Compiladores

CIÊNCIA DA COMPUTAÇÃO

Prof. Dr. João Paulo Aramuni

UNIVERSIDADE FUMEC

Sumário

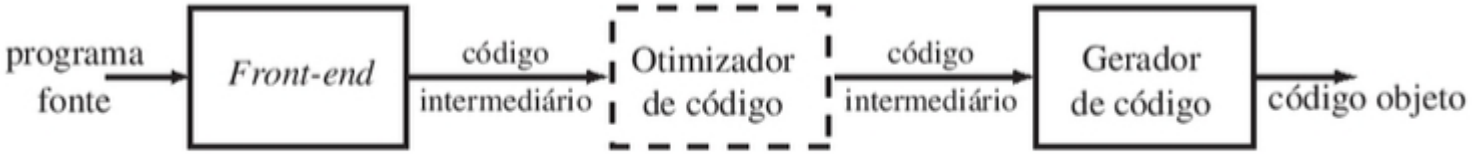
* **Geração de Código**
  * Introdução
  * Seleção das instruções de máquina
  * Definição de registros e de blocos sequenciais de código
  * Otimizações de fluxo de controle

Geração de Código

* **Introdução**
  * A última fase em nosso modelo de compilador é o **gerador de código**.
  * Ele recebe como entrada a representação intermediária (RI) produzida pelo *front-end* do compilador, juntamente com as informações relevantes da tabela de símbolos, e produz como saída um código objeto semanticamente equivalente à entrada, como mostra a Fig. 1 a seguir.

Geração de Código

* **Introdução**
  * Abaixo, Fig. 1:



Posição do gerador de código.

  * Os requisitos impostos sobre o gerador de código são severos. O código objeto precisa preservar o significado semântico do programa fonte e ser de alta qualidade, ou seja, precisa usar efetivamente os recursos disponíveis da máquina destino. Além do mais, o próprio gerador de código precisa ser executado eficientemente.

Geração de Código

* **Introdução**
  * O desafio é que, matematicamente, o problema de gerar um código objeto ótimo para determinado programa fonte é indecidível; muitos dos subproblemas encontrados na geração do código, tais como a alocação de registradores, são computacionalmente intratáveis.

  * Na prática, temos de nos contentar com técnicas heurísticas que geram um código bom, mas não necessariamente ótimo.

Geração de Código

* **Introdução**
  * Felizmente, as heurísticas estão suficientemente maduras para que um gerador de código projetado cuidadosamente possa produzir código várias vezes mais rápido que o código produzido por um gerador ingênuo.

  * Compiladores que precisam produzir códigos objetos eficientes incluem uma fase de otimização antes da geração do código. O otimizador mapeia a representação intermediária em outra, a partir da qual é possível gerar um código mais eficiente.

Geração de Código

* **Introdução**
  * Em geral, as fases de otimização e geração de código de um compilador, chamadas de *back-end*, podem exigir várias passadas pela representação intermediária antes de gerar o código objeto.

  * A otimização do código será discutida adiante na AULA08.

Geração de Código

* **Introdução**
  * Um gerador de código é composto por três tarefas principais: **seleção de instrução, alocação e atribuição de registrador**, e **escalonamento de instrução.**
    * A seleção de instrução compreende a escolha de instruções apropriadas da arquitetura alvo para implementar os comandos da representação intermediária.
    * A alocação e a atribuição de registrador decidem que valores devem ser mantidos em registradores e também quais registradores usar.
    * O escalonamento de instrução envolve a decisão sobre a ordem em que a execução das instruções deve ser escalonada.

Geração de Código

* **Introdução**
  * Veremos os algoritmos utilizados pelos geradores de código para traduzir uma representação intermediária para uma sequência de instruções da linguagem objeto, em arquiteturas com registradores simples.

  * Após discutirmos sobre os aspectos mais amplos no projeto de um gerador de código, estudaremos os tipos de código objeto que um compilador precisa gerar para dar suporte às abstrações incorporadas em uma linguagem fonte típica. Veremos também como os nomes na representação intermediária podem ser convertidos para endereços no código objeto.

Geração de Código

* **Introdução**
  * Muitos geradores de código dividem as instruções da representação intermediária em 'blocos básicos', que consistem em sequências de instruções consecutivas nas quais não há nenhum tipo de desvio.

  * O particionamento da representação intermediária em blocos básicos também é foco do nosso estudo. Veremos as transformações locais simples que podem ser usadas para transformar blocos básicos em blocos básicos modificados, a partir dos quais um código mais eficiente pode ser gerado.

Geração de Código

* **Introdução**
  * Essas transformações são uma forma rudimentar de otimização de código, embora a teoria mais avançada de otimização de código seja discutida na AULA08.

  * Um exemplo de transformação local útil é a descoberta de subexpressões comuns no nível de código intermediário e a substituição resultante das operações aritméticas por operações de cópia mais simples.

Geração de Código

* **Introdução**
  * Estudaremos aqui um algoritmo de geração de código simples, que gera código para cada um dos comandos, um de cada vez, mantendo os operandos em registradores sempre que possível.

  * A saída desse tipo de gerador pode ser facilmente melhorada utilizando técnicas de otimização de código.

Geração de Código

* **Introdução**
  * Embora os detalhes dependam de questões específicas da representação intermediária, da linguagem objeto e do sistema em tempo de execução, tarefas como seleção de instrução, alocação e atribuição de registradores, e escalonamento de instruções são encontradas no projeto de quase todos os geradores de código.

