Compiladores

CIÊNCIA DA COMPUTAÇÃO

Prof. Dr. João Paulo Aramuni

UNIVERSIDADE FUMEC

Sumário

* **Geração de Código Intermediário**
  * Introdução
  * Verificação estática
    * Verificação sintática
    * Verificação de tipo
  * Código de três endereços
  * Comandos de fluxo de controle

Geração de Código Intermediário

* **Introdução**
  * No modelo de análise e síntese de um compilador, o *front-end* analisa um programa fonte e cria uma representação intermediária, a partir da qual o *back-end* gera o código objeto.

Geração de Código Intermediário

* **Introdução**
  * Com uma representação intermediária definida adequadamente, um compilador para a linguagem i e a máquina j pode então ser construído, combinando-se o *front-end* para a linguagem i com o *back-end* para a máquina j.

  * Essa abordagem para a criação de um conjunto de compiladores pode economizar muito esforço: m x n compiladores podem ser construídos escrevendo-se apenas m front-ends e n back-ends.

Geração de Código Intermediário

* **Introdução**
  * Conforme vimos na AULA02, especialmente na Fig. 10, as duas representações intermediárias mais importantes são:
    * Árvores, incluindo árvores de derivação e árvores sintáticas (**abstratas**).
    * Representações lineares, especialmente o "código de três endereços".

Geração de Código Intermediário

* **Introdução**
  * Em árvores sintáticas, a sintaxe **abstrata** nos permite agrupar operadores "semelhantes" para reduzir o número de casos e subclasses dos nós em uma implementação das expressões.
  * Veremos que "semelhante" significa que as regras de verificação de tipo e geração de código para os operadores são semelhantes.

Geração de Código Intermediário

* **Introdução**
  * Por exemplo, normalmente os operadores + e * podem ser agrupados, uma vez que eles podem ser tratados da mesma maneira – seus requisitos em relação aos tipos dos operandos são iguais, e cada um deles resulta em uma única instrução de três endereços, que aplica um operador a dois valores.

  * Em geral, o agrupamento de operadores na sintaxe abstrata é baseado nas necessidades das fases posteriores do compilador.

Geração de Código Intermediário

* **Introdução**
  * A tabela da Fig. 1, abaixo, especifica a correspondência entre a sintaxe **concreta** e **abstrata** para vários dos operadores Java:

SINTAXE CONCRETA	SINTAXE ABSTRATA
=	**assign**
||	**cond**
&&	**cond**
== !=	**rel**
< <= >= >	**rel**
+ -	**op**
* / %	**op**
!	**not**
$-_{unário}$	**minus**
[ ]	**access**

Sintaxe concreta e abstrata para vários operadores Java.

Geração de Código Intermediário

* **Introdução**
  * Os operadores em uma linha possuem a mesma precedência; ou seja == e != têm a mesma precedência.
  * As linhas estão em ordem de precedência crescente; por exemplo, == possui maior precedência do que os operadores && e =.
  * O subscrito unário serve apenas para distinguir um sinal de subtração unário no início, como em -2, de um sinal de subtração binário, como em 2–a. O operador `[ ]` representa o acesso a arranjo, como em `a[i]`.

Geração de Código Intermediário

* **Introdução**
  * A coluna de sintaxe abstrata especifica o agrupamento de operadores.
  * O operador de atribuição = está em um grupo isolado.
  * O grupo **cond** contém os operadores condicionais boolianos && e ||.
  * O grupo **rel** contém os operadores de comparação relacionais nas linhas para == e <.
  * O grupo **op** contém os operadores aritméticos como + e *.
  * O operador de adição unário, a negação booliana e o acesso ao arranjo estão em grupos separados.

Geração de Código Intermediário

* **Introdução**
  * Durante a análise sintática, nós da **árvore sintática** são criados para representar construções de programação significativas.
  * À medida que a análise prossegue, informação é acrescentada aos nós na forma de atributos associados aos nós. A escolha dos atributos depende da tradução a ser realizada.

Geração de Código Intermediário

* **Introdução**
  * O **código de três endereços**, por outro lado, é uma sequência de etapas elementares do programa, como a adição de dois valores.
  * Diferentemente da árvore, não existe uma estrutura hierárquica. Conforme veremos na AULA07, precisamos dessa representação se tivermos de realizar qualquer otimização significativa do código.
  * Nesse caso, dividimos a sequência longa de comandos de três endereços, que forma um programa, em "blocos básicos", que são sequências de comandos sempre executados um após o outro, sem desvio.

Geração de Código Intermediário

* **Introdução**
  * Além de criar uma representação intermediária, um *front-end* de compilador verifica se o programa fonte segue as regras sintáticas e semânticas da linguagem fonte.
  * Essa verificação é chamada de *verificação estática*; em geral, "**estática**" significa "feita pelo **compilador**".
    * (Seu oposto, a verificação "**dinâmica**", significa "enquanto o programa está sendo **executado**". Muitas linguagens também fazem certas verificações dinâmicas. Por exemplo, uma linguagem orientada por objeto, como Java, às vezes precisa verificar os tipos durante a execução do programa, pois o método aplicado a um objeto pode depender da subclasse específica do objeto).
  * A verificação estática garante que certos tipos de erros de programação, incluindo incompatibilidade de tipo, são detectados e relatados durante a compilação.

Geração de Código Intermediário

* **Introdução**
  * É possível que um compilador construa uma árvore sintática ao mesmo tempo que emite trechos de código de três endereços.
  * Contudo, é comum que os compiladores emitam o código de três endereços enquanto o analisador "se movimenta" construindo uma árvore sintática, sem realmente construir a estrutura de dados completa da árvore.

Geração de Código Intermediário

* **Introdução**
  * Em vez disso, o compilador armazena nós e seus atributos necessários para a verificação semântica ou outras finalidades, junto com a estrutura de dados usada para análise.

