CTC-41

Compiladores

Carlos Alberto Alonso Sanches

CTC-41

7) Geração de código

Ambientes de execução, código intermediário

Ambientes de execução

- De acordo com a linguagem compilada e a máquina alvo, pode haver diferentes ambientes de execução.
- Principais elementos que definem esses ambientes:
  - Organização da memória da máquina destino: registradores e RAM.
  - Forma como essa memória é utilizada na execução do código gerado: totalmente estática, totalmente dinâmica ou com o uso de uma pilha (é o mais comum).
- A memória RAM é dividida em área de código e área de dados:
  - Dados globais e estáticos (*extern static* em C) costumam receber endereços fixos antes da execução.
  - Constantes de valor pequeno são inseridas diretamente no código, sem necessitar de espaço de armazenamento; caso contrário, são alocados também em endereços fixos.
- Nos ambientes não estáticos:
  - As chamadas de funções são controladas com alocações dinâmicas.
  - Há uma área de *heap* para as alocações em tempo de execução.

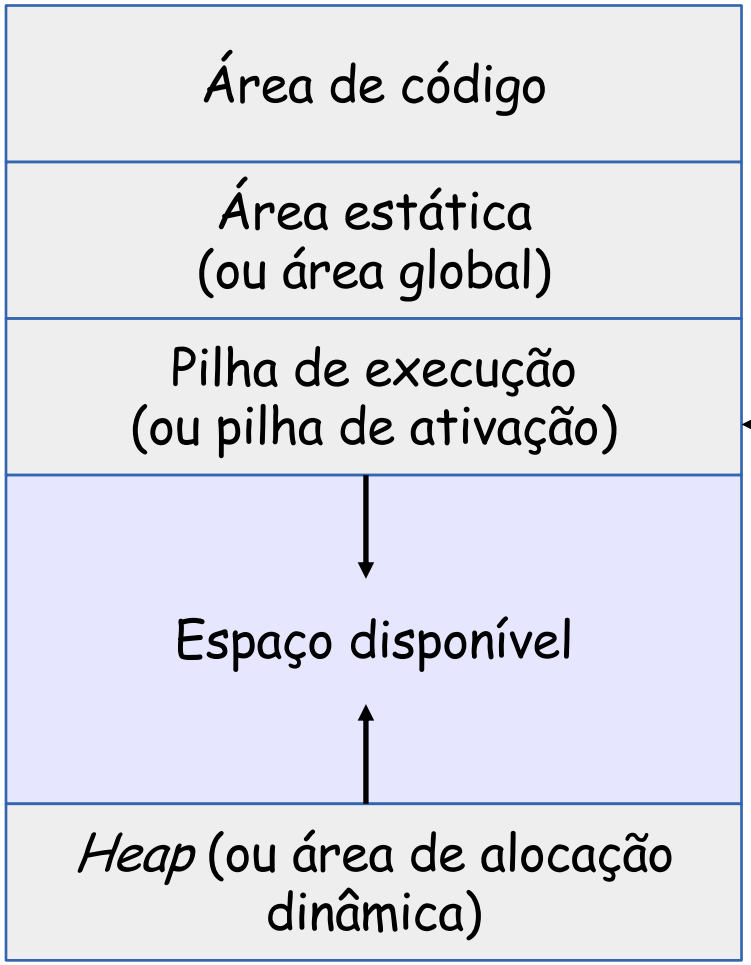
Ambientes pouco comuns

- Os ambientes totalmente estáticos são os mais simples: os dados permanecem em endereços fixos da memória durante toda o tempo de execução.
  - Isso somente é viável para linguagens sem ponteiros, sem alocação dinâmica e sem recursão. Exemplo: Fortran 77.
  - Cada função possui um quadro fixo: quando é chamada, seus argumentos são computados e armazenados nesses endereços.
- Há também ambientes totalmente dinâmicos, como é o caso das linguagens funcionais (LISP, por exemplo):
  - Toda a memória é organizada como um *heap*, com operações de alocação e liberação.
  - Um quadro de chamada de função somente é descartado quando todas as referências a ele tiverem desaparecido.
- Neste curso estudaremos apenas o ambiente com uso da pilha de execução.