Geração de Código

* **Introdução**
  * O critério mais importante para um gerador de código é que ele produza **código correto**.
  * A exatidão assume significado especial devido ao número de casos especiais que um gerador de código poderia enfrentar.
  * Dada a prioridade na correção, o projeto de um gerador de código que possa ser facilmente implementado, testado e gerenciado é um objetivo de projeto importante.

Geração de Código

* **Introdução**
  * A **entrada** para um gerador de código é uma representação intermediária do programa fonte, produzida pelo *front-end*, com as informações da tabela de símbolos que são usadas para determinar os endereços em tempo de execução dos objetos de dados denotados pelos nomes na representação intermediária.

Geração de Código

* **Introdução**
  * As diversas escolhas para a representação intermediária incluem as representações de três endereços, as representações de máquina virtual como bytecodes e código de máquina de pilha; as representações lineares como notação pós-fixada; e as representações gráficas como árvores de sintaxe e grafos acíclicos dirigidos.

  * Muitos dos algoritmos que veremos são enunciados em termos das representações consideradas nas aulas anteriores: código de três endereços, árvores e grafos acíclicos dirigidos. Contudo, as técnicas que discutimos também podem ser aplicadas com outras representações intermediárias.

Geração de Código

* **Introdução**
  * Iremos considerar que o *front-end* escandiu, analisou e traduziu o programa fonte para uma representação intermediária relativamente de baixo nível, de modo que os valores dos nomes que aparecem na representação intermediária podem ser representados por quantidades que a máquina alvo pode manipular diretamente, como número inteiros e de ponto flutuante.

Geração de Código

* **Introdução**
  * Também consideraremos que todos os erros sintáticos e semânticos estáticos foram detectados, que foi feita a verificação de tipos necessária, e que os operadores de conversão de tipo foram inseridos onde necessário.

  * O gerador de código, portanto, pode prosseguir, supondo que sua entrada está livre desses tipos de erros.

Geração de Código

* **Introdução**
  * Vejamos agora questões sobre o **programa objeto**.
  * A arquitetura do conjunto de instruções da máquina alvo tem um impacto significativo sobre a dificuldade de construir um bom gerador de código que produza código de máquina de alta qualidade.
  * As arquiteturas de máquina alvo mais comuns são a RISC (reduced instruction set computer), a CISC (complex instruction set computer) e as baseadas em pilha.

Geração de Código

* **Introdução**
  * Uma máquina RISC (PowerPC por exemplo) tipicamente possui muitos registradores, instruções de três endereços, modos de endereçamento simples e uma arquitetura do conjunto de instruções relativamente simples.

  * Ao contrário, uma máquina CISC (Intel ou AMD por exemplo) normalmente possui menos registradores, instruções de dois endereços, muitos modos de endereçamento, várias classes de registradores, instruções de tamanho variável e instruções com efeitos colaterais.

Geração de Código

* **Introdução**
  * Em uma máquina baseada em pilha, as operações são feitas colocando-se os operandos em uma pilha e depois efetuando-se as operações com os operandos no topo da pilha.
  * Para obter alto desempenho, o topo da pilha tipicamente é mantido em registradores.

Geração de Código

* **Introdução**
  * As máquinas baseadas em pilha quase desaparecem, porque se achava que este tipo de organização era bastante limitador e exigia muitas operações de troca e cópia.

  * Contudo, as arquiteturas baseadas em pilha reviveram com a introdução da Máquina Virtual Java (Java Virtual Machine – JVM). A JVM é um software que interpreta bytecodes em Java, uma linguagem **intermediária** produzida por compiladores Java.

Geração de Código

* **Introdução**
  * O interpretador oferece compatibilidade de software com múltiplas plataformas, um fator importante para o sucesso do Java.

  * Para contornar o baixo desempenho de interpretação, foram projetados os compiladores Java just-in-time (JIT). Esses compiladores JIT traduzem os bytecodes em tempo de execução para o conjunto de instruções de hardware nativo da máquina alvo.

Geração de Código

* **Introdução**
  * Outra técnica para melhorar o desempenho do Java é desenvolver um compilador que compile diretamente para as instruções de máquina da arquitetura alvo, evitando totalmente gerar os bytecodes Java.

  * A produção de um programa em linguagem de máquina absoluta como saída tem a vantagem de poder ser colocado em um local fixo na memória e executado imediatamente. Os programas podem ser compilados e executados rapidamente.

Geração de Código

* **Introdução**
  * A produção de um programa em linguagem de máquina relocável (frequentemente chamado de módulo objeto) como saída permite que os subprogramas sejam compilados separadamente.

  * Um conjunto de módulos objeto relocáveis pode ser ligado e carregado para execução por um editor de ligação e um carregador.

Geração de Código

* **Introdução**
  * Embora tenhamos de pagar pelo custo adicional da ligação e da carga se produzirmos módulos objeto relocáveis, ganhamos muito em flexibilidade, sendo possível compilar sub-rotinas separadamente e chamar outros programas previamente compilados a partir de um módulo objeto.

  * Se a máquina alvo não tratar de relocação automaticamente, o compilador deve prover informações de relocação explícitas ao carregador para ligar os módulos de programa compilados separadamente.

Geração de Código

* **Introdução**
  * A produção de um programa em linguagem assembly como saída torna o processo de geração de código bem mais fácil.
  * Podemos gerar instruções simbólicas e utilizas as facilidades de macro do Montador para auxiliar na geração de código.
  * O preço pago é a etapa de assembly **após** a geração de código.