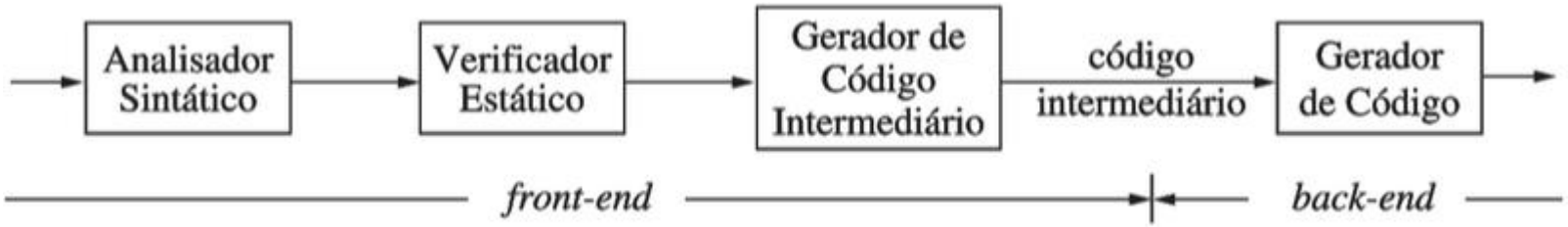
  * Fazendo isso, as partes da árvore sintática necessárias para construir o código de três endereços estão disponíveis quando necessárias, mas desaparecem quando não são mais necessárias.

Geração de Código Intermediário

* **Introdução**
  * Trataremos então dessas representações intermediárias, verificação de tipo estática e geração de código intermediário.

Geração de Código Intermediário

* **Introdução**
  * Para simplificar, assumimos que um *front-end* de um compilador é organizado como na Fig. 2, abaixo,

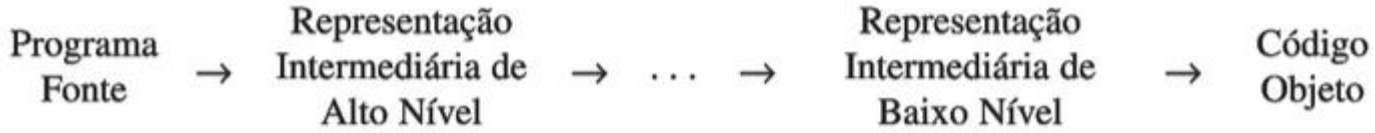


Estrutura lógica do *front-end* de um compilador.

  * em que a análise sintática, a verificação estática e a geração de código intermediário são feitas sequencialmente; às vezes, elas podem ser combinadas e incorporadas na análise sintática.

Geração de Código Intermediário

* **Introdução**
  * No processo de tradução de um programa, em uma dada linguagem fonte, para o código de determinada máquina alvo, um compilador pode construir uma sequência de representações intermediárias, como na Fig. 3, abaixo:



Um compilador pode usar uma seqüência de representações intermediárias.

  * As representações de alto nível estão próximas da linguagem fonte, e as representações de baixo nível estão próximas da máquina alvo.

Geração de Código Intermediário

* **Introdução**
  * As árvores de sintaxe são de alto nível, elas representam a estrutura hierárquica natural do programa fonte e são bem adequadas a tarefas como verificação de tipo estática.
  * Uma representação de baixo nível é adequada para tarefas dependentes de máquina, como a alocação de registradores e a seleção de instrução.
  * O código de três endereços pode variar de alto até baixo nível, dependendo da escolha dos operadores.

Geração de Código Intermediário

* **Introdução**
  * Para as expressões, as diferenças entre as árvores de sintaxe e o código de três endereços são superficiais.

  * Para os comandos de *looping*, por exemplo, uma árvore de sintaxe representa os componentes de um comando, enquanto o código de três endereços contém rótulos e comandos de desvios para representar o **fluxo de controle**, como na linguagem de máquina.

Geração de Código Intermediário

* **Introdução**
  * A escolha ou o projeto de uma representação intermediária varia de um compilador para outro.
  * Uma representação intermediária pode ser uma linguagem de alto nível corrente ou pode consistir em estruturas de dados internas que são compartilhadas pelas fases do compilador.

Geração de Código Intermediário

* **Introdução**
  * C é uma linguagem de programação de alto nível, embora frequentemente seja usada como uma forma intermediária, porque é flexível, compila para código de máquina eficiente e seus compiladores são encontrados com facilidade.

  * O *front-end* do compilador C++ original gera C, tratando o compilador C como um *back-end*.

Geração de Código Intermediário

* **Verificação estática**
  * As verificações estáticas são verificações de consistência que são feitas durante a compilação.
  * Elas não apenas garantem que um programa possa ser compilado com sucesso, mas também têm o poder de detectar os erros de programação mais cedo, antes que um programa seja executado.

Geração de Código Intermediário

* **Verificação estática**
  * A verificação estática inclui *verificação de tipo*, que garante que os operadores são aplicados a operandos compatíveis.
  * Também inclui quaisquer verificações sintáticas que restarem após a análise sintática.
    * Por exemplo, a verificação estática garante que um comando break em C esteja incorporado em um comando while, for ou switch; se não houver uma dessas instruções envolventes, um erro é informado.

Geração de Código Intermediário

* **Verificação estática**
  * A verificação estática inclui:

    * ***Verificação sintática***. Existe mais na sintaxe do que nas gramáticas. Por exemplo, as restrições, como um identificador ser declarado no máximo uma vez em um escopo, ou um comando *break* precisar ter uma instrução *loop* ou *switch* envolvendo-o, são sintáticas, embora não estejam codificadas ou impostas por uma gramática usada para o reconhecimento sintático.

Geração de Código Intermediário

* **Verificação estática**
  * A verificação estática inclui:

    * ***Verificação de tipo***. As regras de tipo associadas aos tipos de uma linguagem garantem que um operador ou função seja aplicado ao número e ao tipo correto de operandos. Se uma conversão entre tipos for necessária, por exemplo, em uma adição entre um inteiro e um float, o verificador de tipo pode inserir um operador na árvore sintática para representar essa conversão. A conversão de tipo geralmente é associada ao termo "coerção".

Geração de Código Intermediário

* **Verificação estática**
  * Valores-L e Valores-R
    * Vamos considerar algumas verificações estáticas simples que podem ser feitas durante a construção de uma árvore sintática para um programa fonte.
    * Em geral, as verificações estáticas complexas podem precisar ser feitas primeiro construindo uma representação intermediária e depois analisando-a.

