– INF01147 –
Compiladores

Otimização de Código

Prof. Lucas M. Schnorr
– Universidade Federal do Rio Grande do Sul –

Plano da Aula de Hoje

- Exercício de revisão da aula anterior

- Otimização – uma introdução
- Otimização de Janela (*peephole*)
- Grafo de Fluxo
- Otimização Local

Exercício de revisão

- Uso de registradores é subdividido
  - Alocação de Registradores (quais variáveis em registradores)
  - Atribuição (associar um registrador a uma variável)
- Quantos registradores são necessários?

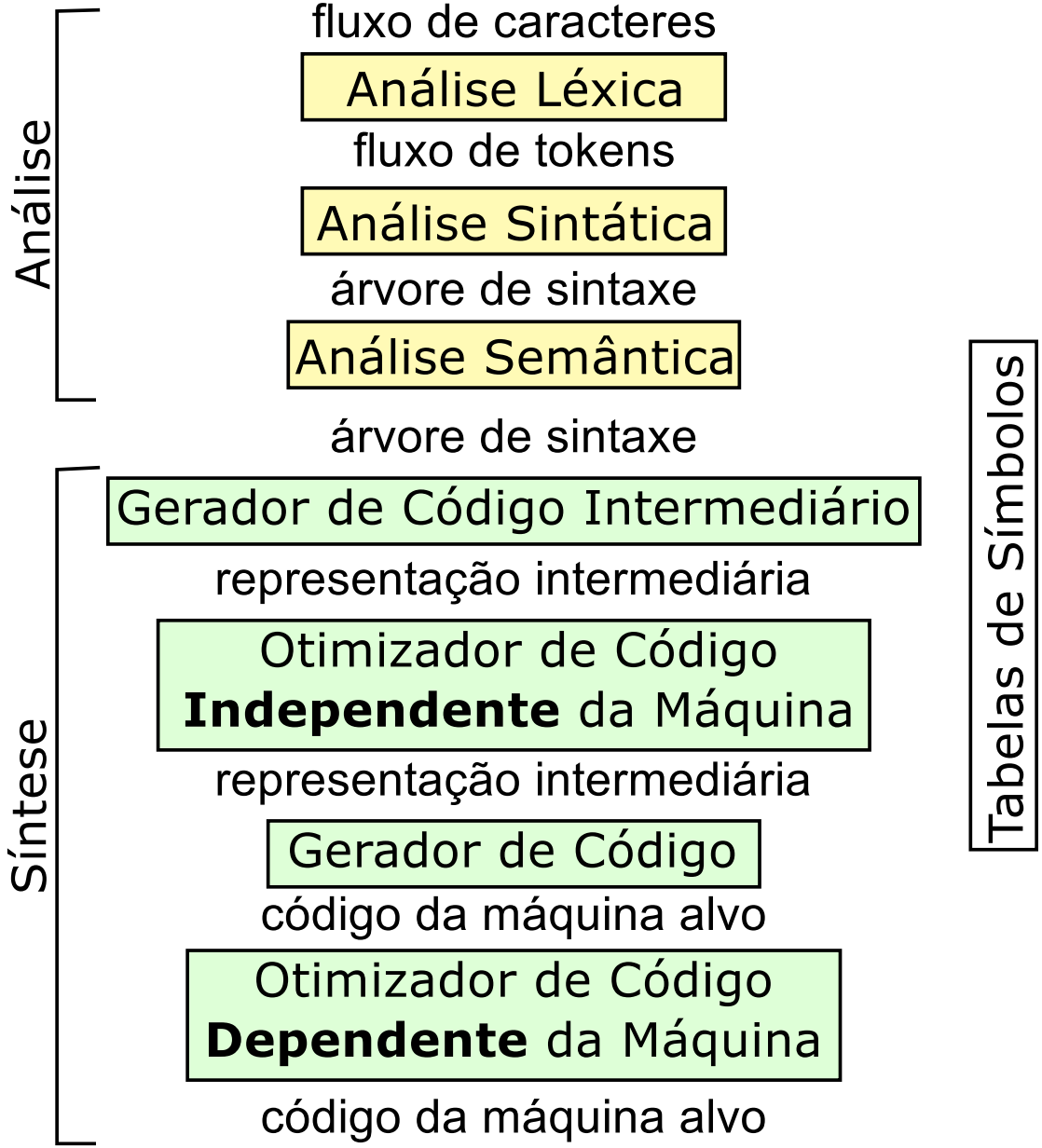
```
a:  s1 = ld(x)
b:  s2 = s1 + 0
c:  s3 = s1 * 8
d:  s4 = s1 - s1
e:  s5 = s1 / s3
f:  s6 = s2 * 1
g:  s7 = s4 - s5
h:  s8 = s6 * s7
i:  s9 = s2 * s1
j:  s10 = s4 * s9
k:  st(s10)
```

