

Construção de Compiladores 1 - 2015.1 - Prof. Daniel Lucrédio

Lista 09 - Geração de código e otimização

1. Cite quais são os 2 tipos de código intermediário apresentados em aula e suas características principais. Quais são as diferenças entre eles?

R. O **código de três endereços** é uma forma de representação intermediária na qual cada instrução pode envolver, no máximo, três endereços de memória:

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

em que op é um operador aritmético (+ ou -, por exemplo) ou algum outro operador que possa operar sobre os valores de y e z. Atenção: o uso de x é diferente do uso de y e z já que y e z podem representar constantes ou literais sem endereços na execução, enquanto é necessário conhecer o endereço de x para que a atribuição possa ser realizada. Isso fica bem claro com o P-código. Outra característica relevante do código de três endereços é a necessidade de utilizar temporários (t1, t2 etc.) para armazenar os valores intermediários das operações. Esses temporários correspondem aos nós interiores da árvore sintática e representam seus valores computados; podem ser armazenados na memória ou em registradores.

O **P-código**, por sua vez, surgiu como um código de montagem alvo padrão produzido pelos compiladores Pascal e foi projetado como código de uma máquina hipotética baseada em pilhas (P-máquina) com interpretador para diversas máquinas reais. Como foi projetado para ser executado diretamente, o P-código contém uma descrição implícita de um ambiente de execução, entre outros detalhes abstraídos quando o consideramos como código intermediário nesse curso.

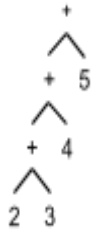
Comparação
Código de três endereços vs. P-código

Código de três endereços	P-código
<ul style="list-style-type: none">- é mais compacto (menos instruções)- é mais próximo do código de máquina- é autosuficiente no sentido de que não precisa de uma pilha para representar o processamento- precisa de temporários para armazenar os valores das computações intermediárias	<ul style="list-style-type: none">- é mais próximo do código de máquina- as instruções exigem menos endereços (os endereços omitidos estão na pilha implícita)- não precisa de temporários, uma vez que a pilha contém todos os valores temporários

2. Apresente a sequência de instruções de código de três endereços correspondente a cada uma das expressões aritméticas a seguir. Quais são as árvores sintáticas abstratas que correspondem à geração de código?

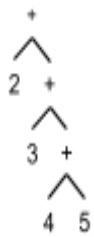
a) $2+3+4+5$

R.



$t1 = 2 + 3$
 $t2 = t1 + 4$
 $t3 = t2 + 5$

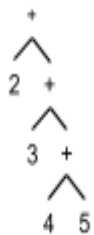
ou



$t1 = 4 + 5$
 $t2 = 3 + t1$
 $t3 = 2 + t2$

b) $2+(3+(4+5))$

R.



$t1 = 4 + 5$
 $t2 = 3 + t1$
 $t3 = 2 + t2$

c) $a*b+a*b*c$

R.



```

t1 = a * b
t2 = a * b
t3 = t2 * c
t4 = t1 + t3

```

ou



```

t1 = a * b
t2 = b * c
t3 = a * t2
t4 = t1 + t3

```

3. Apresente a sequência de instruções de P-código correspondente às expressões aritméticas do exercício anterior

R.
a) 2+3+4+5

```

ldc 2
ldc 3
adi
ldc 4
adi
ldc 5
adi

```

ou

```

ldc 4
ldc 5
adi
ldc 3
adi
ldc 2
adi

```

b) $2+(3+(4+5))$

```
ldc 2
ldc 3
ldc 4
ldc 5
adi
adi
adi
```

ou

```
ldc 4
ldc 5
adi
ldc 3
adi
ldc 2
adi
```

c) $a*b+a*b*c$

```
lod a
lod b
mpi
lod a
lod b
mpi
lod c
mpi
adi
```

4. Escreva a gramática de atributos para geração de código de três endereços para a gramática de expressões aritméticas de inteiros a seguir. Utilizando a gramática resultante, gere o código de três endereços para todas as expressões da questão 2.

```
exp → exp soma termo | termo
soma → '+' | '-'
termo → termo mult fator | fator
mult → '*'
fator → '(' exp ')' | NUM | ID
```