Geração de Código Intermediário

* **Verificação estática**
  * Valores-L e Valores-R
    * Existe uma distinção entre o significado dos identificadores que aparecem no lado esquerdo e direito de uma atribuição.
    * Em cada uma das atribuições:

```
i = 5;
i = i + 1;
```

o lado direito especifica um valor inteiro, enquanto o lado esquerdo especifica onde o valor deve ser armazenado. Os termos *valor-l* e *valor-r* referem-se aos valores que são apropriados no lado esquerdo (*left*) e direito (*right*) de uma atribuição, respectivamente. Ou seja, valores-r são aqueles que normalmente pensamos como "valores", enquanto os valores-l são endereços.

Geração de Código Intermediário

* **Verificação estática**
  * Valores-L e Valores-R
    * A verificação estática precisa garantir que o lado esquerdo de uma atribuição indica um valor-l.
    * Um identificador como i tem um valor-l, semelhante ao acesso a um arranjo, como `a[2]`. Mas uma constante como 2 não é apropriada para o lado esquerdo de uma atribuição, pois possui um valor-r, mas não um valor-l.

Geração de Código Intermediário

* **Verificação estática**
  * **Verificação de Tipo**
    * A verificação de tipo garante que o tipo de uma construção case com aquele esperado por seu contexto.
    * Por exemplo, no comando **if**:

if (*expr*) *stmt*

espera-se que a expressão *expr* tenha o tipo **boolean**.

Geração de Código Intermediário

* **Verificação estática**
  * **Verificação de Tipo**
    * As regras de verificação de tipo seguem a estrutura operador/operando da sintaxe abstrata.
    * Considere que o operador **rel** represente operadores relacionais, como <=. A regra de tipo para o grupo de operadores **rel** é que seus dois operandos devam ter o mesmo tipo, e o resultado tem o tipo booliano.

Geração de Código Intermediário

- **Verificação estática**
  - **Verificação de Tipo**
    - Usando o atributo *type* para o tipo de uma expressão, considere que E consiste em **rel** aplicado a E_1 e E_2.
    - O tipo de E pode ser verificado quando seu nó é construído, executando o seguinte código:

if ($E_1.type == E_2.type$) $E.type = boolean$;

else error;

Geração de Código Intermediário

* **Verificação estática**
  * **Verificação de Tipo**
    * A ideia de casar o tipo real com o esperado continua aplicável, até mesmo nas seguintes situações:

    * *Coerções*. Uma *coerção* ocorre se o tipo de um operando for convertido automaticamente para o tipo esperado pelo operador. Em uma expressão como 2 * 3.14, a transformação usual é converter o inteiro 2 em um número de ponto flutuante equivalente, 2.0, e depois realizar uma operação de ponto flutuante sobre o par resultante de operandos de ponto flutuante. A definição da linguagem especifica as coerções permitidas. Por exemplo, a regra para **rel** discutida anteriormente poderia ser que E_1.*type* e E_2.*type* são convertidos para o mesmo tipo. Nesse caso, seria válido comparar, digamos, um inteiro com um float.

Geração de Código Intermediário

* **Verificação estática**
  * **Verificação de Tipo**
    * A ideia de casar o tipo real com o esperado continua aplicável, até mesmo nas seguintes situações:

    * *Sobrecarga*. O operador + em Java representa a adição quando aplicado a inteiros e significa concatenação quando aplicado a *strings*. Um símbolo é considerado *sobrecarregado* se tiver diferentes significados, cada um deles dependendo do seu contexto. Assim +, é sobrecarregado em Java. O significado de um operador sobrecarregado é determinado considerando-se os tipos conhecidos de seus operandos e resultados. Por exemplo, sabemos que o + em $z = x + y$ é uma concatenação se soubermos que qualquer um dentre x, y ou z são do tipo string. Porém, se soubermos também que um deles é do tipo inteiro, então temos um erro de tipo, e não existe significado nesse uso de +.

Geração de Código Intermediário

* **Verificação estática**
  * **Verificação de Tipo**
    * A verificação de tipo foi vista de forma aprofundada na AULA05.

    * Toda análise que envolve verificação de **tipo,** é automaticamente associada à fase de análise semântica. Cada etapa da 'verificação de tipo' pode ser tratada como uma ação semântica do compilador.

Geração de Código Intermediário

* **Verificação estática**
  * Podemos concluir que a verificação estática está intimamente relacionada à fase de análise semântica.
  * O projetista de compilador pode escolher realizar a verificação estática **durante** a geração de código intermediário ou **separadamente** na fase de análise semântica.

Geração de Código Intermediário

* **Código de três endereços**
  * Uma vez que as árvores sintáticas estão construídas, análise e síntese poderão ser feitas avaliando-se os atributos e executando-se os fragmentos de códigos nos nós da árvore.
  * Veremos as possibilidades percorrendo as árvores sintáticas para gerar o código de três operandos.

Geração de Código Intermediário

* **Código de três endereços**
  * O código de três endereços é construído a partir de dois conceitos: endereços e instruções.
  * Em termos da orientação por objetos, esses conceitos correspondem a classes, e os vários tipos de endereços e instruções correspondem a subclasses apropriadas.
  * Como alternativa, o código de três endereços pode ser implementado usando-se registros com campos para os endereços.

Geração de Código Intermediário

* **Código de três endereços**
  * Um endereço pode ser:

    * ***Um nome***. Por conveniência, permitimos que os nomes do programa fonte apareçam como endereços no código de três endereços. Em uma implementação, um nome do programa fonte é substituído por um apontador para sua entrada na tabela de símbolos, na qual estão contidas todas as informações sobre este nome.

Geração de Código Intermediário

* **Código de três endereços**
  * Um endereço pode ser:
    * ***Uma constante***. Na prática, um compilador deve tratar com muitos tipos diferentes de constantes e variáveis.
    * ***Um temporário gerado pelo compilador***. É vantajoso, especialmente em compiladores otimizados, criar um nome distinto toda vez que um temporário é necessário. Esses temporários podem ser combinados, se possível, quando os registradores são alocados a variáveis.

Geração de Código Intermediário

* **Código de três endereços**
  * O código de três endereços é uma sequência de instruções na forma:

$$x = y \ \mathbf{op} \ z$$

onde x, y e z são nomes, restrições ou temporários gerados pelo compilador; e **op** significa um operador.