Geração de Código

* **Introdução**
  * Iremos considerar um computador tipo RISC muito simples como nossa máquina alvo.
  * Adicionamos a ele alguns modos de endereçamento tipo CISC, de modo que possamos também discutir as técnicas de geração de código para máquinas CISC.

Geração de Código

* **Introdução**
  * Por questão de legibilidade, usamos o código assembly como linguagem objeto.
  * Desde que os endereços possam ser calculados a partir dos deslocamentos e outras informações armazenadas na tabela de símbolos, o gerador de código pode produzir endereços relocáveis e absolutos para os nomes tão facilmente quanto os endereços simbólicos.

Geração de Código

* **Seleção das instruções de máquina**
  * O gerador de código precisa mapear o programa na representação intermediária em uma sequência de código que possa ser executada pela arquitetura alvo.

  * A complexidade da realização desse mapeamento é determinada por fatores como:
    * O nível da representação intermediária;
    * A natureza da arquitetura do conjunto de instruções;
    * A qualidade desejada do código gerado.

Geração de Código

* **Seleção das instruções de máquina**
  * Se a representação intermediária for de alto nível, o gerador de código pode traduzir cada comando da representação intermediária em uma sequência de instruções de máquina usando gabaritos de código.
  * Todavia, essa geração de código comando por comando frequentemente produz um código ruim, que exige otimização adicional.

Geração de Código

* **Seleção das instruções de máquina**
  * Se a representação intermediária refletir alguns dos detalhes de baixo nível da máquina alvo subjacente, o gerador de código pode usar essa informação para gerar sequências de código mais eficientes.

  * A natureza do conjunto de instruções da máquina alvo tem forte efeito sobre a dificuldade da seleção de instruções. Por exemplo, a uniformidade e a completeza do conjunto de instruções são fatores importantes. Se a máquina alvo não prover cada tipo de dado de maneira flutuante serão efetuadas usando-se registradores separados.

Geração de Código

* **Seleção das instruções de máquina**
  * As velocidades das instruções e os idiomas da máquina são outros fatores críticos. Se não nos importarmos com a eficiência do programa objeto, a seleção de instruções é direta.

  * Para cada tipo de comando de três endereços, podemos projetar um esqueleto de código que define o código objeto a ser gerado para essa construção.

Geração de Código

* **Seleção das instruções de máquina**
  * Por exemplo, todo comando de três endereços da forma `x = y + z`, onde `x`, `y` e `z` são alocados estaticamente, pode ser traduzido para a sequência de código:

```
LD   R0, y      // R0 = y       (carrega y no registrador R0)
ADD  R0, R0, z  // R0 = R0 + z  (soma z a R0)
ST   x, R0      // x = R0       (armazena R0 em x)
```

Geração de Código

* **Seleção das instruções de máquina**
  * Essa estratégia frequentemente produz cargas e armazenamentos redundantes. Por exemplo, a sequência de comandos de três endereços:

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

  * seria traduzida para:

```
LD   R0, b        // R0 = b
ADD  R0, R0, c    // R0 = R0 + c
ST   a, R0        // a = R0
LD   R0, a        // R0 = a
ADD  R0, R0, e    // R0 = R0 + e
ST   d, R0        // d = R0
```

Geração de Código

* **Seleção das instruções de máquina**
  * Neste exemplo, o quarto comando é redundante, porque carrega um valor que acabou de ser armazenado, e também a terceira instrução, se a não for usado subsequentemente.

  * A qualidade do código gerado usualmente é determinada em função de sua velocidade e tamanho. Na maioria das máquinas, um programa em dada representação intermediária pode ser implementado por muitas sequências de código diferentes, com diferenças significativas de custo entre as diferentes implementações.

Geração de Código

* **Seleção das instruções de máquina**
  * Uma tradução ingênua do código intermediário pode, portanto, produzir um código objeto correto, porém inaceitavelmente ineficiente.

  * Por exemplo, se a máquina alvo tiver uma instrução de "incremento" (INC), o comando de três endereços, `a = a + 1` pode ser implementado de modo mais eficiente por uma única instrução `INC a`, em vez de uma sequência mais óbvia que carrega `a` em um registrador, soma um ao registrador e depois armazena o resultado de volta em `a`:

```
LD   R0 , a       // R0 = a
ADD R0, R0, #1    // R0 = R0 + 1
ST   a, R0        // a = R0
```

Geração de Código

* **Seleção das instruções de máquina**
  * É preciso conhecer os custos da instrução a fim de projetar boas sequências de código, mas 'infelizmente' informações precisas a respeito do custo de uma instrução costumam ser difíceis de obter.

  * Decidir qual sequência do código de máquina é melhor para determinada construção de três endereços também pode exigir conhecimento a respeito do contexto em que essa construção aparece.

Geração de Código

* **Seleção das instruções de máquina**
  * A seleção de instrução também pode ser modelada como um processo de casamento de árvores de padrões, em que representamos a representação intermediária e as instruções de máquina como árvores.

  * Depois, tentamos 'substituir' uma árvore de representação intermediária com um conjunto de subárvores que correspondam às instruções de máquina. Se associarmos um custo a cada subárvore de instrução de máquina, poderemos usar técnicas da programação dinâmica para gerar sequências ótimas de código.

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * Um problema importante na geração de código é decidir que valores residirão em registradores e em quais registradores.

  * Os registradores são considerados a unidade computacional mais rápida da máquina alvo, mas usualmente as arquiteturas não possuem um número suficientes deles para abrigar todos os valores de um programa.

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * Os valores não mantidos em registradores precisam residir na memória.
  * As instruções envolvendo operandos em registrador são invariavelmente menores e mais rápidas do que aquelas envolvendo operandos na memória; assim, a utilização eficiente dos registradores é particularmente importante.