Regra gramatical	Regra semântica
$exp \rightarrow exp \text{ soma termo} \mid \text{termo}$	<pre>exp₁.nome = newtemp() exp₁.tacode = exp₂.tacode ++ termo.tacode ++ exp₁.nome "=" exp₂.nome soma.nome termo.nome</pre>

<code>exp → termo</code>	<code>exp.nome = termo.nome exp.tacode = termo.tacode</code>
<code>soma → '+'</code>	<code>soma.nome = "+" soma.tacode = ""</code>
<code>soma → '-'</code>	<code>soma.nome = "" soma.tacode = ""</code>
<code>termo → termo mult fator</code>	<code>termo₁.nome = newtemp() termo₁.tacode = termo₂.tacode ++ fator.tacode ++ termo₁.nome "=" termo₂.nome mult.nome fator.nome</code>
<code>termo → fator</code>	<code>termo.nome = fator.nome termo.tacode = fator.tacode</code>
<code>mult → '*'</code>	<code>mult.nome = "*" mult.tacode = ""</code>
<code>fator → '(' exp ')'</code>	<code>fator.nome = exp.nome fator.tacode = exp.tacode</code>
<code>fator → NUM</code>	<code>fator.nome = NUM.strval fator.tacode = ""</code>
<code>fator → ID</code>	<code>fator.nome = ID.strval fator.tacode = ""</code>

5. Considerando-se a mesma gramática do exercício anterior, escreva a gramática de atributos para a geração de P-código. Utilizando a gramática resultante, gere o P-código para todas as expressões da questão 2.

Regra gramatical	Regra semântica
<code>exp → exp soma termo termo</code>	<code>exp₁.pcode = exp₂.pcode ++ termo.pcode ++ soma.pcode</code>
<code>exp → termo</code>	<code>exp.pcode = termo.pcode</code>
<code>soma → '+'</code>	<code>soma.pcode = "adi"</code>
<code>soma → '-'</code>	<code>soma.pcode = "sbi"</code>
<code>termo → termo mult fator</code>	<code>termo₁.pcode = termo₂.pcode ++ fator.pcode ++ mult.pcode</code>
<code>termo → fator</code>	<code>termo.pcode = fator.pcode</code>
<code>mult → '*'</code>	<code>mult.pcode = "mpi"</code>

fator \rightarrow '(' exp ')'	fator.pcode = exp.pcode
fator \rightarrow NUM	fator.pcode = "ldc" NUM.strval
fator \rightarrow ID	fator.pcode = "lod" ID.strval

6. Apresente as instruções de três endereços correspondentes às expressões em C a seguir.

a) $(x=y=2)+3*(x=4)$

R.

```
y = 2
x = 2
x = 4
t1 = 3*4
t2 = 2 + t1
```

b) $a[a[i]]=b[i=2]$

R.

```
i = 2
t1 = 2*elem_size(b)
t2 = &b+t1
t3 = *t2
t4 = i
t5 = t4*elem_size(a)
t6 = &a+t5
t7 = *t6
t8 = t7*elem_size(a)
t9 = &a+t8
*t9 = t3
```

7. Apresente a sequência de instruções de P-código correspondente às expressões em C do exercício anterior

R.

a)

```
lda x
lda y
ldc 2
stn
stn
ldc 3
lda x
ldc 4
stn
mpi
adi
```

b)

```
lda a
lda a
lod i
ixa elem_size(a)
ind 0
```

```

ixa elem_size(a)
lda b
lda i
ldc 2
stn
ixa elem_size(b)
ind 0
sto

```

8. Cite as várias fontes de otimização apresentadas em aula explicando o que vem a ser cada uma delas

R.