Geração de Código Intermediário

* **Código de três endereços**
  * Os arranjos são tratados usando as duas variantes de instruções a seguir:

$$x[\,y\,] = z$$
$$x = y[\,z\,]$$

A primeira coloca o valor de z no endereço $x[y]$, e a segunda coloca o valor de $y[z]$ no endereço x.

Geração de Código Intermediário

* **Código de três endereços**
  * As instruções de três endereços são executadas em sequência numérica, a menos que sejam forçadas a fazer de outra forma por um desvio condicional ou incondicional.

  * Usaremos as seguintes instruções para fluxo de controle:

`ifFalse` *x* `goto L`	se *x* é falso, execute em seguida a instrução rotulada com `L`
`ifTrue` *x* `goto L`	se *x* é verdadeiro, execute em seguida a instrução rotulada com `L`
`goto L`	execute em seguida a instrução rotulada com `L`

  * Um rótulo `L` pode ser conectado a qualquer instrução iniciando-se com um prefixo L:. Uma instrução pode ter mais de um rótulo.

  * Finalmente, precisamos de instruções de três endereços que copiam um valor. Exemplo: $x = y$.

Geração de Código Intermediário

* **Código de três endereços**
  * Aqui está uma lista das formas mais comuns de instruções de três endereços:

  * 1) Instruções de atribuição da forma $x = y\ op\ z$, onde op é um operador aritmético binário ou lógico, e x, y e z são endereços.

  * 2) Atribuições da forma $x = op\ y$, onde op é um operador unário. Os operadores unários essenciais incluem o menos unário, a negação lógica, os operadores de deslocamento, e os operadores de conversão que, por exemplo, convertem um inteiro para um número de ponto flutuante.

  * 3) Instruções de cópia da forma $x = y$, onde se atribui a x o valor de y.

Geração de Código Intermediário

* **Código de três endereços**
  * Aqui está uma lista das formas mais comuns de instruções de três endereços:

  * 4) Um desvio incondicional `goto` L. A instrução de três endereços com rótulo L é a próxima a ser executada.

  * 5) Desvios condicionais da forma `if x goto` L e `ifFalse x goto` L. Essas instruções executam a instrução com rótulo *L* em seguida se *x* for verdadeiro e falso, respectivamente. Caso contrário executa-se a instrução de três endereços seguinte na sequência de instruções, como de costume.

Geração de Código Intermediário

* **Código de três endereços**
  * Aqui está uma lista das formas mais comuns de instruções de três endereços:

  * 6) Desvios condicionais tais como `if x relop y goto` L, que aplicam um operador relacional (<, ==, >= etc.) a x e y, e executam em seguida a instrução com rótulo L se x se colocar na relação *relop* com y. Senão, a instrução de três endereços após `if x relop y goto` L é executada em seguida, na sequência.

Geração de Código Intermediário

* **Código de três endereços**
  * Aqui está uma lista das formas mais comuns de instruções de três endereços:

  * 7) As chamadas de procedimento e retornos são implementadas usando as seguintes instruções: `param` *x* para parâmetros; `call` *p*, *n* e *y* = `call p, n` para chamadas de procedimento e função, respectivamente; e return *y*, onde *y*, representando um valor retornado, é opcional. O inteiro *n*, indicando o número de parâmetros reais em 'call p, n', não é redundante porque as chamadas podem ser aninhadas. Ou seja, alguns dos primeiros comandos `param` poderiam ser parâmetros de uma chamada que vem após *p* retornar seu valor; esse valor se torna outro parâmetro da segunda chamada.

Geração de Código Intermediário

* **Código de três endereços**
  * Aqui está uma lista das formas mais comuns de instruções de três endereços:

  * 8) Instruções indexadas de cópia da forma `x=y[i]` e `x[i]=y`. A instrução `x=y[i]` atribui a `x` o valor contido no endereço *i* unidades de memória além do endereço de `y`. A instrução `x[i]=y` atribui ao conteúdo do endereço *i* unidades além do `x` o valor de `y`.

  * 9) Atribuições de endereço e apontador da forma `x=&y`, `x=*y` e `*x=y`. A instrução `x=&y` define o valor-*r* de `x` para ser o endereço (valor-*l*) de `y`. Presumidamente `y` é um nome, talvez um temporário, que denota uma expressão com um valor-*l* como `A[i][j]`, e `x` é um nome de apontador ou um temporário.

Geração de Código Intermediário

* **Código de três endereços**
  * Aqui está uma lista das formas mais comuns de instruções de três endereços:

  * Na instrução `x=*y`, presumidamente `y` é um apontador ou um temporário cujo valor-*r* é um endereço. O valor-*r* de `x` é feito igual ao conteúdo deste endereço. Finalmente, `*x=y` define o valor-*r* do objeto apontado por `x` com o valor-*r* de `y`.

Geração de Código Intermediário

* **Código de três endereços**
  * Veja o exemplo abaixo:
    * Considere a instrução de alto nível:

```
do i = i + 1; while (a[i] < v);
```

  * Duas traduções possíveis dessa instrução aparecem na Fig. 4. A tradução na Fig. 4 utiliza um rótulo simbólico L, conectado à primeira instrução. A tradução em (b) mostra números com as posições das instruções, começando arbitrariamente na posição 100. Nas duas traduções, a última instrução é um desvio condicional para a primeira instrução.

