Aula 07 – Geração de Código

Prof. Eduardo Zambon

Departamento de Informática (DI)
Centro Tecnológico (CT)
Universidade Federal do Espírito Santo (Ufes)

Compiladores
***Compiler Construction* (CC)**

Introdução

- O *front-end* do compilador realiza as análises léxicas, sintáticas e semânticas do programa fonte e constrói uma representação intermediária (IR) do código.
- O *back-end* analisa a IR e gera código-alvo para uma dada arquitetura.
- **Estes *slides***: discussão geral sobre os principais métodos de geração de código.
- **Objetivos**: apresentar as ações principais necessárias para geração de código.

Referências

Chapter 8 – Code Generation
K. C. Louden

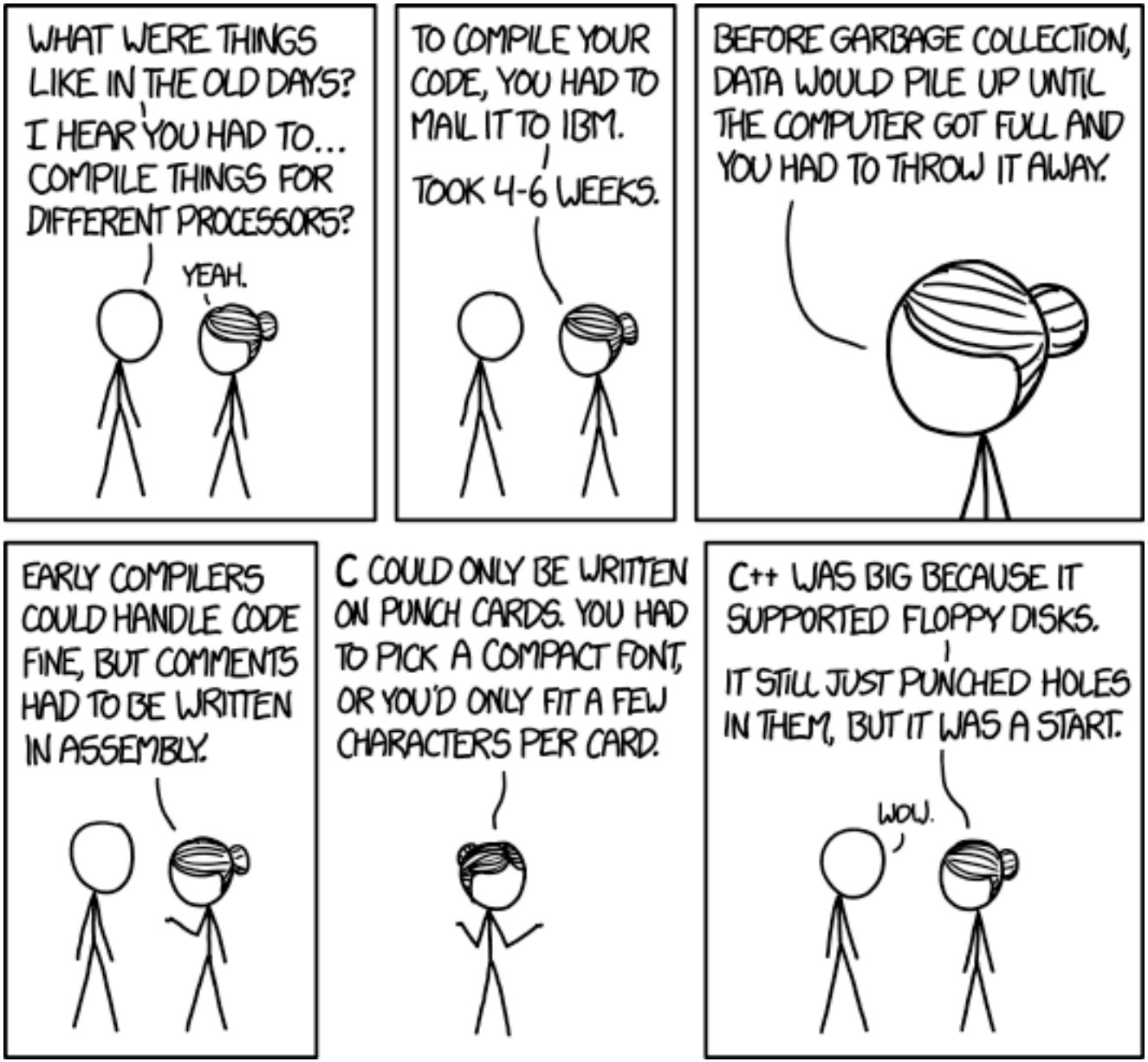
Chapter 7 – Code Shape
K. D. Cooper

Chapters 10 & 11
D. Thain

Introdução

- Objetivo fundamental de um compilador é gerar código executável.
- O código gerado para uma máquina alvo deve estar correto, i.e., deve capturar corretamente a semântica do código fonte, conforme a especificação da LP de entrada.
- Processo de geração de código depende:
  - Da semântica da linguagem fonte.
  - Das características da arquitetura alvo.
  - Da estrutura do ambiente de execução.
  - Do SO rodando na máquina alvo.

Old Days – http://xkcd.com/1755/



Representação e Código Intermediário

- Uma estrutura de dados que representa o programa fonte durante a tradução é chamada de representação intermediária (*intermediate representation* – IR).
- IRs podem ser categorizadas segundo a estrutura em:
  - ***Graph-based***: codificam o programa fonte em um grafo.
  - **Lineares**: "*assembly*" para uma máquina abstrata.
  - **Híbridas**: combinação de elementos de ambas.
- IRs baseadas em grafos podem ser de variados tipos: *parse trees*, ASTs, DAGs, grafos de dependências, etc.
- IRs lineares mais comuns: código de um, dois ou três endereços.
- Diferentes módulos do compilador utilizam diferentes IRs.
- Para geração de código, IRs lineares são mais convenientes.