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * O uso de registradores frequentemente é subdividido em dois subproblemas:

    * 1) Alocação de registradores, etapa na qual selecionamos o conjunto de variáveis que residirão nos registradores em cada ponto do programa.

    * 2) Atribuição de registradores, etapa na qual determinamos um registrador específico em que uma variável residirá.

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * É difícil encontrar uma atribuição ótima de registradores para as variáveis, até mesmo com máquinas de um único registrador.
  * O problema é ainda mais complicado porque o hardware e/ou o sistema operacional da máquina alvo podem exigir que sejam observadas certas convenções no uso de registrador.

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * Ter familiaridade com a máquina alvo e seu conjunto de instruções é um pré-requisito no projeto de um bom gerador de código.
  * Infelizmente, em uma discussão geral sobre a geração de código, não é possível descrever nenhuma máquina alvo com detalhes suficientes para gerar um bom código para uma linguagem completa nessa máquina.

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * Usaremos como linguagem objeto o código **assembly** para um computador simples, que representa máquinas com muitos registradores.
  * Contudo, as técnicas de geração de código aqui apresentadas podem ser usadas em muitas outras classes de máquinas.

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * Nosso computador alvo modela uma máquina de três endereços com operações de carga e armazenamento, operações de cálculo, operações de desvios incondicionais e desvios condicionais.
  * O computador subjacente é uma máquina endereçável por byte, com n registradores de propósito geral.

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * Uma linguagem assembly completa teria muitas instruções. Para evitar esconder os conceitos em uma infinidade de detalhes, usaremos um conjunto de instruções muito limitado e presumiremos que todos os operandos são inteiros.

  * A maior parte das instruções consiste em um operador, seguido por um destino, seguido por uma lista de operandos de origem. Um rótulo poderá preceder uma instrução.

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * Exemplo: A instrução de três endereços `x = y - z` pode ser implementada pelas instruções de máquina:

```
LD   R1, y          // R1 = y
LD   R2, z          // R2 = z
SUB  R1, R1, R2     // R1 = R1 - R2
ST   x, R1          // x = R1
```

  * Podemos fazer melhor, vejamos a seguir.

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * Um dos objetivos de um bom algoritmo de geração de código é, sempre que possível, evitar o uso de todas essas quatro instruções.
  * Por exemplo, y e/ou z podem ter sido computados em um registrador e, nesse caso, podemos evitar os passos LD.
  * Do mesmo modo, é possível evitar o armazenamento de x se seu valor for usado a partir do conjunto de registradores e não for subsequentemente necessário.

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * Suponha que `a` seja um arranjo cujos elementos sejam valores de 8 bytes, talvez números reais. Suponha também que os elementos de `a` são indexados a partir de 0.

  * Podemos executar a instrução de três endereços `b = a[i]` usando as seguintes instruções de máquina:

```
LD   R1, i        // R1 = i
MUL  R1, R1, 8    // R1 = R1 * 8
LD   R2, a(R1)    // R2 = conteúdo(a + conteúdo(R1))
ST   b, R2        // b = R2
```

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * Ou seja, o segundo passo calcula $8i$, e o terceiro passo coloca no registrador R2 o valor no i-ésimo elemento de `a` – aquele encontrado no endereço que está $8i$ bytes após o endereço base do arranjo `a`.

  * Similarmente, a atribuição no arranjo `a` representada pela instrução de três endereços `a[j] = c` é implementada por:

```
LD   R1, c          // R1 = c
LD   R2, j          // R2 = j
MUL  R2, R2, 8      // R2 = R2 * 8
ST   a(R2), R1      // conteúdo(a + conteúdo(R2)) = R1
```

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * Finalmente, considere uma instrução de desvio condicional de três endereços como:

```
if x < y goto L
```

  * O equivalente código de máquina seria algo como:

```
LD    R1, x          // R1 = x
LD    R2, y          // R2 = y
SUB   R1, R1, R2     // R1 = R1 - R2
BLTZ  R1, M          // if R1 < 0 jump to M
```

  * Neste exemplo, `M` é o rótulo que representa a primeira instrução de máquina gerada a partir da instrução de três endereços que tem o rótulo `L`. Assim como para qualquer instrução de três endereços, esperamos poder economizar algumas dessas instruções de máquina porque os operandos necessários já estão nos registradores ou porque o resultado nunca precisa ser armazenado.

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * Frequentemente, nós associamos um custo à compilação e execução de um programa.
  * Dependendo de qual aspecto do programa estamos interessados em otimizar, algumas medidas de custo comuns são o tempo de compilação e o tamanho, o tempo de execução e o consumo de energia do programa objeto.

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * Determinar o custo real da compilação e execução de um programa é um problema complexo.
  * Encontrar um programa objeto ótimo para determinado programa fonte é, em geral, um problema indecidível.
  * Na geração de código normalmente temos de nos contentar com as técnicas heurísticas que produzem códigos objetos bons, mas não necessariamente ótimos.

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * Iremos considerar que cada instrução da linguagem objeto possui um custo associado. Para simplificar, consideramos o custo de uma instrução como sendo **um** mais associados aos modos de endereçamento dos operandos.

  * Esse custo corresponde ao tamanho da instrução em palavras. Os modos de endereçamento envolvendo registradores possuem zero custo adicional, enquanto aqueles envolvendo um endereço ou constante da memória possuem um custo adicional de um, porque tais operandos têm de ser armazenados nas palavras seguindo a instrução.