Otimização

Otimização de Código – Introdução

- Aspectos contraditórios
  - Uso de memória
  - Velocidade de execução
- Código ótimo
  - Problema não decidível $\rightarrow$ Utilizar heurísticas
  - Devemos pensar em melhorias ao invés de otimização

Estrutura de um Compilador em fases



Otimização de Código

- Otimização do código intermediário
  - Eliminar atribuições redundantes
  - Suprimir subexpressões comuns
  - Eliminar temporários desnecessários
  - Trocar instruções de lugar
- Otimização de código objeto
  - Troca de instruções de máquina por instruções mais rápidas
  - Melhor utilização de registradores

Otimização de Código

- Como melhorar?

```
int x[SIZE];
int main () {
  int i;
  for (i = 0; i < SIZE; i++) x[i] = 1;
}
```

- Possível alteração no código de usuário

```
int x[SIZE];
int main () {
  register int *p;
  for (p = (int*)x; p < (int*)x+SIZE;) *p++ = 1;
}
```

Otimização de Janela

peephole optimization

Otimização de Janela – Introdução

- Maioria dos compiladores
  - Cuidadosa seleção, alocação e atribuição de registradores
- Uma estratégia alternativa
  - Gerar código simples, ingênuo
  - Aplicar transformações ao código gerado

- Otimização de Janela
  - Observa-se poucas instruções por vez
  - Substituindo a sequência por uma menor ou mais rápida
- Características
  - Cada melhoria gera novas oportunidades
  - Várias passagens pelo código (janelas de tamanho diferentes)

Otimização de Janela

- Eliminação de instrução redundante
- Otimizações de fluxo de controle
- Simplificações algébricas
- Uso de idiomas de máquina
- Avaliação de constantes
- Propagação de cópias

Otimização de Janela – Eliminação de Instrução Redundante

- Dada esta sequência

```
LD   a    R0
ST   R0   a
```

- Podemos remover a instrução de armazenamento

- O que aconteceria se houvesse um rótulo na instrução ST?

Otimização de Janela – Eliminando código inacessível

- Remoção de instruções inacessíveis (código morto)

```
      ADD   r1  r2  r3
      JUMP  r1
      SUB   r2  r3  r5
      DIV   r2  r3  r5
R:    MULT  r3  r2  r1
```

- Remover instruções não-rotuladas
  - Imediatamente depois de desvios incondicionais
  - Repetir para remover uma sequência de instruções

Otimização de Janela – Eliminando código inacessível

- Caso da variável global `debug`

```
        if debug == 1 goto L1
        goto L2
L1:     imprime informação de depuração
L2:
```

- Eliminar um desvio
- Testar valor de debug na tabela de símbolos

Otimização de Janela – Eliminando código inacessível

- Instruções TAC definem um nome que não é utilizado
  - $a = bopc$ é morto se a não é utilizado
- Caso

```
b = 4 - 2
t1 = b / 2
t2 = a * t1
t3 = t2 * b
t4 = t3 + c
t5 = t3
t6 = t4
d = t4 * t4
```

Otimização de Janela – Otimizações de Fluxo de Controle

- Código intermediário frequentemente tem
  - Desvios para desvios
  - Desvios para desvios condicionais
  - Desvios condicionais para desvios
- Todos podem ser eliminados

- Qual transformação?

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

- Podemos remover a instrução rotulada L1?
  - Sim, se houver um desvio incondicional antes dela

Otimização de Janela – Otimizações de Fluxo de Controle

- Vamos supor
  - Existe apenas um desvio para L1
  - L1 é precedido por um desvio incondicional
- Então

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

- Pode ser transformada em

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

- Qual a diferença entre as duas versões?
  - Número de instruções é o mesmo
  - Às vezes saltamos um desvio na versão otimizada
    - Nunca na primeira versão

Otimização de Janela – Simplificação Algébrica

- Vários casos ingênuos

```
a:  s1 = ld(x)
b:  s2 = s1 + 0
c:  s3 = s1 * 8
d:  s4 = s1 - s1
e:  s5 = s1 / s3
f:  s6 = s2 * 1
g:  s7 = s4 - s5
h:  s8 = s6 * s7
i:  s9 = s2 * s1
j:  s10 = s4 * s9
k:  st(s10)
```

Otimização de Janela – Simplificação Algébrica

- Aplicação de fórmulas algébricas simples
  - Transformar em expressões equivalentes
    - Simplicidade/Desempenho

- Alguns exemplos

x + 0	0 + x	x
1 * x	x * 1	x
2 * x	x * 2	x + x
x**2	pow(x,2)	x * x
a * (1/b)	(1/b) * a	a/b
x > y		x-y >0

Otimização de Janela – Avaliação de Constantes

- Expressões aritméticas avaliadas em tempo de compilação

```
ADD   1    2   t3
MUL   t6   t3  t6

SET   1        t2
SUB   3    t2  t4
```

- Tarefas do compilador
  - Calcular o resultado
  - Emitir código usando o resultado calculado

Otimização de Janela – Uso de Idiomas de Máquina

- Saber quais as instruções que estão disponíveis
  - Escolher a mais apropriada (menor custo em tempo)
- Exemplo

$$x = x + 1$$

- Qual instrução usar se houver as seguintes possibilidades?