- IRs lineares são chamadas de código intermediário (IC).

Código Intermediário

Considere a expressão `a-2*b`:

Código de um endereço (*one-address code*):

```
push 2
push b
mult
push a
sub
```

Código de dois endereços (*two-address code*):

```
load r1, 2
load r2, b
mult r2, r1
load r1, a
sub  r1, r2
```

Código de três endereços (*three-address code*):

```
load r1, 2
load r2, b
mult r3, r1, r2
load r4, a
sub  r5, r4, r3
```

- Código de um endereço é baseado em pilha.
- Código de dois endereços é baseado em operações destrutivas em registradores. Comum em máquinas CISC.
- Código de três endereços é baseado em operações não-destrutivas em registradores. Comum em máquinas RISC.

Código de Três Endereços

- Em código de três endereços, a maioria das operações tem a forma `i = j op k`, onde um operador `op` recebe dois operandos `i` e `k` e produz o resultado em `i`.
- Notações alternativas: `i` $\leftarrow$ `j op k` ou `op j, k` $\Rightarrow$ `i`.
- *Exemplo*: o código do slide anterior nesta notação fica como a seguir:

```
t1 = 2
t2 = b
t3 = t1 * t2
t4 = a
t5 = t4 - t3
```

- Os nomes `t[1-5]` são ditos temporários.
- A tarefa de decidir aonde cada temporário fica armazenado (registrador ou memória) chama-se alocação de registradores (veja Aula 08).

P-code

- *P-code* surgiu como um padrão de código *assembly* produzido por compiladores Pascal entre 1970 e 1980.
- *P-code* foi projetado como a linguagem de uma máquina hipotética baseada em pilha, chamada *P-machine*.
- Assim, *P-code* é um exemplo de implementação de código de um endereço (*one-address code*).
- Introduziu o conceito de portabilidade e máquina virtual: somente o interpretador de *P-code* precisava ser reescrito para uma nova arquitetura alvo.
- Como *P-code* foi projetado para ser executado, há uma suposição implícita de um ambiente de execução particular.
- A especificação de uma *P-machine* inclui o tamanho dos tipos de dados, organização da memória, etc.