Geração de Código

- **Definição de registros e de blocos sequenciais de código**
  - Alguns exemplos:
    - A instrução LD R0, R1 copia o conteúdo do registrador R1 para o registrador R0. Essa instrução tem o custo de um, porque nenhuma palavra adicional de memória é necessária.
    - A instrução LD R0, M carrega o conteúdo do endereço de memória M no registrador R0. O custo é dois, porque o endereço de memória de M está na palavra seguinte à instrução.
    - A instrução LD R1, *100(R2) carrega no registrador R1 o valor dado por: *conteúdo*(*conteúdo*(100 + *conteúdo*(R2))). O custo é três porque a constante 100 é armazenada na palavra seguinte à instrução.

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * Consideraremos que o custo de um programa na linguagem objeto para dada entrada é a soma dos custos das instruções individuais quando o programa é executado nessa entrada.
  * Bons algoritmos de geração de código buscam minimizar a soma dos custos das instruções executadas pelo programa objeto gerado para entradas típicas.
  * Veremos que, em algumas situações, podemos realmente gerar um código ótimo para expressões em certas classes de máquinas baseadas em registrador.

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * Podemos realizar um trabalho melhor de alocação de registradores se soubermos como os valores são definidos e usados.

  * Podemos realizar um trabalho melhor de seleção de instrução examinando as sequências de comandos de três endereços.

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * Introduziremos uma representação do código intermediário na forma de grafo que é útil para discutir a geração de código mesmo que o grafo não seja construído explicitamente por um algoritmo de geração de código.
  * A geração de código se beneficia do contexto.

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * Esta representação é construída da seguinte forma:
    * 1) Posicione o código intermediário em blocos básicos, os quais são sequências máximas de instruções consecutivas de três endereços com as seguintes propriedades:
      * (a) O fluxo de controle só pode entrar no bloco básico por meio da primeira instrução do bloco, ou seja, não existem desvios para o meio do bloco.
      * (b) O controle sairá do bloco sem interrupção ou desvio, exceto possivelmente na última instrução do bloco.
    * 2) Os blocos básicos formam os nós de um grafo de fluxo, cujas arestas denotam quais blocos podem seguir quaisquer outros blocos.

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * A partir da AULA08, discutiremos as mudanças efetuadas nos grafos de fluxo que transformam o código intermediário original em um código intermediário 'otimizado', a partir do qual pode ser gerado um código objeto melhor.
  * O código intermediário 'otimizado' é transformado em código de máquina usando as técnicas de geração de código contidas aqui.

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * Nossa próxima tarefa é particionar uma sequência de instruções de três endereços em blocos básicos.
  * Iniciamos um novo bloco básico com a primeira instrução do programa fonte e continuamos acrescentando instruções até encontrar um desvio incondicional, um desvio condicional ou um rótulo na instrução seguinte.
  * Na ausência de desvios e rótulos, o controle prossegue sequencialmente de uma instrução para a seguinte. A ideia é formalizada no algoritmo a seguir:

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * Algoritmo para particionamento das instruções de três endereços em blocos básicos:
  * Algoritmo 1:
  * **ENTRADA**: A sequência de instruções de três endereços.
  * **SAÍDA**: Uma lista de blocos básicos para essa sequência na qual cada instrução é atribuída a exatamente um bloco básico.
  * **MÉTODO**: Primeiro, determinamos as instruções no código intermediário que são *líderes*, ou seja, as primeiras instruções em algum bloco básico. A instrução logo após o fim do programa intermediário não é incluída como um líder. As regras para encontrar os líderes são:

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * 1) A primeira instrução de três endereços no código intermediário é um líder.
  * 2) Qualquer instrução que seja o destino de um desvio condicional ou incondicional é um líder.
  * 3) Qualquer instrução que siga imediatamente um desvio condicional ou incondicional é um líder.

  * Então, para cada líder, seu bloco básico consiste em si mesmo e em todas as instruções até o próximo líder, sem incluí-lo, ou até o fim do programa intermediário.

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * Exemplo: O código intermediário da Fig. 2 transforma uma matriz a de 10 x 10 em uma matriz identidade. Embora não seja importante de onde vem esse código, ele poderia ser a tradução do pseudocódigo da Fig. 3.

  * Ao gerar o código intermediário, assumimos que os elementos de arranjo com valor real ocupam 8 bytes cada, e que a matriz a é armazenada por linha.

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * Primeiro, a instrução 1 é um líder pela regra (1) do algoritmo 1 visto anteriormente. Para encontrar os outros líderes, precisamos primeiro encontrar os desvios. Neste exemplo, existem três desvios, todos condicionais, nas instruções 9, 11 e 17.
  * Pela regra (2), os destinos desses desvios são líderes; eles são representados pelas instruções 3, 2 e 13, respectivamente.
  * Depois, pela regra (3), cada instrução após um desvio é um líder; essas são as instruções 10 e 12. Observe que nenhuma instrução vem após 17 nesse código, mas se houvesse mais instruções a instrução 18 também seria um líder.

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * Concluímos que os líderes são as instruções 1, 2, 3, 10, 12 e 13. O bloco básico de cada líder contém todas as instruções a partir de si mesmo até a instrução imediatamente antes do próximo líder. Assim, o bloco básico 1 é apenas 1, para o líder 2 o bloco é apenas 2.

  * O líder 3, contudo, tem um bloco básico consistindo nas instruções de 3 até 9, inclusive. O bloco da instrução 10 é 10 e 11; o bloco da instrução 12 é apenas 12, e o bloco da instrução 13 vai de 13 a 17.