Uso dos registradores

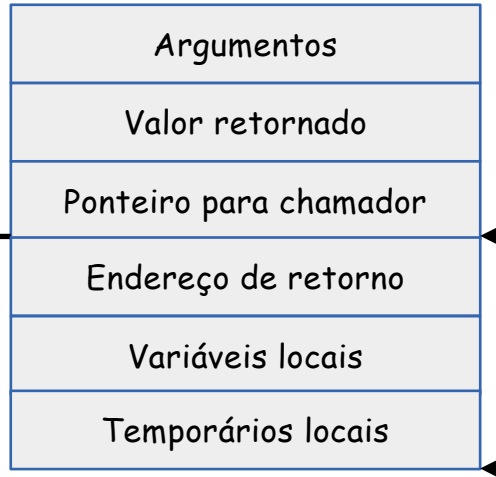
- Os registradores são dispositivos de armazenamento e acesso eficientes.
- Geralmente, há registradores de uso específico, onde os mais importantes são:
  - PC: contador de programa (endereço da próxima instrução a ser executada)
  - SP: ponteiro de pilha (endereço corrente do topo da pilha)
  - FP: ponteiro de quadro (endereço do quadro da função em execução)
- Se houver muitos registradores, toda a área estática e até mesmo de chamada de funções pode ser armazenada neles, tornando a execução muito rápida

Memória com pilha de execução



Alocado na pilha para cada chamada

Registro (ou quadro ou *frame*) de chamada (ou ativação) de função



Nas linguagens com aninhamento de funções (exemplo: Pascal), será preciso também um ponteiro para o registro do ambiente de definição (escopo onde a função foi declarada), que permitirá acesso a esses dados locais

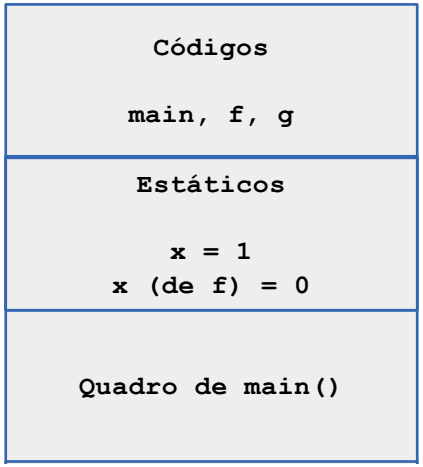
Ponteiro para chamador: também chamado de vínculo de controle

Ponteiro para ambiente de definição: também chamado de vínculo de acesso ou vínculo estático

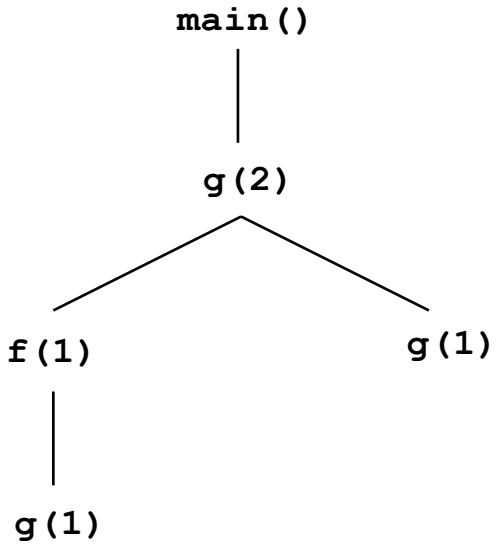
Exemplo com pilha

```
int x = 2;

void g(int);

void f(int n){
   static int x =
1;
   g(n);
   x--;
}

void g(int m){
   int y = m-1;
   if(y > 0){
      f(y);
      x--;
      g(y);
   }
}

main(){
   g(x);
   return 0;
}
```

Organização da memória



Árvore de ativações



Ativação e retorno de chamadas

- Considere uma linguagem sem aninhamento de funções.
- Ideia da sequência de ativação:
  - Computar argumentos e armazená-los na pilha
  - Reserve espaço na pilha para o valor a ser retornado
  - Empilhar FP (ponteiro para chamador)
  - Copiar SP em FP (novo registro de ativação)
  - Empilhar endereço de retorno e dados locais
  - Copiar em PC o início do código da nova função ativada
- Ideia da sequência de retorno:
  - Copiar FP em SP (desempilha quadro da última função)
  - Copiar em FP o ponteiro para chamador
  - Devolver valor retornado e desempilhá-lo (atualiza SP)
  - Desempilhar argumentos (atualiza SP)
  - Copiar em PC o endereço de retorno