P-code

- Uma *P-machine* é formada por:
  - Uma memória de código.
  - Uma memória de dados para as variáveis.
  - Uma pilha para dados temporários.
  - Um conjunto mínimo de registradores para manter a pilha e guiar a execução (*sp*, *fp* e *pc*).
- Exemplo de *P-code* para a expressão `2*a+(b-3)`.

```
ldc 2          ; load constant 2
lod a          ; load value of variable a
mpi            ; integer multiplication
lod b          ; load value of variable b
ldc 3          ; load constant 3
sbi            ; integer subtraction
adi            ; integer addition
```

As operações aritméticas não possuem operandos explícitos porque atuam sobre os valores na pilha.

Técnicas Básicas de Geração de Código (*P-code*)

- Geração de código intermediário pode ser vista como uma computação de atributos.
- Código gerado é um atributo sintetizado do tipo *string* que pode ser definido por uma gramática de atributos e gerado por um caminhamento em pós-ordem da AST.
- Considere, por exemplo, um subconjunto das operações aritméticas em C (|| = cat w/ space, ++ = cat w/ enter).

Grammar Rule	Semantic Rules
$exp_1 \rightarrow$ **id** = exp_2	exp_1 .pcode = "**lda**" || **id** .strval ++ exp_2 .pcode ++ "**stn**"
$exp \rightarrow aexp$	exp .pcode = $aexp$ .pcode
$aexp_1 \rightarrow aexp_2$ + $factor$	$aexp_1$ .pcode = $aexp_2$ .pcode ++ $factor$ .pcode ++ "**adi**"
$aexp \rightarrow factor$	$aexp$ .pcode = $factor$ .pcode
$factor \rightarrow$ (exp)	$factor$ .pcode = exp .pcode
$factor \rightarrow$ **num**	$factor$ .pcode = "**ldc**" || **num** .strval
$factor \rightarrow$ **id**	$factor$ .pcode = "**lod**" || **id** .strval

Gramática de Atributos para *3-Address Code* (TAC)

Grammar Rule	Semantic Rules
$exp_1 \rightarrow \texttt{id} = exp_2$	$exp_1.name = exp_2.name$ $exp_1.tacode = exp_2.tacode \; ++$ $\texttt{id}.strval \parallel \text{“=”} \parallel exp_2.name$
$exp \rightarrow aexp$	$exp.name = aexp.name$ $exp.tacode = aexp.tacode$
$aexp_1 \rightarrow aexp_2 + factor$	$aexp_1.name = newtemp()$ $aexp_1.tacode =$ $aexp_2.tacode \; ++ \; factor.tacode$ $++ \; aexp_1.name \parallel \text{“=”} \parallel aexp_2.name$ $\parallel \text{“+”} \parallel factor.name$
$aexp \rightarrow factor$	$aexp.name = factor.name$ $aexp.tacode = factor.tacode$
$factor \rightarrow (\; exp \;)$	$factor.name = exp.name$ $factor.tacode = exp.tacode$
$factor \rightarrow \texttt{num}$	$factor.name = \texttt{num}.strval$ $factor.tacode = \text{""}$
$factor \rightarrow \texttt{id}$	$factor.name = \texttt{id}.strval$ $factor.tacode = \text{""}$

Técnicas Básicas de Geração de Código

- Gramáticas de atributos simples como as apresentadas nos slides anteriores podem ser implementadas diretamente como ações semânticas no *parser*.
- Na prática, a geração de código exige um caminhamento pela AST, como ilustrado pelo pseudo-código abaixo.

```
procedure genCode ( T: treenode );
begin
   if T is not nil then
      generate code to prepare for code of left child of T ;
      genCode(left child of T) ;
      generate code to prepare for code of right child of T ;
      genCode(right child of T) ;
      generate code to implement the action of T ;
end;
```

- Código acima foi apresentado para uma árvore binária mas pode ser expandido para um número arbitrário de filhos.