Geração de Código

* Abaixo, Fig. 2 e 3 respectivamente:

```
 1)    i = 1
 2)    j = 1
 3)    t1 = 10 * i
 4)    t2 = t1 + j
 5)    t3 = 8 * t2
 6)    t4 = t3 - 88
 7)    a[t4] = 0.0
 8)    j = j + 1
 9)    if j <= 10 goto (3)
10)    i = i + 1
11)    if i <= 10 goto (2)
12)    i = 1
13)    t5 = i - 1
14)    t6 = 88 * t5
15)    a[t6] = 1.0
16)    i = i + 1
17)    if i <= 10 goto (13)
```

Código intermediário para definir uma matriz de 10 × 10 como uma matriz de identidade

```
for i from 1 to 10 do
        for j from 1 to 10 do
                a[i,j] = 0.0;
for i from 1 to 10 do
        a[i,i] = 1.0;
```

Código fonte para a Figura 2

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * Saber se o valor de uma variável será usado em seguida é essencial para gerar um bom código.
  * Se o valor de uma variável correntemente em um registrador nunca for referenciado posteriormente, então esse registrador pode ser atribuído a outra variável.

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * O uso de um nome em um comando de três endereços é definido da seguinte forma: suponha que o comando de três endereços i atribua um valor a x;
  * Se o comando j tiver x como operando, e o controle puder fluir a partir do comando i para j ao longo de um caminho que não possui atribuições intervenientes para x, então dizemos que o comando j usa o valor de x calculado no comando i. Dizemos ainda que x está vivo no comando i.

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * Queremos determinar para cada comando de três endereços $x = y + z$ quais são os próximos usos de x, y e z. Para o presente, não nos preocupamos com os usos fora do bloco básico contendo esse comando de três endereços.

  * Nosso algoritmo para determinar se uma variável está viva e o seu próximo uso faz uma passada de trás para a frente em cada bloco básico. Armazenamos a informação na tabela de símbolos.

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * Podemos facilmente pesquisar em um fluxo de comandos de três endereços para encontrar as extremidades dos blocos básicos como no algoritmo 1 visto anteriormente.

  * Como os procedimentos podem ter quaisquer efeitos colaterais, consideramos por conveniência que cada chamada de procedimento inicia um novo bloco básico.

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * Algoritmo para determinar informação de tempo de vida e próximo uso de variáveis para cada comando em um bloco básico:
  * Algoritmo 2:
  * **ENTRADA:** Um bloco básico B com comandos de três endereços. Consideramos que a tabela de símbolos inicialmente mostra todas as variáveis não temporárias em B como estando vivas na saída.
  * **SAÍDA:** Em cada comando i: $x = y + z$ em B, associamos a i a informação de tempo de vida e próximo uso de x, y e z.
  * **MÉTODO:** Começamos no último comando em B e percorremos a partir do fim até o início de B.

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * Para cada comando i: $x = y + z$ em B, fazemos o seguinte:

  * 1) Associamos ao comando i a informação correntemente encontrada na tabela de símbolos em relação ao próximo uso e tempo de vida de x, y e z.
  * 2) Na tabela de símbolos, definimos x como 'não vivo' e 'nenhum próximo uso'.
  * 3) Na tabela de símbolos, definimos y e z como 'vivo' e os próximos usos de y e z para i.

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * Aqui, usamos + como um símbolo representando qualquer operador. Se o comando de três endereços i tiver a forma x = + y ou x = y, os passos são os mesmos, ignorando z. Observe que a ordem dos passos (2) e (3) não pode ser trocada porque x pode ser y ou z.

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * Uma vez que um programa em código intermediário é particionado em blocos básicos, representamos o fluxo de controle entre eles por meio de um grafo de fluxo.

  * Os nós do grafo de fluxo são os blocos básicos. Existe uma aresta do bloco *B* para o bloco *C* se e somente se for possível que o primeiro comando no bloco *C* venha imediatamente após o último comando no bloco *B*.

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * Existem duas maneiras pelas quais esse tipo de aresta poderia ser justificada:
    * Existe um desvio condicional ou incondicional a partir do fim de *B* para o início de *C*.
    * *C* vem imediatamente após *B* na ordem original dos comandos de três endereços, e *B* não termina com um desvio incondicional.
  * Dizemos que B é um predecessor de *C*, e *C* é um sucessor de *B*.

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * Frequentemente, incluímos no grafo de fluxo dois nós, chamados de entrada e saída, que não fazem parte das instruções intermediárias executáveis.
  * Existe uma aresta da entrada para o primeiro nó executável do grafo de fluxo, ou seja, para o bloco básico que contém a primeira instrução do código intermediário.
  * Há uma aresta para a saída a partir de qualquer bloco básico que contenha uma instrução que poderia ser a última instrução executada no programa.

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * Se a instrução final do programa não for um desvio incondicional, então o bloco contendo a instrução final do programa será um predecessor da saída, mas o mesmo acontecerá com qualquer bloco básico que tiver um desvio para o código que não faça parte do programa.

  * Vejamos um exemplo.

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * O conjunto de blocos básicos construídos na Fig. 2 gera o grafo de fluxo da Fig. 4.