ADD
INC

Otimização de Janela – Propagação de cópias

- Procura-se por construções a = b
  - Trocar todas as ocorrências de a por b enquanto não houver mudança em nenhum dos dois
    - Exemplo

```
b = 4 - 2
t1 = b / 2
t2 = a * t1
t3 = t2 * b
t4 = t3 + c
t5 = t3
t6 = t5 + c
d = t4 * t6
```

Grafos de Fluxo

Considerando código TAC

Otimização de Código

- Principal dificuldade
  - Saber quando uma instrução irá manipular uma variável
  - Em tempo de execução
- Quando o fluxo avança sem desvios
  - Simples: calculamos a vida e morte de variáveis
  - Grafo de interferências $\rightarrow$ Pode nos levar a uma melhoria
- O que fazer quando o fluxo é desviado?

```
      x = 10
l1:   if (cond) goto l2
      y = x+1
      if (x >= 20) goto l1
      x = 10
      y = 0
      goto l1
l2:   print y
```

- Decomposição do TAC em blocos básicos

Blocos básicos

- Instrução líder pode ser
  - Primeira instrução de um programa
  - Instrução destino de uma operação de desvio
  - Instrução imediatamente após uma operação de desvio
- Bloco básico
  - Trecho de programa que inicia em uma instrução líder
  - Vai até o líder seguinte sem nenhum comando de desvio
- Grafo de fluxo
  - DAG que indica o fluxo de controle entre blocos básicos

Exemplo 1

```
1    begin
2       prod = 0
3       i = 1
4       do begin
5           prod = prod + a[i] * b[i]
6           i = i + 1
7       end
8       while i <= 20
9    end
```

Código TAC do Exemplo 1

```
(1)   prod = 0
(2)   i = 1
(3)   t1 = 4 * i
(4)   t2 = a[t1]
(5)   t3 = 4 * i
(6)   t4 = b[t3]
(7)   t5 = t2 * t4
(8)   t6 = prod + t5
(9)   prod = t6
(10)  t7 = i + 1
(11)  i = t7
(12)  if i <= 20 goto (3)
(13)  halt
```

- Quais são as instruções líderes?
- Quais são os blocos básicos?
- Qual é o grafo de fluxo de controle?

Código TAC do Exemplo 1

```
(1)   prod = 0
(2)   i = 1
(3)   t1 = 4 * i
(4)   t2 = a[t1]
(5)   t3 = 4 * i
(6)   t4 = b[t3]
(7)   t5 = t2 * t4
(8)   t6 = prod + t5
(9)   prod = t6
(10)  t7 = i + 1
(11)  i = t7
(12)  if i <= 20 goto (3)
(13)  halt
```

Exemplo 2 – quicksort

```
1   void quickSort (m, n) {
2     int i, j;
3     int v, x;
4     if (n <= m) return;
5     i = m-1; j = n; v = a[n];
6     while(1) {
7       do i=i+1; while (a[i] < v);
8       do j=j-1; while (a[j] > v);
9       if (i >= j) break;
10      x = a[i]; a[i] = a[j]; a[j] = x;
11    }
12    x = a[i]; a[i] = a[n]; a[n] = x;
13    quicksort(m,j);
14    quicksort(i+1, n);
15  }
```

- Vamos nos focar nas linhas 5 à 12

Exemplo 2 – quicksort TAC

```
 1  i = m - 1
 2  j = n
 3  t1 = 4 * n
 4  v = a[t1]
 5  i = i + 1
 6  t2 = 4 * i
 7  t3 = a[t2]
 8  if t3 < v goto (5)
 9  j = j - 1
10  t4 = 4 * j
11  t5 = a[t4]
12  if t5 > v goto (9)
13  if i >= j goto (23)
14  t6 = 4 * i
15  x = a[t6]
16  t7 = 4 * i
17  t8 = 4 * j
18  t9 = a[t8]
19  a[t7] = t9
20  t10 = 4 * j
21  a[t10] = x
22  goto (5)
23  t11 = 4 * i
24  x = a[t11]
25  t12 = 4 * i
26  t13 = 4 * n
27  t14 = a[t13]
28  a[t12] = t14
29  t15 = 4 * n
30  a[t15] = x
```

