UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
ALUNO: THIAGO THEIRY DE OLIVEIRA
DISCIPLINA: COMPILADORES
DOCENTE: SAMUEL XAVIER DE SOUZA

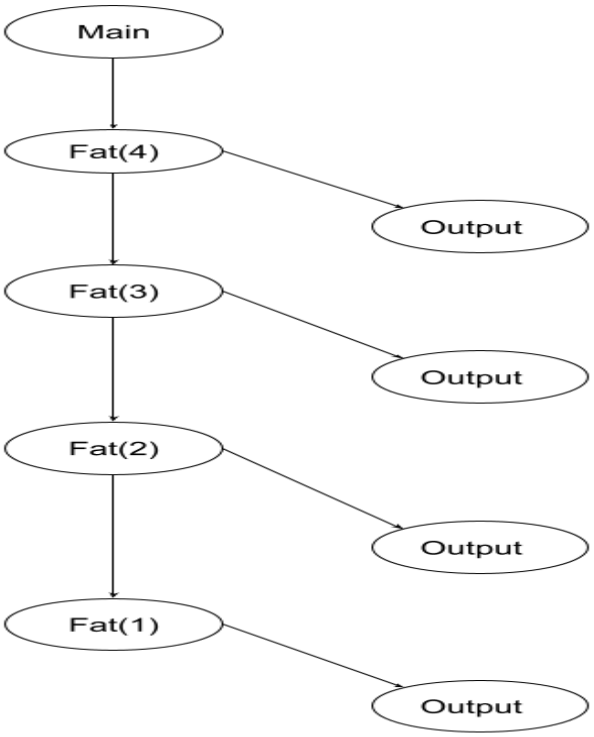
Lista de Compiladores

Questões respondidas: **cap 1**: 2,6,8 ; **cap 7:** 1, 3, 4, 7 ; **cap 11:** 6, 8, RQ 2 [pág. 610]

capítulo 6

2. Mostre a árvore de chamadas e o histórico de execução para o seguinte programa em C:

```
void Output(int n, int x) {
   printf("The value of %d! is %s.nn", n, x);
}
int Fat(int n) {
   int x;
     if (n > 1)
   x = n * Fat(n - 1);
   else
     x = 1;
   Output(n, x);
   return x;
}
void main() {
   Fat(4);
}
```