- **Alocação de registradores:** O bom uso dos registradores é a característica mais importante do código eficiente, uma vez que é por meio deles que as operações são efetuadas. Quanto maior o número de registradores e melhor seu uso, maior a velocidade do código gerado;

- **Remoção de operações desnecessárias:** Às vezes um código-fonte dá origem (após a tradução para código intermediário ou alvo) a operações que não serão executadas ou que poderiam ser realizadas de uma maneira mais “inteligente” como as sub-expressões comuns, os códigos inatingíveis e os saltos desnecessários. Ao se evitar a geração de código para essas operações redundantes ou desnecessárias realiza-se uma otimização do tamanho do código gerado;

- **Adaptação de operações caras:** Algumas operações podem ser substituídas por uma versão mais “barata” computacionalmente (executa em tempo menor ou envolvendo menos registradores etc.) o que é chamado de redução de força. Além disso, essa otimização também diz respeito ao empacotamento de constantes (troca de expressões constantes pelo valor calculado), alinhamento de procedimentos (inserir o código do procedimento onde ele é chamado evitando-se, assim, todas as operações de ativação) e remoção de redução em cauda (quando a última instrução do procedimento é uma chamada a si próprio), e uso de dialetos de máquina. Todas essas alterações são realizadas com o intuito de aumentar a velocidade de execução do código-alvo; e

- **Previsão de comportamento do programa:** O conhecimento apropriado do comportamento dos programas a serem escritos na linguagem, por parte do projetista do compilador, permite que se possa otimizar as operações usadas com maior frequência aumentando, assim, a velocidade.

9. Quais os tipos de otimização (pequena escala, local, global ou interprocedimento) empregados nos exemplos a seguir:

a)

```
x = y * (-(-1))
```

```
x = y
```

R. pequena escala

b)

```
t1 = a + 2
```

```
t5 = a + 2
```

```
t2 = t1
```

```
t3 = b + c
```

```
t3 = b + c
```

```
t4 = a + 2
```

```
t5 = t4
```

R. local

c)

```

while(c<n*n-2+3.14) {
    c = c-1;
    cout << "laço "+c;
}

t1 = n*n-2+3.14;
while(c<t1);
    c = c-1;
    cout << "laço"+c;
}

```

R. global

d)

```

int soma(int a,int b) {
    return a+b;
}
...
x = 2;
y = 3;
c = soma(x,y);
cout << c;
...

...
x = 2;
y = 3;
t1 = x;
t2 = y;
t3 = t1+t2;
c = t3;
cout << c;
...

```

R. global

10. As otimizações a seguir são válidas? Justifique.

a)

```

while(c<n+1) {
    c = c-1;
    n = n+5;
    cout << "laço "+c;
}

t1 = n+1;
while(c<t1);
    c = c-1;
    n = n+5;
    cout << "laço"+c;
}

```

R. Não, pois o valor de n é alterado dentro do laço

b)

```

void funcao(int a) {
    if(a==0)
        print("Bum!");
    else {
        print("Faltam "+a+" secs");
        sleep(1000);
        funcao(a-1);
        print("Passaram-se "+a+" secs");
    }
}

void funcao(int a) {
    L:
    if(a==0)
        print("Bum!");
    else
        print("Faltam "+a+" secs");
        sleep(1000);
        a = a-1;
        goto L;
        print("Passaram-se "+a+"
secs");
}

```

R. Não, pois a chamada recursiva não é a última chamada do procedimento. O comando print("Passaram-se "+a+" secs") aparece depois da chamada recursiva.

c)

```

void f() {
    int i, j;
    for (i=0; i < 10; i++)
        cout << i << endl;
    for (j=10; j < 0; j--)
        cout << j << endl;
}

void f() {
    int i;
    for (i=0; i < 10; i++)
        cout << i << endl;
    for (i=10; i < 0; i--)
        cout << i << endl;
}

```



```
        cout << i << j << endl;                cout << i << i << endl;
    }                                             }
```

R. Não, pois há a necessidade de imprimir o valor de i (10, nesse exemplo) após o segundo laço, e portanto essa variável não pode ser reaproveitada.