Exemplo 2 – quicksort TAC (líderes)

```
 1  i = m - 1
 2  j = n
 3  t1 = 4 * n
 4  v = a[t1]
 5  i = i + 1
 6  t2 = 4 * i
 7  t3 = a[t2]
 8  if t3 < v goto (5)
 9  j = j - 1
10  t4 = 4 * j
11  t5 = a[t4]
12  if t5 > v goto (9)
13  if i >= j goto (23)
14  t6 = 4 * i
15  x = a[t6]
16  t7 = 4 * i
17  t8 = 4 * j
18  t9 = a[t8]
19  a[t7] = t9
20  t10 = 4 * j
21  a[t10] = x
22  goto (5)
23  t11 = 4 * i
24  x = a[t11]
25  t12 = 4 * i
26  t13 = 4 * n
27  t14 = a[t13]
28  a[t12] = t14
29  t15 = 4 * n
30  a[t15] = x
```

Exercício – Calcule o grafo de fluxo com blocos básicos

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

Otimização de Código

Otimização de Código

- Melhor se considerarmos um grafo de blocos básicos
  - É a escolha da maioria dos compiladores

- Existem dois tipos de otimização
  - Local *versus* Global

- Local
  - Transformações que preservem a funcionalidade
    - Todas aquelas vistas em Otimização de Janela
      $\rightarrow$ mas limitadas as instruções do bloco
    - **Eliminação de sub-expressões comuns**
- Global
  - Movimentação de código
  - Variáveis de indução
  - Redução de força

Otimização – Eliminação de sub-expressões comuns

- Duas operações são comuns se produzem o mesmo resultado
  - Calcular uma única vez
  - Reutilizar nas próximas vezes, referenciando o resultado

- Expressão é
  - Viva
    - Se os operandos usados para calcular a expressão não mudaram
  - Morta
    - Caso contrário

- Local e Global

Otimização – Eliminação de sub-expressões comuns

- Código fonte

```
1    int main ()
2    {
3      int x, y, z;
4      ...
5      x = (1+20) * -x;
6      x = x*x+(x/y);
7      y = z = (x/y)/(x*x);
8      ...
9    }
```

- TAC

```
1    t1 = 1 + 20
2    t2 = -x
3    x = t1 * t2
4    t3 = x * x
5    t4 = x / y
6    y = t3 + t4
7    t5 = x / y
8    t6 = x * x
9    z = t5 / t6
10   y = z
```

Otimização – Eliminação de sub-expressões comuns

- TAC original

```
1     t1 = 1 + 20
2     t2 = -x
3     x = t1 * t2
4     t3 = x * x
5     t4 = x / y
6     y = t3 + t4
7     t5 = x / y
8     t6 = x * x
9     z = t5 / t6
10    y = z
```

- TAC melhorado

```
1     t2 = -x
2     x = 21 * t2
3     t3 = x * x
4     t4 = x / y
5     y = t3 + t4
6     z = t4 / t3
7     y = z
```

Exemplo considerando o pedaço do TAC quicksort

- TAC original

```
14  t6 = 4 * i
15  x = a[t6]
16  t7 = 4 * i
17  t8 = 4 * j
18  t9 = a[t8]
19  a[t7] = t9
20  t10 = 4 * j
21  a[t10] = x
22  goto (5)
```

- TAC melhorado

```
14  t6 = 4 * i
15  x = a[t6]
16  t8 = 4 * j
17  t9 = a[t8]
18  a[t6] = t9
19  a[t8] = x
20  goto B2
```

Otimização – Eliminação de sub-expressões comuns

- Como implementar a eliminação de sub-expressões comums?
  - Representar expressões com um DAG

- Folhas são identificadores e constantes
  - Valores inicias da computação efetuado no bloco
- Nós interiores são operadores aplicados às folhas e uma lista de identificadores
  - Cálculos efetuados no bloco básico
  - Nomes de variáveis que assumem os valores calculados

Exemplo

```
t1 = 4 * i
t2 = a[t1]
t3 = 4 * i
t4 = b[t3]
t5 = t2 * t4
t6 = prod + t5
prod = t6
t7 = i + 1
i = t7
if i <= 20 goto (3)
```

Uso do DAG (de expressões)

- Sub-expressões são detectadas automaticamente
  - Caso do $4 * i$
- Identificadores utilizados no bloco são os presentes nas folhas do DAG
  - Caso do *prod*, *a* e *i*
- Quando um nó n associado a um TAC do tipo 'x := ... op ...' ainda contém x em seu rótulo no final da construção, é que o TAC calcula alguma coisa que pode ser usado fora do BB.

Conclusão

- Leituras Recomendadas para a aula de hoje
  - Livro do Dragão
    - Seções 8.4, 8.7
  - Livro do Keith
    - Capítulo 8 (introdução)
  - Série Didática
    - Capítulo 6

- Próxima Aula
  - Otimização