Técnicas Básicas de Geração de Código

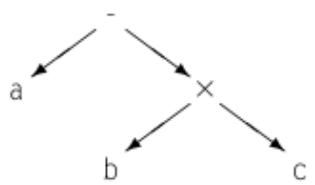
- Exemplo abaixo mostra a implementação de geração de código para expressões aritméticas.
- Recebe como entrada uma AST e emite como resultado TAC (formato ILOC, livro do Cooper).
- Chamada de `NextRegister()` somente incrementa e retorna um contador global de temporários.

```
expr(node) {
  int result, t1, t2;
  switch(type(node)) {
    case ×, ÷, +, -:
      t1 ← expr(LeftChild(node));
      t2 ← expr(RightChild(node));
      result ← NextRegister( );
      emit(op(node), t1, t2, result);
      break;

    case IDENT:
      t1 ← base(node);
      t2 ← offset(node);
      result ← NextRegister( );
      emit(loadAO, t1, t2, result);
      break;

    case NUM:
      result ← NextRegister( );
      emit(loadI, val(node), none,
           result);
      break;
  }
  return result;
}
```

(a) Treewalk Code Generator



(b) Abstract Syntax Tree for

a - b x c

```
loadI   @a       ⇒ r1
loadAO  rarp, r1 ⇒ r2
loadI   @b       ⇒ r3
loadAO  rarp, r3 ⇒ r4
loadI   @c       ⇒ r5
loadAO  rarp, r5 ⇒ r6
mult    r4, r6   ⇒ r7
sub     r2, r7   ⇒ r8
```

(c) Naive Code

Geração de Código para Referências a Estruturas

- Nos slides anteriores, vimos como IC pode ser gerado para expressões aritméticas simples e atribuições.
- Nos exemplos apresentados, todas as variáveis eram simples e identificadas pelo nome.
- A geração de código alvo requer a substituição desses nomes pela sua localização real: registradores, endereços absolutos de memória (para variáveis globais), ou *offsets* do registro de ativação (para variáveis locais).
- No entanto, há situações que exigem um cálculo para localizar o endereço em questão.
- Isso ocorre, por exemplo, com indexação de vetores e no acesso a um campo de uma estrutura.

Operações para Cálculo de Endereços

Modificações na notação de *three-address code*:

- As operações aritméticas usuais podem ser usadas para computar endereços.
- No entanto, é necessário introduzir novos modos de endereçamento:
  - *Address of*: denotado por `&`.
  - *Indirect*: denotado por `*`.
- Para armazenar a constante 2 no endereço da variável `x` mais 10 *bytes*, basta fazer:

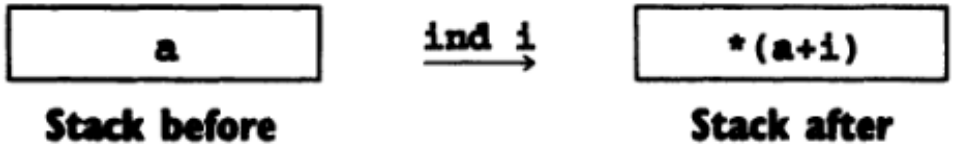
```
t1 = &x + 10
*t1 = 2
```

- A implementação desses novos modos de endereçamento requer uma modificação na quádrupla para inclusão de novos campos.

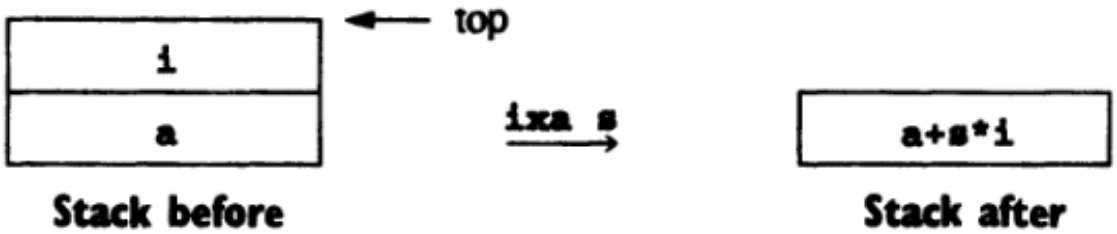
Operações para Cálculo de Endereços

Modificações na notação de *P-code*:

- Duas novas instruções são introduzidas para lidar com os modos de endereçamento.
- `ind` ("*indirect load*"):



- `ixa` ("*indexed address*"):



- Mesmo exemplo do slide anterior em *P-code*:

```
lda x       |     ldc 2
ldc 10      |     sto
ixa 1
```

Referências de Vetores

- Uma referência em um vetor requer indexar uma variável por uma expressão para se obter o endereço de um certo elemento do vetor.
- Como vetores são armazenados sequencialmente na memória, o endereço de cada posição deve ser calculado do endereço base e um *offset* que depende do índice.
- O endereço de um elemento de um vetor `a[t]` é calculado pela fórmula:

$$\text{base\_addr}(a) + (t - \text{lower\_bound}(a)) * \text{elem\_size}(a) \ .$$

- O acesso a uma posição do vetor exige a geração de código para o cálculo do endereço correspondente.
- Vetores multidimensionais requerem um cálculo similar para cada uma das dimensões.

Referências de Vetores – Exemplo

O comando em C

```
a[i+1] = a[j*2] + 3;
```

corresponde ao código de três-endereços abaixo.

```
t1 = j * 2
t2 = t1 * elem_size(a)
t3 = &a + t2
t4 = *t3
t5 = t4 + 3
t6 = i + 1
t7 = t6 * elem_size(a)
t8 = &a + t7
*t8 = t5
```

Referências de Vetores – Exemplo

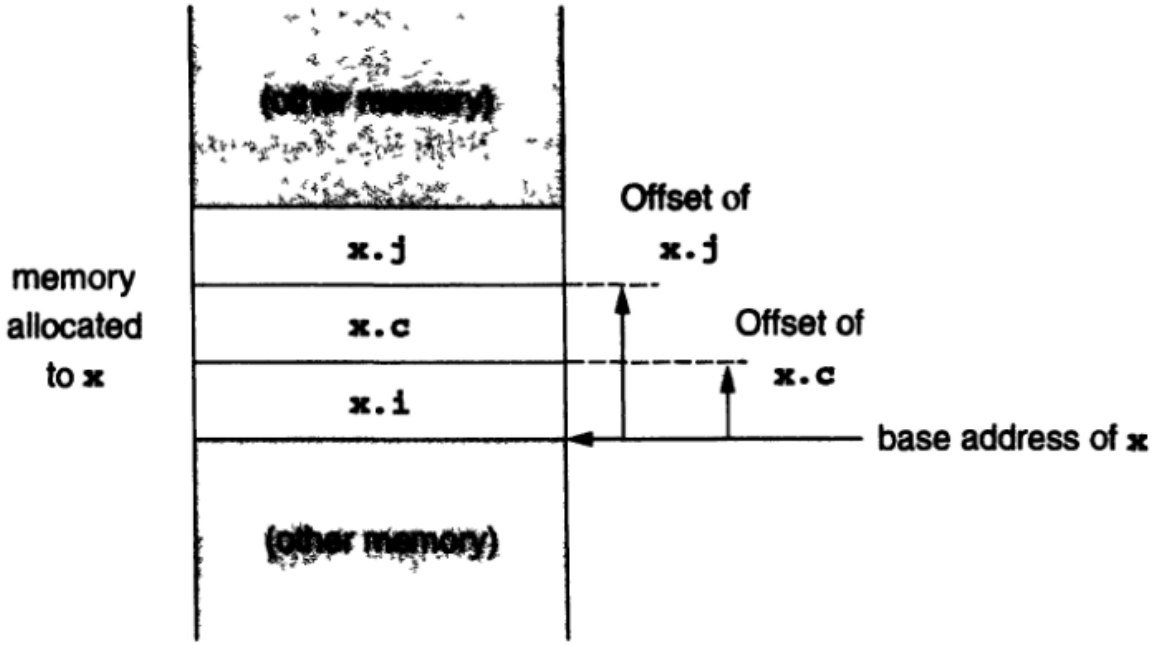
P-code para `a[i+1] = a[j*2] + 3;`

```
lda a
lod i
ldc 1
adi
ixa elem_size(a)
lda a
lod j
ldc 2
mpi
ixa elem_size(a)
ind 0
ldc 3
adi
sto
```