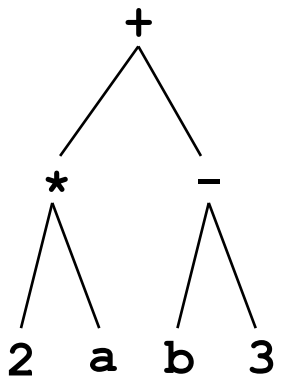
Código intermediário

- A geração do código intermediário é geralmente a primeira etapa da fase de síntese do compilador: consiste na transformação da árvore sintática em um código independente da máquina alvo.
- Vantagens:
  - Possibilita uma otimização prévia do código intermediário, sem levar em conta a especificação dos detalhes da máquina alvo;
  - Simplifica a implementação do compilador, pois resolve as dificuldades de uma forma mais gradativa;
  - Permite a tradução do código intermediário para diversas máquinas alvo.
- Desvantagem: exige um passo extra na compilação, tornando-a mais lenta.
- Há vários tipos de códigos intermediários. Um dos mais conhecidos é o código de três endereços, que permite especificar instruções com dois operandos e um resultado, cada um deles ocupando um endereço da memória.
  - Formato geral: x = y op z
- Iremos ver apenas alguns exemplos de geração de código de três endereços, sem abordar a sua posterior otimização.

Exemplo

- Considere a expressão matemática: `2 * a + (b - 3)`

Árvore sintática



Código de três endereços

```
t1 = 2 * a
t2 = b - 3
t3 = t1 + t2
```

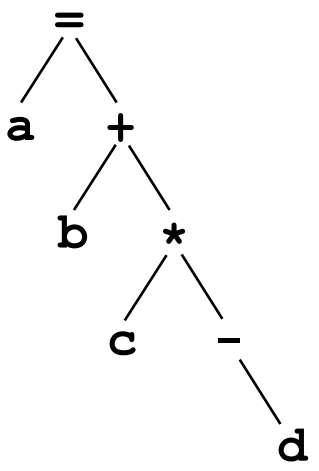
- Os temporários `t1`, `t2` e `t3` correspondem aos resultados associados aos nós internos da árvore.
- Podem ser atribuídos a registradores, ou mantidos na memória RAM.

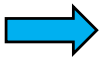
Código de três endereços

- Na implementação do código de três endereços, as instruções geralmente são manipuladas através de uma lista encadeada.
- Tipos básicos de instruções:
  - Atribuições com uso de expressões
  - Desvios, também utilizados em laços
  - Declaração e chamada de funções
  - Acessos indexados
- Veremos a seguir como gerar código intermediário para cada um desses tipos básicos.

Atribuições com uso de expressões

- Considerando o operador aritmético `op`, há três formas básicas:
  - `esq = dir`
  - `esq = dir`$_1$ `op dir`$_2$
  - `esq = op dir`
- Expressões mais complexas exigem múltiplas instruções com a criação de variáveis temporárias, cujos rótulos devem ser distintos dos identificadores utilizados no programa fonte.
- Exemplo: `a = b + c * - d`

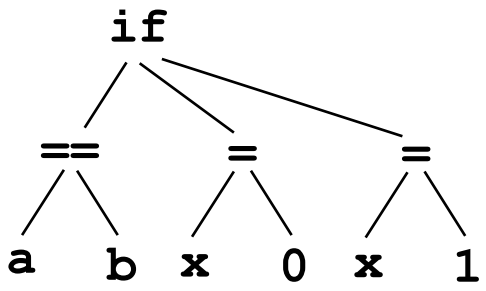


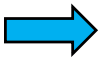


```
t1 = - d
t2 = c * t1
t1 = b + t2
a = t1
```

Desvios

- Seja `L` um rótulo que identifica uma linha de código, e `opr` um operador relacional.
- Há duas formas básicas de desvios:
  - Incondicional: `goto L`
  - Condicional: `if x opr y goto L`
- Exemplo: `if (a == b) x = 0; else x = 1`



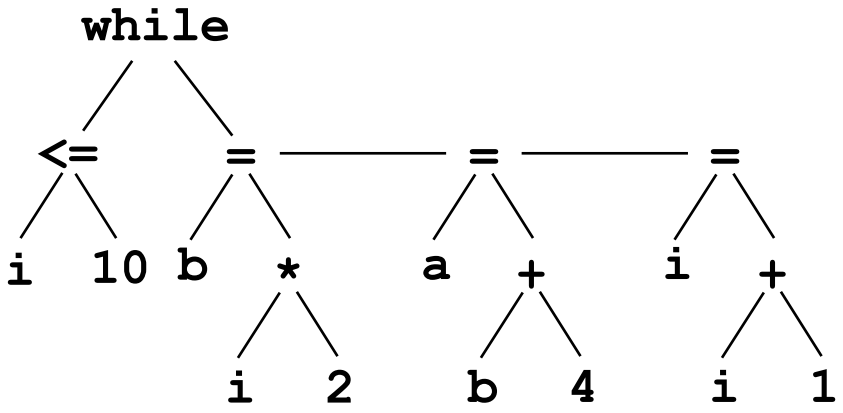


```
t1 = a == b
if_true t1 goto L1
x = 1
goto L2
L1: x = 0
L2:
```