Geração de Código Intermediário

* **Código de três endereços**
  * Veja o exemplo abaixo:
    * Considere a instrução de alto nível:

```
do i = i + 1; while (a[i] < v);
```

    * Abaixo, Fig.4, duas formas de atribuir rótulos às instruções de três endereços.

```
L:  t1 = i + 1
    i = t1
    t2 = i
    t3 = a [ t2 ]
    if t3 < v goto L
```

(a) Rótulos simbólicos

```
100:  t1 = i + 1
101:  i = t1
102:  t2 = i
103:  t3 = a [ t2 ]
104:  if t3 < v goto 100
```

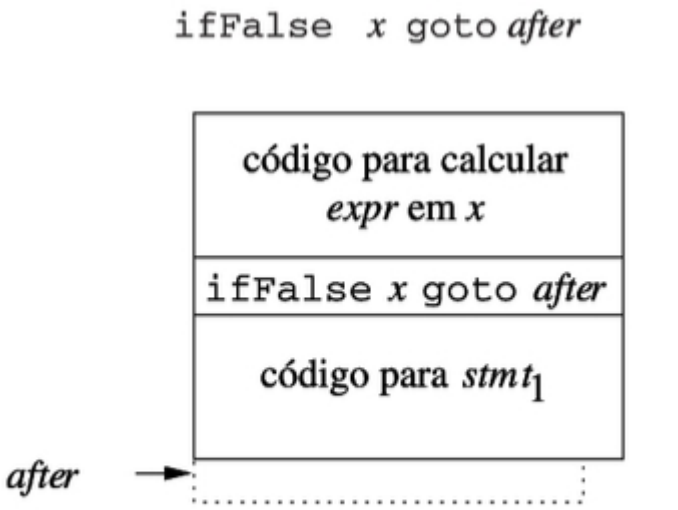
(b) Posição numérica

Geração de Código Intermediário

* **Código de três endereços**
  * A escolha dos operadores primários é uma questão importante no projeto de uma forma intermediária.
  * O conjunto de operadores claramente precisa ser rico o suficiente para implementar as operações da linguagem fonte.
  * A proximidade dos operadores com as instruções da máquina facilitam a implementação da forma intermediária em uma máquina alvo.
  * Contudo, se o *front-end* tiver de gerar longas sequências de instruções para algumas operações da linguagem fonte, então o otimizador e o gerador de código podem ter de trabalhar mais para redescobrir a estrutura e gerar um bom código para essas operações.

Geração de Código Intermediário

* **Código de três endereços**
  * Tradução de comandos:
    * Os comandos são traduzidos para código de três endereços usando instruções de desvio para implementar o **fluxo de controle** através do comando.
    * O layout da Fig. 5, ilustra a tradução de **`if`** *`expr`* **`then`** *`stmt`*:



Layout de código para o comando if

Geração de Código Intermediário

* **Código de três endereços**
  * Tradução de comandos:
    * O comando de desvio no layout pula a tradução de *stmt* se *expr* for avaliado como **false.**
    * Outras construções de comandos são semelhantemente traduzidas usando saltos apropriados para transferir o controle ao redor do código de seus constituintes.

Geração de Código Intermediário

* **Código de três endereços**
  * Tradução de expressões:
    * Agora, ilustraremos a tradução de expressões considerando as expressões contendo operadores binários **op**, acessos a arranjos e atribuições, além de constantes e identificadores.
    * Para simplificar, em um acesso a arranjo $y[z]$, exigimos que y seja um identificador.
    * Usaremos a técnica simples de gerar uma instrução de três endereços para cada nó operador na árvore sintática para uma expressão. Nenhum código é gerado para identificadores e constantes, pois eles podem aparecer como endereços nas instruções.

Geração de Código Intermediário

* **Código de três endereços**
  * Tradução de expressões:
    * Se um nó *x* da classe *Expr* tiver operador **op**, então uma instrução é emitida para calcular o valor no nó *x* para um nome "temporário" gerado pelo compilador, digamos, *t*.

    * Assim, `i-j+k` é traduzido para duas instruções:

```
t1 = i - j
t2 = t1 + k
```

Geração de Código Intermediário

* **Código de três endereços**
  * Tradução de expressões:
    * Com os acessos a arranjos e atribuições, é necessário distinguir entre *valores-l* e *valores-r*.
    * Por exemplo, `2 * a[i]` pode ser traduzido calculando o valor-r de a[i] em um temporário, como em:

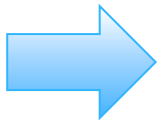
```
t1 = a [ i ]
t2 = 2 * t1
```

    * No entanto, não podemos simplesmente usar um temporário no lugar de `a[i]`, se `a[i]` aparecer do lado esquerdo de uma atribuição.

Geração de Código Intermediário

* **Código de três endereços**
  * Tradução de expressões:
    * Vejamos outro exemplo:
    * Aplicar a geração de código intermediário de três endereços à árvore sintática para a expressão `a[i] = 2 * a[j-k]`:

```
a[i] = 2*a[j-k]
```



```
t3 = j - k
t2 = a [ t3 ]
t1 = 2 * t2
a [ i ] = t1
```

Geração de Código Intermediário

* **Comandos de fluxo de controle**
  * A tradução de comandos, como if-else e while, está ligada à tradução das expressões boolianas. Nas linguagens de programação, as expressões boolianas normalmente são usadas para:
    * 1) *Alterar o fluxo de controle*. As expressões boolianas são usadas como expressões condicionais em comandos que alteram o fluxo de controle. O valor dessas expressões boolianas é dado implicitamente pela posição atingida em um programa. Por exemplo, em **`if`** `(`E`)` `S`, a expressão E deve ser verdadeira se o comando S for alcançado.

Geração de Código Intermediário

* **Comandos de fluxo de controle**
  * A tradução de comandos, como if-else e while, está ligada à tradução das expressões boolianas. Nas linguagens de programação, as expressões boolianas normalmente são usadas para:
    * 2) *Computador valores lógicos*. Uma expressão booliana pode representar *true* (verdadeiro) ou *false* (falso) como valores. Essas expressões boolianas podem ser avaliadas em analogia com as expressões aritméticas usando instruções de três endereços com operadores lógicos.

Geração de Código Intermediário

* **Comandos de fluxo de controle**
  * O uso intencional das expressões boolianas é determinado por seu contexto sintático.
  * Por exemplo, uma expressão após a palavra-chave **if** é usada para alterar o fluxo de controle, enquanto outra, do lado direito de um comando de atribuição, é usada para denotar um valor lógico.

Geração de Código Intermediário

* **Comandos de fluxo de controle**
  * Esses contextos sintáticos podem ser especificados de diversas maneiras: podemos usar dois não-terminais diferentes, usar atributos herdados ou ativar um flag durante a análise.

  * Como alternativa, podemos construir uma árvore de sintaxe e invocar diferentes procedimentos para dois usos diferentes das expressões boolianas.

Geração de Código Intermediário

* **Comandos de fluxo de controle**
  * Vamos nos concentrar no uso de expressões boolianas para alterar o fluxo de controle.
  * Por questão de clareza, usaremos um novo não-terminal B para essa finalidade.