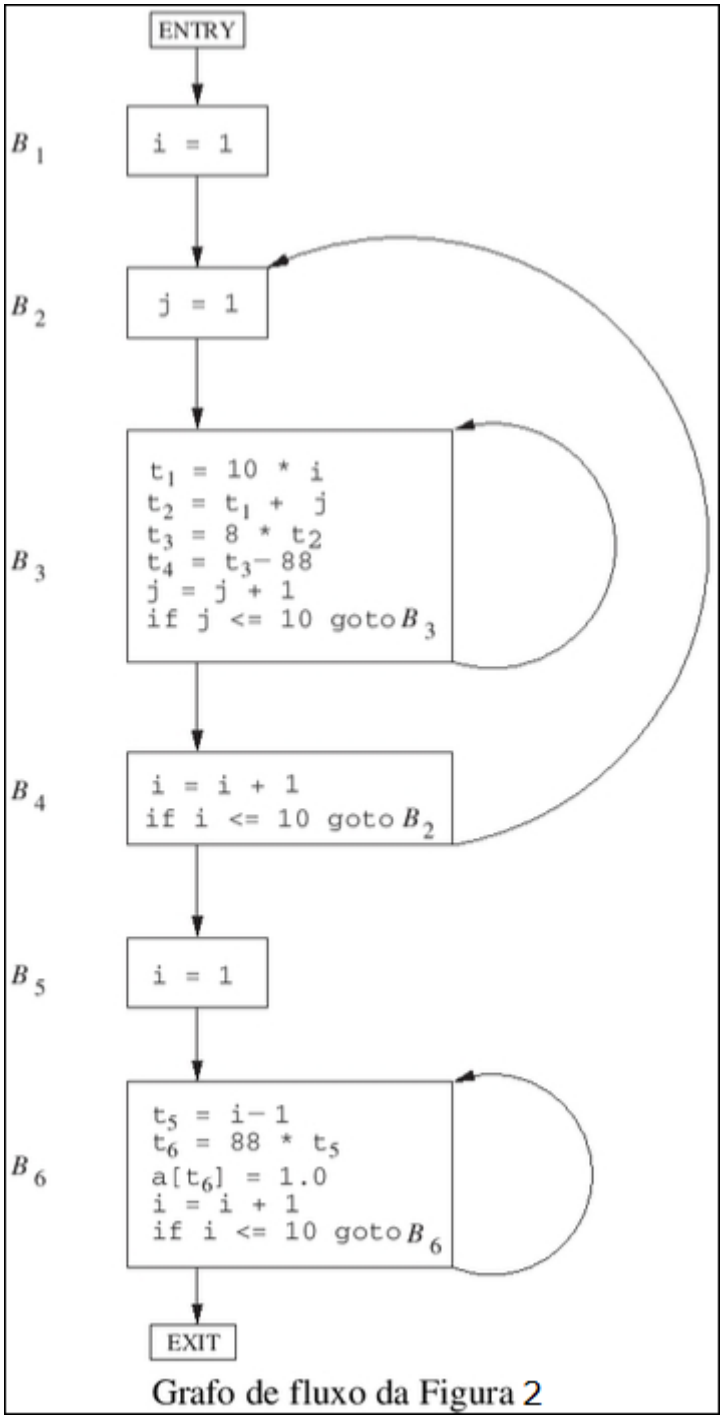
  * A entrada aponta para o bloco básico *B1*, porque *B1* contém a primeira instrução do programa. O único sucessor de *B1* é *B2*, porque *B1* não termina em um desvio incondicional, e o líder de *B2* vem imediatamente após o fim de *B1*.

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * O bloco *B3* tem dois sucessores. Um é ele mesmo, porque o líder de *B3*, a instrução 3, é destino do desvio condicional no fim de *B3*, representando a instrução 9. O outro sucessor é *B4*, porque o controle pode seguir pelo desvio condicional no fim de *B3* e em seguida entrar no líder de *B4*.

  * Somente *B6* aponta para a saída do grafo de fluxo, pois a única maneira de chegar ao código que vem após o programa para o qual construímos o grafo de fluxo é via o desvio condicional que encerra *B6*.

* Ao lado, Fig. 4:



Grafo de fluxo da Figura 2

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * Observe na Fig. 4 que, no grafo de fluxo, é normal substituir os desvios por números de instrução ou rótulos por desvios para blocos básicos.
  * Lembre-se de que todo desvio condicional ou incondicional é para o líder de algum bloco básico, e é para esse bloco que o desvio agora se referirá.

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * O motivo para essa mudança é que, depois de construir o grafo de fluxo, é comum efetuar alterações substanciais nas instruções dos diversos blocos básicos.
  * Se os desvios fossem para as instruções, teríamos de modificar os destinos dos desvios toda vez que uma das instruções de destino fosse alterada.

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * Os grafos de fluxo, sendo grafos muito comuns, podem ser representados por qualquer uma das estruturas de dados apropriadas para grafos.

  * O conteúdo dos nós (blocos básicos) precisa de sua própria representação. Poderíamos representar o conteúdo de um nó por um apontador para o líder no arranjo de instruções de três endereços, com um contador do número de instruções ou um segundo apontador para a última instrução. Contudo, como o número de instruções em um bloco básico pode mudar com frequência, provavelmente será mais eficiente criar uma lista encadeada de instruções para cada bloco básico.

Geração de Código

* **Definição de registros e de blocos sequenciais de código**
  * Frequentemente, podemos obter uma melhoria substancial no tempo de execução do código simplesmente realizando a otimização **local** dentro de cada bloco básico.

  * A otimização **global** mais completa, que examina como a informação flui entre os blocos básicos de um programa, é explicada na AULA08. Esse é um assunto complexo, com muitas técnicas diferentes a serem consideradas.

Geração de Código

* **Otimizações de fluxo de controle**
  * Enquanto a maioria dos compiladores de produção gera código bom por meio de uma cuidadosa seleção de instrução e alocação de registradores, alguns usam uma estratégia alternativa: **geram um código simples e depois melhoram a qualidade do código objeto aplicando transformações de 'otimização' ao programa objeto**.