Referências a Campos de *structs*

- Computar o endereço de um campo de um *struct* é um problema similar ao acesso de uma posição de um vetor.
- No entanto, esse caso é mais simples pois o *offset* de cada campo é fixo.

```
typedef struct rec
   { int i;
     char c;
     int j;
   } Rec;
...
Rec x;
```



Referências a Campos de *structs* – Exemplo

O comando em C

```
x.j = x.i;
```

corresponde ao seguinte código de três-endereços

```
t1 = &x + field_offset(x, j)
t2 = &x + field_offset(x, i)
*t1 = *t2
```

e os seguintes comandos em *P-code*

```
lda x
lod field_offset(x, j)
ixa 1
lda x
ind field_offset(x, i)
sto
```

Geração de Código para Declarações de Controle

- Vamos agora descrever o processo de geração de código para declarações de controle.
- Dois comandos principais deste tipo: *if* e *while*.
- Geração de código intermediário para esse tipo de comando requer a geração de rótulos (*labels*).
- Rótulos representam endereços no código-alvo para onde são feitos *jumps*.
- Se for necessário eliminar os rótulos no código-alvo final, o compilador deve realizar um processo chamado de *back-patching*.
- Esse processo requer uma nova passada no IC para reescrever os comandos de salto com os endereços reais, que antes não eram conhecidos.

Geração de Código para Declarações de Controle

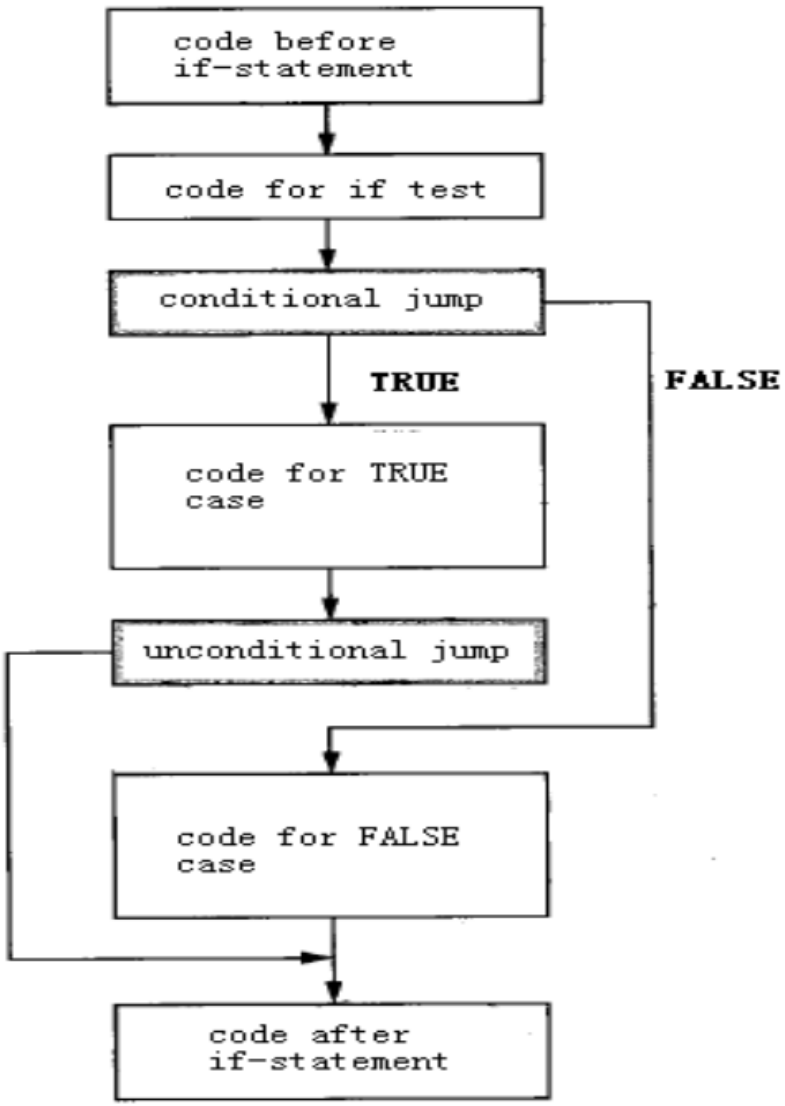
- Regras para os comandos *if* e *while* em linguagem C:

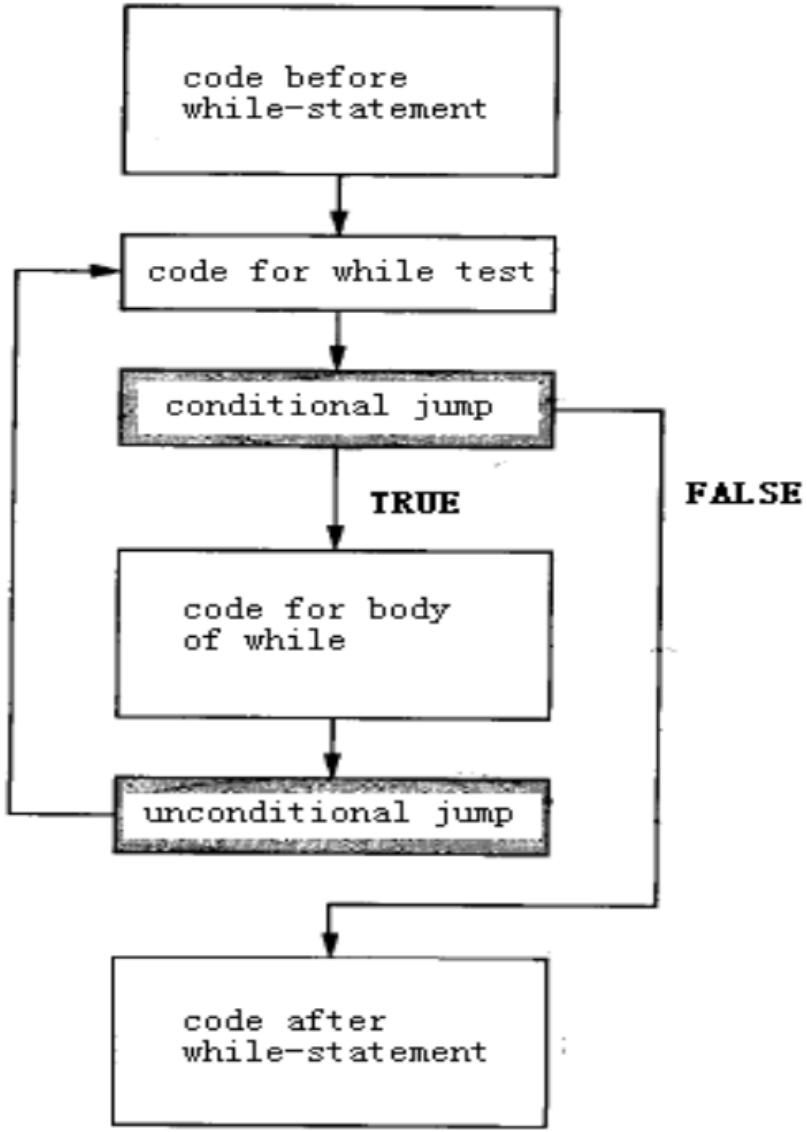
$$\textit{if-stmt} \rightarrow \texttt{if (}\ \textit{exp}\ \texttt{)}\ \textit{stmt} \mid \texttt{if (}\ \textit{exp}\ \texttt{)}\ \textit{stmt}\ \texttt{else}\ \textit{stmt}$$
$$\textit{while-stmt} \rightarrow \texttt{while (}\ \textit{exp}\ \texttt{)}\ \textit{stmt}$$

- O maior problema nesse processo é traduzir as características do código estruturado em uma forma equivalente "sem estrutura" (com saltos).
- Código com saltos pode ser implementado diretamente por comandos em *assembly* de qualquer arquitetura.
- Compiladores geram código desses comandos em uma forma padrão.