Geração de Código Intermediário

* **Comandos de fluxo de controle**
  * As expressões boolianas são compostas dos operadores boolianos (os quais denotamos com &&, || e !, usando a convenção da linguagem C para os operadores AND, OR e NOT, respectivamente) aplicados a elementos que são variáveis boolianas ou expressões relacionais.

  * As expressões relacionais são da forma E_1 **rel** E_2, onde E_1 e E_2 são expressões aritméticas.

Geração de Código Intermediário

* **Comandos de fluxo de controle**
  * Consideraremos as expressões boolianas geradas pela seguinte gramática:

$$B \rightarrow B \,||\, B \mid B \,\&\&\, B \mid !\, B \mid (\, B \,) \mid E \ \mathbf{rel}\ E \mid \mathbf{true} \mid \mathbf{false}$$

  * Usamos o atributo **rel**.*op* para indicar qual dos seis operadores de comparação <, <=, =, !=, > ou >= é representado por **rel**. Como é comum, assumimos que || e && sejam associativos à esquerda, e que || tenha a precedência mais baixa, então && e então !.

Geração de Código Intermediário

* **Comandos de fluxo de controle**
  * Dada a expressão B_1 || B_2, se determinarmos que B_1 é verdadeiro, podemos concluir que a expressão inteira é verdadeira sem ter de avaliar B_2.
  * De forma semelhante, dado B_1 && B_2, se B_1 for falso, então a expressão inteira é falsa.

Geração de Código Intermediário

* **Comandos de fluxo de controle**
  * A definição semântica da linguagem de programação determina se todas as partes de uma expressão booliana precisam ser avaliadas.

  * Se a definição da linguagem permitir (ou exigir) que partes de uma expressão booliana não sejam avaliadas, então o compilador pode otimizar a avaliação de expressões boolianas computando apenas o suficiente de uma expressão para determinar seu valor.

Geração de Código Intermediário

* **Comandos de fluxo de controle**
  * Assim, em uma expressão como $B_1 \,||\, B_2$, nem B_1, nem B_2 é necessariamente avaliado em sua totalidade.
  * Se B_1 ou B_2 for uma expressão com efeitos colaterais (por exemplo, se contiver uma função que muda o valor de uma variável global), então uma resposta inesperada pode ser obtida.

Geração de Código Intermediário

* **Comandos de fluxo de controle**
  * Nó código de desvio, os operadores boolianos &&, || e ! são traduzidos para desvios.
  * Os próprios operadores não aparecem no código; em vez disso, o valor de uma expressão booliana é representado por uma posição na sequência de código.

Geração de Código Intermediário

* **Comandos de fluxo de controle**
  * Vejamos um exemplo:
    * O comando de alto nível:

```
if ( x < 100 || x > 200 && x != y ) x = 0;
```

    * poderia ser traduzido para o código de três endereços da Fig. 6, abaixo. Nessa tradução, a expressão booliana é verdadeira se o controle alcançar o rótulo L_2. Se a expressão for falsa, o controle vai imediatamente para L_1, saltando L_2 e a atribuição `x = 0`.

```
        if x < 100 goto L2
        ifFalse x > 200 goto L1
        ifFalse x != y goto L1
L2:     x = 0
L1:
```

Código de desvio.

Geração de Código Intermediário

* **Comandos de fluxo de controle**
  * Agora, vamos considerar a tradução de expressões boolianas para um código de três endereços no contexto dos comandos como aqueles gerados pela gramática a seguir:

$$
\begin{aligned}
S &\rightarrow \textbf{if}\ (\ B\ )\ S_1 \\
S &\rightarrow \textbf{if}\ (\ B\ )\ S_1\ \textbf{else}\ S_2 \\
S &\rightarrow \textbf{while}\ (\ B\ )\ S_1
\end{aligned}
$$

Geração de Código Intermediário

* **Comandos de fluxo de controle**
  * Nessas produções, o não-terminal *B* representa uma expressão booliana e o não-terminal *S* representa um comando.

  * Com uma expressão booliana *B*, associamos dois rótulos: *B.true*, rótulo para o qual o controle flui se *B* for verdadeiro, e *B.false*, rótulo para o qual o controle flui se *B* for falso.

Geração de Código Intermediário

* **Comandos de fluxo de controle**
  * Uma expressão booliana *B* é traduzida para instruções de três endereços que avaliam *B* usando desvios condicionais e incondicionais para um dos dois rótulos: *B.true* e *B.false*.

  * A regra de produção $B \rightarrow E_1 \ \mathbf{rel} \ E_2$, é traduzida diretamente para uma instrução de três endereços de comparação, com desvios para os endereços apropriados. Por exemplo *B* da forma $a < b$ se traduz para:

```
if a < b goto B.true
goto B.false
```

Geração de Código Intermediário

* **Comandos de fluxo de controle**
  * As produções restantes para *B* são traduzidas da seguinte forma:

  * 1) Suponha que *B* seja da forma $B_1 \mid\mid B_2$. Se B_1 for verdadeiro, então sabemos imediatamente que o próprio *B* é verdadeiro, assim $B_1.true$ é o mesmo que $B.true$. Se B_1 for falso, então B_2 deve ser avaliado, de modo que fazemos $B_1.false$ ser o rótulo da primeira instrução no código de B_2. As saídas, verdadeiro e falso, de B_2 são iguais às saídas verdadeiro e falso de *B*, respectivamente.

Geração de Código Intermediário

* **Comandos de fluxo de controle**
  * As produções restantes para *B* são traduzidas da seguinte forma:
  * 2) A tradução de $B_1 \,\&\&\, B_2$ é semelhante.
  * 3) Nenhum código é necessário para uma expressão *B* na forma $!B_1$: apenas troque as saídas, verdadeiro e falso, de B_1 para obter as saídas, verdadeiro e falso, de *B*.
  * 4) As constantes **true** e **false** são traduzidas para desvios para $B.true$ e $B.false$, respectivamente.