  * O termo 'otimização' é um tanto enganoso, pois não existe garantia de que o código resultante é ótimo sob qualquer medida matemática. Apesar disso, muitas transformações simples podem melhorar significativamente o tempo de execução ou o espaço demandado pelo programa objeto.

Geração de Código

* **Otimizações de fluxo de controle**
  * Uma técnica simples, porém eficaz, para melhorar o código objeto localmente é a otimização de janela (*peephole*), que é feita examinando-se umas poucas instruções objeto visíveis em uma janela deslizante (chamada janela) e substituindo-as por uma sequência menor ou mais rápida, sempre que possível.

  * A otimização de janela também pode ser aplicada diretamente após a geração de código intermediário para melhorar a representação intermediária.

Geração de Código

* **Otimizações de fluxo de controle**
  * A ‘janela’ é uma pequena janela deslizante em um programa. O código na janela não precisa ser contíguo, embora algumas implementações o exijam.

  * É característica da otimização de janela que cada melhoria possa gerar oportunidades para melhorias adicionais. Em geral, para se obter o máximo de benefício são necessários passadas repetidas pelo código objeto.

Geração de Código

* **Otimizações de fluxo de controle**
  * São exemplos de transformações de programa que são características das otimizações de janela:
    * Eliminação de instrução redundante
    * **Otimizações de fluxo de controle**
    * Simplificação algébricas
    * Uso de idiomas de máquina

  * Neste curso, daremos ênfase as 'Otimizações de fluxo de controle'.

Geração de Código

* **Otimizações de fluxo de controle**
  * Os algoritmos simples de geração de código intermediário frequentemente produzem desvios para desvios, desvios para desvios condicionais ou desvios condicionais para desvios.

  * Esses desvios desnecessários podem ser eliminados no código intermediário ou no código objeto pelos tipos de otimizações de janela a seguir.

Geração de Código

* **Otimizações de fluxo de controle**
  * Podemos substituir a sequência:

```
      goto L1
      ...
L1:   goto L2
```

  * Pela sequência:

```
      goto L2
      ...
L1:   goto L2
```

Geração de Código

* **Otimizações de fluxo de controle**
  * Se agora não houver desvios para `L1`, talvez seja possível eliminar o comando `L1: goto L2`, desde que ele seja precedido por um desvio incondicional.

  * Ou seja, otimizar o produto resultante da otimização.

Geração de Código

* **Otimizações de fluxo de controle**
  * De modo semelhante, a sequência:

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

  * Pode ser substituída pela sequência:

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

Geração de Código

* **Otimizações de fluxo de controle**
  * Finalmente, suponha que haja apenas um desvio para `L1` e `L1` seja precedido por um desvio incondicional. Então, a sequência:

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

  * Pode ser substituída pela sequência:

```
        if a < b goto L2
        goto L3
        ...
L3
```

Geração de Código

* **Otimizações de fluxo de controle**
  * Embora o número de instruções nas duas sequências seja o mesmo, às vezes saltamos o desvio incondicional na segunda sequência, mas nunca na primeira.

  * Assim, a segunda sequência é superior à primeira em tempo de execução.

Resumo

* ***Geração de código***: É a fase final de um compilador. O gerador de código mapeia a representação intermediária produzida pelo *front-end*, ou, se houver uma fase de otimização realizada pelo otimizador de código, para o programa objeto.

* ***Seleção de instrução***: É o processo de escolher instruções da linguagem objeto para cada comando na representação intermediária.

* ***Alocação de registrador***: É o processo de decidir que valores da representação intermediária devem ser mantidos em registradores. A coloração de grafos é uma técnica efetiva para realizar alocação de registrador em compiladores.

* ***Atribuição de registrador***: É o processo de decidir que registrador deve manter um determinado valor da representação intermediária.

Resumo

* ***Máquina Virtual***: Uma máquina virtual é um interpretador para uma linguagem intermediária de bytecode, produzida para linguagens como Java e C#.

* ***Uma máquina CISC***: Tipicamente é uma máquina de dois endereços com relativamente poucos registradores, várias classes de registrador, e instruções de tamanho variável com modos de endereçamento complexos.

* ***Uma máquina RISC***: Tipicamente é uma máquina de três endereços com muitos registradores, na qual as operações são feitas nos registradores.

* ***Bloco básico***: Um bloco básico é uma sequência máxima de comandos consecutivos de três endereços, em que o fluxo de controle só pode entrar no primeiro comando do bloco e sair no último comando sem interromper ou desviar, exceto possivelmente no último comando do bloco básico.

Resumo

* ***Grafo de fluxo***: Um grafo de fluxo é uma representação gráfica de um programa, em que os nós do grafo são blocos básicos e as arestas do grafo mostram como o controle flui entre os blocos.

* ***Otimizações de janela***: São transformações locais de melhoria de código que podem ser aplicadas a um programa, usualmente por meio de uma janela deslizante.

* ***Seleção de instrução***: Pode ser feita por um processo de reescrita da árvore, em que os padrões de árvore correspondentes a instruções de máquina são usadas para transformar uma árvore de sintaxe. Podemos associar custos às regras de reescrita de árvore e aplicar a programação dinâmica para obter uma transformação ótima para classes úteis de máquinas e expressão.

Obrigado.

joaopauloaramuni@gmail.com
joaopauloaramuni@fumec.br

UNIVERSIDADE FUMEC
50 Anos