Geração de Código para Declarações de Controle

Arranjo típico de código para *if* e *while*.





Geração de Código para Declarações de Controle

Exemplo de código de três endereços para *if* e *while*.

if (*E*) *S1* **else** *S2*

```
<code to evaluate E to t1>
if_false t1 goto L1
<code for S1>
goto L2
label L1
<code for S2>
label L2
```

while (*E*) *S*

```
label L1
<code to evaluate E to t1>
if_false t1 goto L2
<code for S>
goto L1
label L2
```

Back-patching

- Problema dos arranjos anteriores: em alguns casos, saltos para um rótulo devem ser gerados antes da definição do próprio rótulo.
- Solução padrão: deixar um buraco no código aonde o salto deveria ocorrer.
- Outra possibilidade: gerar uma instrução de salto para uma localização *dummy*.
- Assim, quando a localização real do salto for conhecida, esse endereço é usado para consertar (*back-patch*), a instrução faltando.
- Complicações: algumas arquiteturas possuem dois tipos de saltos: curtos e longos, e o salto longo consome mais espaço de código.

Geração de Código para Chamadas de Funções

- Geração de código para chamadas de funções é totalmente dependente do ambiente de execução e da arquitetura alvo.
- Envolve dois conceitos fundamentais: definição e chamada de funções.
- Uma definição cria o nome da função, os seus parâmetros e código, mas a função não executa no momento da definição.
- Uma chamada cria os valores para os parâmetros e realiza um salto para o código da função, que executa e então retorna.

Geração de Código para Chamadas de Funções

- O IC de uma definição de função deve incluir uma instrução para marcar o início do código da função e outra instrução para marcar o ponto de retorno.
- De forma esquemática o IC fica como abaixo.

```
entry instruction
<code for the function body>
return instruction
```

- De forma similar, uma chamada de função deve ter uma instrução indicando o começo da computação dos argumentos e por fim a instrução de chamada de fato.
- De forma esquemática o IC fica como abaixo.

```
begin argument computation instruction
<code to compute arguments>
call instruction
```

Código de 3-Endereços para Chamadas de Funções

- Em código de três endereços, a instrução de entrada (`entry`) deve dar um nome para o ponto de entrada da função (como feito na instrução `label`).
- De forma similar, a instrução de retorno é indicada por `return`.
- Por exemplo, a função em C abaixo

```
int f(int x, int y)
{ return x + y + 1; }
```

pode ser traduzida para o seguinte IC de três endereços

```
entry f
t1 = x + y
t2 = t1 + 1
return t2
```

Código de 3-Endereços para Chamadas de Funções

Para uma chamada de função são necessárias três instruções novas no código de três endereços:

- `begin_args`: indica o início da computação dos argumentos. Leva à criação de um novo *frame pointer* em um ambiente de execução baseado em pilha.
- `arg`: indica que um dado nome (com seu respectivo endereço) é o argumento seguinte da função que será chamada.
- `call`: realiza o salto para a entrada da função.
- Por exemplo, a chamada `f(2+3,4)` pode ser traduzida para o seguinte IC de três endereços

```
begin_args
t1 = 2 + 3
arg t1
arg 4
call f
```

Conclusão

- Quase todos os exemplos de código destes *slides* são do livro do Louden. Utilizam um formato de código de três inventado pelo autor.
- Existem vários formatos de TAC diferentes, mas todos têm similaridades.
- O importante aqui é entender a ideia do formato e da sua geração.
- No Laboratório 07 vamos utilizar TAC para gerar código para uma arquitetura hipotética (*Tiny Machine – TM*).

Aula 07 – Geração de Código

Prof. Eduardo Zambon

Departamento de Informática (DI)
Centro Tecnológico (CT)
Universidade Federal do Espírito Santo (Ufes)

Compiladores
***Compiler Construction* (CC)**