Geração de Código Intermediário

* **Comandos de fluxo de controle**
  * Considere novamente o comando a seguir.
    * O comando de alto nível:

```
if ( x < 100 || x > 200 && x != y ) x = 0;
```

    * poderia ser traduzido para o código de três endereços da Fig. 7, abaixo, usando as definições vistas:

```
        if x < 100 goto L2
        goto L3
L3:     if x > 200 goto L4
        goto L1
L4:     if x != y goto L2
        goto L1
L2:     x = 0
L1:
```

Tradução do fluxo de controle de um comando if simples.

Geração de Código Intermediário

* **Comandos de fluxo de controle**
  * Observe que o código gerado não é ótimo, porque a tradução tem três instruções (`gotos`) a mais que o código da Fig. 6.
  * A instrução goto L_3 é redundante, pois L_3 é o rótulo da próxima instrução. As duas instruções `goto` L_1 podem ser eliminadas se for usado `ifFalse` no lugar das instruções `if`, como na Fig. 6.

Geração de Código Intermediário

* **Comandos de fluxo de controle**
  * Na Fig. 7, a comparação x > 200 é traduzida para o fragmento de código:

```
        if x > 200 goto L4
        goto L1
L4:     ...
```

  * Em vez disso, considere a instrução:

```
        ifFalse x > 200 goto L1
L4:     ...
```

  * Essa instrução `iffalse` tira proveito do fluxo natural de uma instrução para a próxima na sequência, de modo que o controle simplesmente 'segue' para o rótulo L_4 se x > 200 for falso, evitando assim um desvio.

Geração de Código Intermediário

* **Comandos de fluxo de controle**
  * Comandos **break, continue** e **goto**
    * A construção de linguagem de programação mais elementar para alterar o fluxo de controle em um programa é o comando `goto`.
    * Em C, um comando como `goto L` desvia o controle para o comando rotulado com `L` – é preciso haver exatamente um comando com rótulo `L` nesse escopo.

Geração de Código Intermediário

* **Comandos de fluxo de controle**
  * Comandos **break**, **continue** e **goto**
    * Os comandos `goto` podem ser implementados mantendo-se uma lista de desvios não preenchidos para cada rótulo e então, **remendando-os** com o destino quando ele for conhecido.

    * Java não possui comandos `goto`. Contudo, Java permite desvios disciplinados, chamados de comandos **break**, que desviam o controle para fora de uma construção envolvente, e comandos **continue**, que disparam a próxima iteração de um *loop* envolvente.

Geração de Código Intermediário

* **Comandos de fluxo de controle**
  * Comandos **break, continue** e **goto**
    * O comando **continue** pode ser tratado de maneira semelhante ao comando **break.**
    * A diferença principal entre os dois é que o destino do desvio gerado é diferente.

Geração de Código Intermediário

* **Comandos de fluxo de controle**
  * Comandos **switch**
    * O comando ‘switch’ ou ‘case’ está disponível em diversas linguagens. Veja a sintaxe para o comando switch na Fig. 8, abaixo:

```
switch ( E ) {
        case V1: S1
        case V2: S2
            ...
        case Vn-1: Sn-1
        default: Sn
}
```

Sintaxe do comando switch.

Geração de Código Intermediário

* **Comandos de fluxo de controle**
  * Comandos **switch**
    * Existe uma expressão seletora E, que deve ser avaliada, seguida por n valores constantes V_1, V_2, ... , V_n que a expressão poderia assumir, talvez incluindo um 'valor' default, o qual sempre casa com a expressão, se nenhum outro valor casar.

Geração de Código Intermediário

* **Comandos de fluxo de controle**
  * Comandos **switch**
    * A tradução que se quer para um comando switch é um código para:

    * 1) Avaliar a expressão E.
    * 2) Encontrar o valor V_j na lista de casos que seja o mesmo que o valor da expressão. Lembre-se de que o valor default casa com a expressão se nenhum dos valores mencionados explicitamente nos casos casar.
    * 3) Executar o comando S_j associado ao valor encontrado.

Geração de Código Intermediário

* **Comandos de fluxo de controle**
  * Comandos **switch**
    * O caso (2) é um desvio de n vias, que pode ser implementado de diversas maneiras. Se o número de casos for pequeno, digamos 10 no máximo, então é razoável usar uma sequência de desvios condicionais, cada um testando um valor individual e transferindo-se para o código do comando correspondente.
      * Se o número de valores ultrapassar 10 ou mais, é mais eficiente construir uma tabela **hash** para os valores, com os rótulos dos diversos comandos como entradas. Se nenhuma entrada para o valor da expressão switch for encontrada, será gerado um desvio para o comando default.

Geração de Código Intermediário

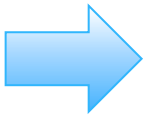
* **Comandos de fluxo de controle**
  * Comandos **switch**
    * O código intermediário da Fig. 9, a seguir, é uma tradução conveniente do comando switch da Fig. 8.
    * Todos os testes aparecem no fim, de modo que um gerador de código simples pode reconhecer o desvio multicaminho e gerar código eficiente para ele, usando a implementação mais apropriada.

Geração de Código Intermediário

* **Comandos de fluxo de controle**
  * Comandos **switch**
    * Abaixo, Fig's. 8 e 9:

```
switch ( E ) {
        case V1: S1
        case V2: S2
            ...
        case Vn-1: Sn-1
        default: Sn
}
```

Sintaxe do comando switch.



```
            código para avaliar E em t
            goto test
L1:         código para S1
            goto next
L2:         código para S2
            goto next
            ...
Ln-1:       código para Sn-1
            goto next
Ln:         código para Sn
            goto next
test:       if t = V1 goto L1
            if t = V2 goto L2
            ...
            if t = Vn-1 goto Ln-1
            goto Ln
next:
```

Tradução de um comando switch.

Geração de Código Intermediário

* **Código intermediário para procedimentos**
  * Os procedimentos e sua implementação também geram código intermediário de três endereços.
  * Usaremos o termo **função** para um procedimento que retorna um valor.
  * Discutiremos rapidamente as declarações de função e o código de três endereços para chamadas de função.

Geração de Código Intermediário

* **Código intermediário para procedimentos**
  * No código de três endereços, uma chamada de função é desmembrada na avaliação dos parâmetros em preparação para uma chamada, seguida pela própria chamada.
  * Para simplificar, consideraremos que os parâmetros são passados por valor.