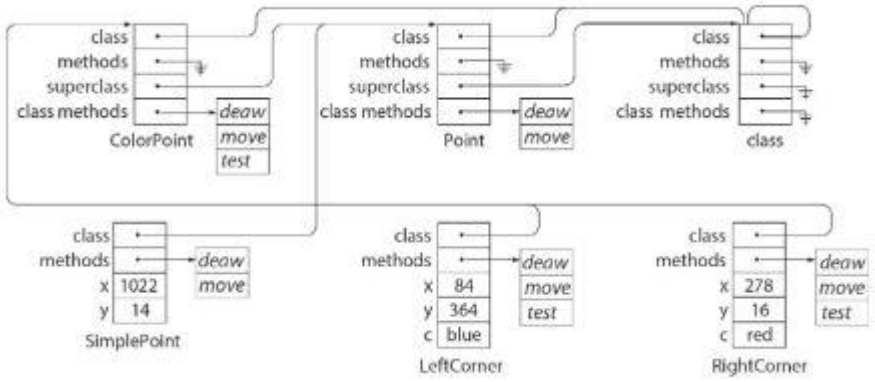


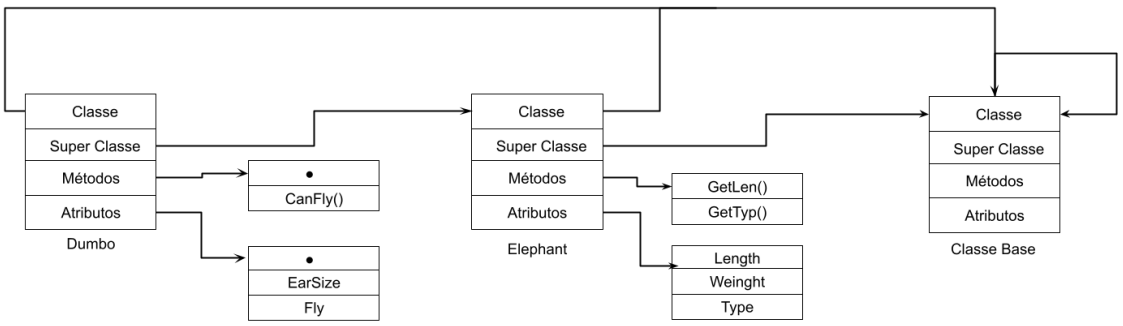
1. Main() **chama** Fat(4)
2. Fat(4) **chama** Fat(3)
3. Fat(3) **chama** Fat(2)
4. Fat(2) **chama** Fat(1)
5. Fat(1) **chama** Output(1,1)
6. Output(1,1) **retorna para** Fat(1)
7. Fat(1) **retorna para** Fat(2)
8. Fat(2) **chama** Output(2,2)
9. Output(2,2) **retorna para** Fat(2)
10. Fat(2) **retorna para** Fat(3)
11. Fat(3) **chama** Output(3,6)
12. Output(3,6) **retorna para** Fat(3)
13. Fat(3) **retorna para** Fat(4)
14. Fat(4) **chama** Output(4,24)
15. Output(4,24) **retorna para** Fat(4)
16. Fat(4) **retorna para** main()

6. Desenhe as estruturas que o compilador precisaria criar para dar suporte a um objeto do tipo Dumbo, definido da seguinte forma:

```
class Elephant {
   private int Length;
   private int Weight;
   static int type;

   public int GetLen();
   public int GetTyp();
}

class Dumbo extends Elephant {
   private int EarSize;
   private boolean Fly;

   public boolean CanFly();
}
```

exemplo da secção 6.3.4





8. Considere o programa escrito em pseudocódigo tipo Pascal mostrado na Figura 6.13. Simule sua execução sob as regras de vinculação de parâmetros de chamada por valor, chamada por referência, chamada por nome e chamada por valor-resultado. Mostre os resultados das instruções print em cada caso.

```
1  procedure main;
2     var a : array[1...3] of int;
3         i : int;
4     procedure p2(e : int);
5        begin
6         e := e + 3;
7         a[i] := 5;
8         i := 2;
9         e := e + 4;
10        end;
11    begin
12     a := [1, 10, 77];
13     i := 1;
14     p2(a[i]);
15     for i := 1 to 3 do
16       print(a[i]);
17     end.
```

Chamada por valor:

a:= [1, 10, 77]
i:= 1
P2(1) {
e:= 1 + 3;
a[1]:= 5;
i:= 2
e:= 4 + 4; }
print(a[i]) = 5 10 77

Chamada por referência:

```
a:= [1, 10, 77]
i:= 1
P2(&a[1]) {
e:= e + 3; => a[1]:= a[1] + 3
a[1]:= 5;
i:= 2
e:= e + 4; => a[1]:= a[1] + 4; }
print(a[i]) = 9 10 77
```

Chamada por nome:

```
a:= [1, 10, 77]
i:= 1
P2(a[i]) {
e:= e + 3; => a[i] + 3 => a[1] := a[1] + 3
a[i]:= 5 => a[1] := 5;
i:= 2
e:= e + 4; => a[i] + 4 => a[2] := a[2] + 4
print(a[i]) = 5 14 77
```

Chamada por valor-resultado:

```
a:= [1, 10, 77]
i:= 1
P2(a[i] = 1) {
e:= e + 3;
a[1]:= 5;
i:= 2
e:= e + 4;
a[1]:= e
}
print(a[i]) = 8 10 77
```

capítulo 7

1. O layout de memória afeta os endereços atribuídos a variáveis. Suponha que as variáveis de caractere não tenham restrição de alinhamento, as variáveis de inteiro pequeno devam ser alinhadas em limites de meia palavra (2 bytes), as variáveis inteiras alinhadas em limites de palavra (4 bytes) e as variáveis inteiras longas alinhadas em limites de palavra dupla (8 bytes). Considere o seguinte conjunto de declarações:

```
char a;
long int b;
int c;
short int d;
longint e;
charf;
```

Desenhe um mapa da memória para essas variáveis:

1 word = 4 bytes , char = 1 byte, short int = 2 bytes, int = 4 bytes, long int = 8 bytes

a. Supondo que o compilador não possa reordenar as variáveis.

a			b		c	d		e		f
1 byte	3 bytes livres	4 bytes livres	4 bytes	4 bytes	4 bytes	2 bytes	2 bytes livres	4 bytes	4 bytes	1 byte livres
1word		1 word	1 word	1 word	1 word	1 word		1 word	1 word	

b. Supondo que o compilador possa reordenar as variáveis para economizar espaço.

a	f	d	c	b		e	
1 byte	1 byte	2 bytes	4 bytes	4 bytes	4 bytes	4 bytes	4 bytes
1word			1 word	1 word	1 word	1 word	1 word

3. Para cada um dos tipos de variável a seguir, indique onde, na memória, o compilador poderia alocar o espaço para tal variável. As respostas possíveis incluem registradores, registros de ativação, áreas de dados estáticas (com diferentes visibilidades) e heap de runtime.

a. Uma variável local a um procedimento.

R = Como se trata de um procedimento e é uma variável local, o compilador pode alocá-lo em um registrador de ativação, que são feitos para armazenar informações separadas e fazer o controle de dados de procedimentos.

b. Uma variável global.

R= Por ser uma variável global ela é armazenada na área de dados estáticos, pois é um processo padrão que o compilador faz para variáveis com tempo de vida estáticos ou visibilidade global.

c. Uma variável global alocada dinamicamente.

R= Pelo fato de ser dinâmica, sendo assim, não sabemos o seu tamanho e o compilador não pode prever o tempo de vida, o compilador pode colocá-lo no heap de tempo de execução, já que o *Heap* contém blocos de memória alocadas dinamicamente, a pedido do processo, durante sua execução. Varia de tamanho durante a vida do processo.

d. Um parâmetro formal.

R= usados na declaração da função, por ser um parâmetro de um procedimento, logo pode ser armazenado em um registrador de ativação ou um registrador somente, assim o compilador pode ter controle dos contextos das variáveis.

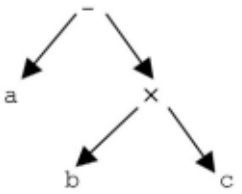
e. Uma variável temporária gerada pelo compilador.

R = Da mesma forma de uma variável local, uma variável temporária é única e tem tempo de vida curta, logo pode ser armazenada em um registrador ou registrador de ativação.

4. Use o algoritmo de geração de código de travessia em árvore, da Seção 7.3, para gerar um código simples para a árvore de expressão a seguir. Considere um conjunto ilimitado de registradores.

```
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;
}
```

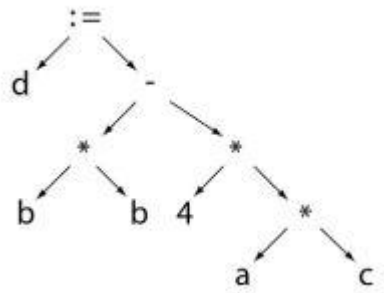
(a) Gerador de código de travessia em árvore



(b) Árvore sintática abstrata para a - b × 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) Código simples



load @b => r1
loadA0 r_{arp},r1 => r2
mult r2,r2 => r3
load 4 => r4
load @a => r5
loadA0 r_{arp},r5 => r6
load @c => r7
loadA0 r_{arp},r7 => r8
mult r6,r8 => r9
mult r4,r9 => r10
sub r3,r10 => r11
load @d => r12
store r11 => r12

7. Gere a ILOC predicada para a seguinte sequência de código. (Nenhum desvio deve aparecer na solução.)

Código-fonte	if (x < y) then a ← c + d else a ← e + f

```
if (x < y)
   then z = x * 5;
   else z = y * 5;
w = z + 10;
```

```
        cmp_LT   r_x, r_y   ⇒ r_1
        not      r_1        ⇒ r_2
(r_1)?  add      r_c, r_d   ⇒ r_a
(r_2)?  add      r_e, r_f   ⇒ r_a
```

cmpLT r_x,r_y => r1
not r1 => r2
(R1)? loadI 5 => r3
(R1)? mult r3, r_x => rz
(R2)? loadI 5 => r3
(R2)? mult r3, r_y => rz
loadI 10 => r4
add r4 +rz => rw

capítulo 11

6. Um otimizador peephole real precisa lidar com operações de fluxo de controle, incluindo desvios condicionais, saltos e instruções rotuladas.

a. O que este otimizador deve fazer quando traz um desvio condicional para a janela de otimização?