Laços

- Como o uso de um desvio condicional, torna-se simples a implementação de laços.
- Considere o exemplo abaixo:

```
while (i <= 10) {
   b = i * 2;
   a = b + 4;
   i = i + 1;
}
```

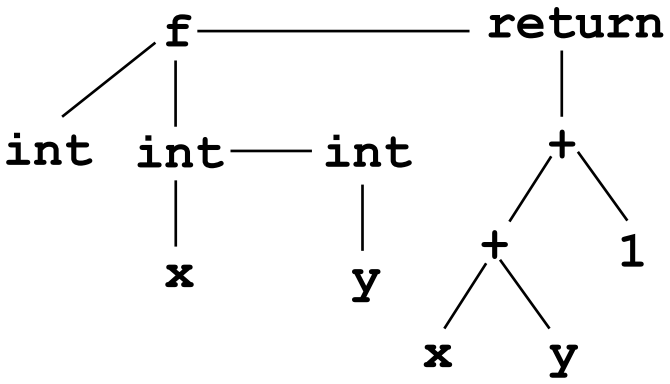


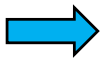

```
L1: t1 = i > 10
if_true t1 goto L2
b = i * 2
a = b + 4
i = i + 1
goto L1
L2:
```

Declaração de funções

- Na declaração de uma função, são definidos o nome, seus parâmetros, valor retornado (se houver) e o código a ser executado.
- Na árvore sintática, o nó de uma função aponta para seu tipo de retorno, sua lista de parâmetros e para o seu próprio código.
- Exemplo:

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





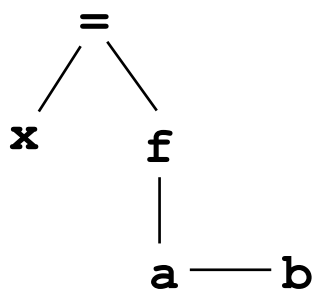
```
f: t1 = x + y
t2 = t1 + 1
return t2
```

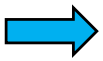
Chamada de funções

- A chamada de uma função f com n parâmetros de entrada tem o seguinte formato:

```
param x1
...
param xn
call f, n
```

- Exemplo: `x = f(a,b)`

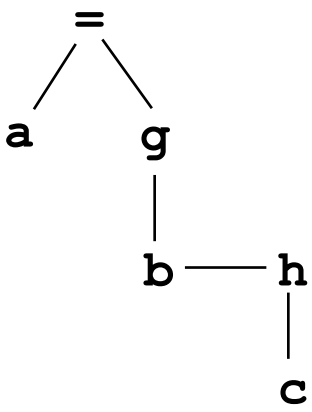


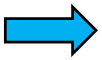


```
param a
param b
x = call f, 2
```

Outro exemplo de chamada

- Considere este novo exemplo: `a = g(b, h(c))`



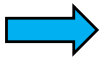


```
param b
param c
t1 = call h, 1
param t1
a = call g, 2
```

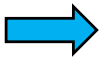
- Os parâmetros anteriores, uma vez consumidos, são desconsiderados na próxima chamada de função.

Acessos indexados

- Em uma variável indexada, o acesso a cada elemento é feito através de um endereço base e de um deslocamento proporcional ao valor do índice.
- Como exemplo, considere um vetor de inteiros indexado a partir da posição 0, e onde cada elemento ocupa 4 bytes:

- `y = x[i]` 

```
t1 = i * 4
y = x[t1]
```

- `a[i] = b[i] + c[i]` 

```
t1 = i * 4
t2 = b[t1]
t3 = c[t1]
t4 = t2 + t3
a[t1] = t4
```

Geração de código objeto

- Principais requisitos para os geradores de código objeto:
  - Código gerado correto e de boa qualidade;
  - Código gerado que faça uso efetivo dos recursos da máquina alvo;
  - Código gerado que execute eficientemente.
- Aspectos que costumam ser considerados:
  - Forma do código objeto gerado: linguagem absoluta (*load and go compilers*), relocável (permite compilação separada de subprogramas) ou *assembly* (uso de instruções simbólicas que permitem facilidades como macros, mas que exigem um passo posterior de tradução);
  - Seleção das instruções de máquina: escolher a sequência que permita um código mais curto e rápido;
  - Alocação dos registradores disponíveis: instruções envolvendo registradores são mais curtas e rápidas que as que fazem acesso à memória.
- Neste curso, faremos uma abordagem muito simplificada: utilizaremos uma máquina virtual e focaremos apenas na corretude do código objeto gerado.