Geração de Código Intermediário

* **Código intermediário para procedimentos**
  * Vejamos um exemplo:
    * Suponha que `a` seja um arranjo de inteiros e que `f` seja uma função de inteiros para inteiros. Então, a atribuição de alto nível:

```
n = f(a[i]);
```

    * poderia ser traduzida para o seguinte código de três endereços:

```
1) t1 = i
2) t2 = a [ t1 ]
3) param t2
4) t3 = call f, 1
5) n = t3
```

Geração de Código Intermediário

* **Código intermediário para procedimentos**
  * As duas primeiras linhas computam o valor da expressão `a[i]` para o temporário t2.
  * A linha 3 faz t2 ser um parâmetro real para a chamada na linha 4 de f com um parâmetro.
  * A linha 4 atribui o valor retornado pela chamada de função a t3.
  * A linha 5 atribui o valor retornado a n.

Geração de Código Intermediário

* **Código intermediário para procedimentos**
  * Observações importantes:

    * *Verificação de tipo*. Dentro das expressões, uma função é tratada como qualquer outro operador. Portanto, a discussão sobre verificação de tipo, vista anteriormente, pode ser aproveitada, inclusive as regras para coerções. Por exemplo, se f é uma função com um parâmetro do tipo real, então o inteiro 2 é convertido para um real na chamada $f(2)$.

Geração de Código Intermediário

* **Código intermediário para procedimentos**
  * Observações importantes:

    * *Chamadas de função*. Ao gerar instruções de três endereços para uma chamada de função **id**(E, E, ..., E), é suficiente gerar as instruções de três endereços para avaliar ou reduzir os parâmetros. E para endereços, seguidos por uma instrução `param` para cada parâmetro.

Geração de Código Intermediário

* **Código intermediário para procedimentos**
  * O procedimento é uma construção de programação tão importante e tão frequentemente utilizada que é imperativo para um compilador que ele gere um bom código para chamadas e retornos de procedimento.

  * As rotinas em tempo de execução que tratam passagem de parâmetro de procedimento, chamadas e retornos, fazem parte do pacote de **suporte em tempo de execução**.

Resumo

* As técnicas vistas até aqui podem ser combinadas para construir um *front-end* de um compilador simples. O *front-end* pode ser desenvolvido de forma incremental:

* ***Selecione uma representação intermediária***: Uma representação intermediária tipicamente é alguma combinação de uma notação gráfica e código de três endereços. Como nas árvores de sintaxe, um nó em uma notação gráfica representa uma construção; os filhos de um nó representam suas subconstruções. O código de três endereços tem seu nome a partir das instruções no formato $x = y \ \mathbf{op} \ z$, com no máximo um operador por instrução. Existem instruções adicionais para fluxo de controle.

Resumo

* ***Traduza expressões***: As expressões com várias operações podem ser desmembradas em uma sequência de operações individuais conectando-se ações a cada produção da forma E -> E_1 **op** E_2. A ação ou cria um nó para E com os nós para E_1 e E_2 como filhos, ou gera uma instrução de três endereços, que aplica **op** aos endereços para E_1 e E_2 e coloca o resultado em um novo nome temporário, o qual se torna o endereço para E.

* ***Verifique os tipos***: O tipo de uma expressão E_1 **op** E_2 é determinado pelo operador **op** e pelos tipos de E_1 e E_2. Uma coerção é uma conversão de tipo implícita, como de *integer* para *float*. O código intermediário contém conversões de tipo explícitas para garantir um casamento exato entre os tipos dos operandos e os tipos esperados por um operador.

Resumo

* ***Use uma tabela de símbolos para implementar declarações***: Uma declaração especifica o tipo de um nome. A largura de um tipo é a quantidade de memória necessária para um nome com esse tipo. Usando larguras, o endereço relativo de um nome em tempo de execução pode ser calculado com um deslocamento a partir do início da área de dados. O tipo e o endereço relativo de um nome são colocados na tabela de símbolos devido a sua declaração, assim o tradutor pode mais tarde recuperá-los quando o nome aparecer em uma expressão.

* ***Linearidade dos arranjos***: Para ter um acesso rápido, os elementos de um arranjo são armazenados em endereços consecutivos. Os arranjos de arranjos são armazenados linearmente, de modo que possam ser tratados como um arranjo unidimensional de elementos individuais. O tipo de um arranjo é usado para calcular o endereço de um elemento do arranjo relativo à base do arranjo.

Resumo

* ***Gere código de desvio para expressões boolianas***: No código de desvio, o valor de uma expressão booliana está implícito na posição alcançada no código. O código de desvio é útil porque uma expressão booliana *B* tipicamente é usada para fluxo de controle, como em **if** (*B*) *S*. Os valores boolianos podem ser traduzidos desviando-se para `t = true` ou `t = false`, conforme o caso, onde `t` é um nome temporário. Usando rótulos de desvios, uma expressão booliana pode ser traduzida pelos rótulos herdados correspondentes às suas saídas de verdadeiro e falso. As constantes *true* e *false* são traduzidas para um desvio para as saídas de verdadeiro e falso, respectivamente.

* ***Implemente comandos usando o fluxo de controle***: Os comandos podem ser traduzidos com base nos rótulos. O comando condicional S -> **if** (*B*) *S*1 pode ser traduzido conectando-se um novo rótulo que marca o início do código para *S*1, e passando-se o novo rótulo para as saídas de verdadeiro e falso de *B*, respectivamente.

Resumo

* ***Como alternativa, use o remendo***: Remendo é uma técnica para gerar código para expressões boolianas e comandos em um passo. A ideia é manter listas de desvios incompletos, onde todas as instruções de desvios na lista possuem o mesmo destino. Quando o destino se tornar conhecido, todas as instruções em sua lista serão completadas com o preenchimento do destino.

* ***Implemente registros***: Os nomes dos campos em um registro ou classe podem ser tratados como uma sequência de declarações. Um tipo registro codifica os tipos e endereços relativos dos campos. Um objeto da tabela de símbolos pode ser usado para essa finalidade.

Obrigado.

joaopauloaramuni@gmail.com
joaopauloaramuni@fumec.br

UNIVERSIDADE FUMEC
50 Anos