R= A presença de operações de fluxo de controle complica a simplificação. O modo mais fácil de tratar delas é limpar a janela do simplificador quando esta alcançar um desvio, um salto ou uma instrução rotulada. Isto impede que o simplificador mova efeitos para caminhos onde não estavam presentes.

b. A situação é diferente quando ele encontra um salto?

R= Não, o modo mais fácil de tratar delas é limpar a janela do simplificador

c. O que acontece com uma operação rotulada?

R= Não, o modo mais fácil de tratar delas é limpar a janela do simplificador

d. O que o otimizador pode fazer para melhorar esta situação?

R= O simplificador pode alcançar resultados melhores examinando o contexto ao redor de desvios, mas isto introduz vários casos especiais ao processo. processo, o que pode levar a várias formas de resolver dependendo desses casos especiais, como rastrear e eliminar os rótulos mortos, remova os rótulos, combina os blocos e simplificar a fronteira por meio da fronteira antiga, entre outros métodos.Mas de qualquer forma, o simplificador deve rastrear o número de usos para cada rótulo e eliminar os que não podem mais ser referenciados

8. Transformadores peephole simplificam o código enquanto selecionam uma implementação concreta para ele. Suponha que este transformador seja executado antes do escalonamento de instruções ou alocação de registradores, e que ele pode usar um conjunto ilimitado de nomes de registrador virtual.

a. O transformador peephole pode alterar a demanda por registradores?

R= Sim, porque é possível que a simplificação diminua a demanda dos registradores. uma das técnicas mais comuns aplicadas na otimização peephole é a sequências nulas em que

pode excluir operações inúteis, como também a de combinar operações em que substitui várias operações por um equivalente e as operações do modo de endereço em que usa os modos de endereço para simplificar o código. Ou seja, essas técnicas resultam em otimizar o código para que utilize menos memória e seja mais performático, fazendo com que tenha uma demanda menor de uso de registradores.

b. Ele pode alterar o conjunto de oportunidades que estão disponíveis ao escalonador para reordenar o código?

R= sim, pois se como foi citado anteriormente em que a otimização peephole envolve a mudança de um pequeno conjunto de instruções para um conjunto equivalente que tenha melhor desempenho. consequentemente pode alterar o conjunto de oportunidades que estão disponíveis ao escalonador, já que é utilizado algoritmos de escalonamento que estabelecem a lógica de tal decisão para decidir o momento em que cada processo obterá a CPU

RQ2. Muitos compiladores usam IRs com um nível de abstração mais alto nos primeiros estágios da compilação e depois passam para uma IR mais detalhada no back end. Que considerações poderiam ser argumentadas contra a exposição dos detalhes de baixo nível nos primeiros estágios da compilação?

R= Atualmente, há uma grande variedade de diferentes IRs em uso, muitos compiladores expandem sua IR para uma forma detalhada de baixo nível antes de selecionar instruções. Essas IRs detalhadas podem ser estruturais, como em nossa AST a exposição de mais detalhes em uma AST, por exemplo, deve levar a um código melhor, assim como o aumento no número de operações da máquina-alvo que o gerador de código considera. Juntos, porém, esses fatores criam uma situação em que o gerador de código pode descobrir muitas maneiras diferentes de implementar determinada subárvore, quando o gerador de código considera várias combinações possíveis para determinada subárvore, precisa de um modo para escolher entre elas. Se o construtor de compiladores puder associar um custo a cada padrão, então o esquema de casamento pode selecionar padrões de um modo que minimize os custos. Se os custos realmente refletirem o desempenho, este tipo de seleção de instruções controlada por custo deve levar a um bom código. Porém além desta técnica ser inerentemente dispendiosa, o mecanismo usado para testar a equivalência tem forte impacto sobre o tempo exigido para testar cada sequência candidata.