Computação e Programação

Exame Tipo

Departamento de Matemática, IST

Dezembro de 2003

Grupo I

Desenvolva em MATLAB uma função *f* que quando recebe como argumento uma matriz quadrada de números inteiros, percorre a matriz dada elemento a elemento recorrendo a dois ciclos while encaixados e calcula o par [*x*,*y*] tal que *x* é o número de linhas que são iguais a alguma coluna e *y* é o índice da primeira dessas linhas. Sugestão: defina uma função auxiliar *leqc* tal que *leqc(m,i,j)* dá verdadeiro se a linha *i* da matriz *m* for igual à coluna *j*, e falso caso contrário, assumindo que a matriz *m* é quadrada.

Resolução:

```
function R=f(M)
[L,C]=size(M);
X=0;
Y=0;
I=1;
while I<=L
   J=1;
   E=0;
   while J<=C & E==0
      if leqc(M,I,J)
         E=1;
         X=X+1;
         if Y==0
            Y=I;
         end
      end
      J=J+1;
   end
   I=I+1;
end
R=[X,Y];

function R=leqc(M,I,J)
K=1;
R=1;
while K<=size(M,1) & R==1
   if M(I,K)~=M(K,J)
      R=0;
   end
   K=K+1;
end
```

Grupo II

a) Desenvolva em F um módulo que disponibiliza o tipo de dado *fila de espera com prioridades* (fecp). Por fecp entende-se uma fila de espera em que os elementos são guardados por ordem de chegada desde que tenham a mesma prioridade e em que cada elemento passa à frente dos de prioridade inferior. Por prioridade entende-se um inteiro positivo. A prioridade é tanto maior quanto menor for o inteiro. Escolhendo a seguinte implementação para o tipo fecp

```
type, public :: fecp
   private
   type(no), pointer :: prim
   integer :: comp
end type fecp

type, private :: no
   integer :: elem
   integer :: prio
   type(no), pointer :: seg
end type no
```

desenvolva as seguintes funções e subrotinas sobre fecp:

- `vazia()`: função que retorna a fecp vazia;
- `chegada(x,p,f)`: subrotina que implementa a chegada do elemento `x` com prioridade `p` à fecp `f`;
- `prim(f)`: função que devolve o primeiro elemento da fecp `f`;
- `prior(f)`: função que devolve a prioridade do elemento que está à frente da fecp `f`;
- `saida(f)`: subrotina que apaga o primeiro elemento da fecp `f`;
- `vaziaQ(f)`: função que devolve `.true.` se a fecp f estiver vazia e `.false.` caso contrário;
- `comp(f)`: função que devolve o comprimento da fecp `f`;

Resolução:

```
function vazia() result(f)
type(fecp) :: f

f%prim=>null()
f%comp=0
end function vazia


subroutine chegada(x,p,f)
integer, intent(in) :: x,p
type(fecp), intent(inout) :: f
type(no), pointer :: aux1, aux2

allocate(aux1)
aux1%elem=x
aux1%prio=p
aux1%seg=>null()
```

```
if (f%comp/=0) then
   if (f%prim%prio > p) then
      aux1%seg=>f%prim
      f%prim=>aux1
   else
      aux2=>f%prim
      do
         if (.not.associated(aux2%seg)) then
            aux2%seg=>aux1
            exit
         else
            if (aux2%seg%prio>p) then
               aux1%seg=>aux2%seg
               aux2%seg=>aux1
               exit
            end if
            aux2=>aux2%seg
         end if
      end do
   end if
else
   f%prim=>aux1
end if
f%comp=f%comp+1
end subroutine chegada


function prim(f) result(e)
type(fecp), intent(in) :: f
integer :: e

if (f%comp/=0) then
   e=f%prim%elem
end if
end function prim


function prior(f) result(p)
type(fecp), intent(in) :: f
integer :: p

if (f%comp/=0) then
   p=f%prim%prio
end if
end function prior


subroutine saida(f)
type(fecp), intent(inout) :: f
type(no), pointer :: aux

if (f%comp/=0) then
   aux=>f%prim
   f%prim=>f%prim%seg
   deallocate(aux)
   f%comp=f%comp-1
end if
end subroutine saida
```

```
function vaziaQ(f) result(b)
type(fecp), intent(in) :: f
logical :: b

if (f%comp/=0) then
   b=.false.
else
   b=.true.
end if
end function vaziaQ


function comp(f) result(c)
type(fecp), intent(in) :: f
integer :: c

c=f%comp
end function comp
```

b) Dadas duas fecps `f1` e `f2`, diz-se que `f1` é subfila `p` de `f2` sse os elementos de `f1` com prioridade `p` aparecem em `f2` pela mesma ordem e com a mesma prioridade, podendo `f2` ter mais elementos dessa prioridade se tiverem chegado depois. Desenvolva em F, recorrendo ao módulo anterior, uma subrotina `sub(f1,f2,p,r)` tal que `r` é `.true.` se `f1` for uma subfila `p` de `f2` e `.false.` caso contrário.

Resolução:

Começa-se por definir uma subrotina que apaga da fila todos os elementos com prioridade maior do que p, ou seja, os primeiros elementos da fila que não são da prioridade pretendida. Apresenta-se a solução recursiva desta subrotina.

```
recursive subroutine apagaprimeiros(f,p)
type(fecp), intent(inout) :: f
integer, intent(in) :: p

if (.not.vaziaQ(f)) then
   if (prior(f)<p) then
      call saida(f)
      call apagaprimeiros(f,p)
   end if
end if
end subroutine apagaprimeiros
```

Em seguida, define-se uma subrotina que compara os elementos da mesma prioridade de `f1` e de `f2`. Apresenta-se a solução recursiva desta subrotina.

```
recursive subroutine compara(f1,f2,p,r)
type(fecp), intent(inout) :: f1,f2
integer, intent(in) :: p
logical, intent(out) :: r

if (vaziaQ(f1)) then
   r=.true.
elseif (vaziaQ(f2)) then
   r=.false.
elseif (prior(f1)/=p) then
   r=.true.
elseif(prior(f2)/=p) then
   r=.false.
elseif(prim(f1)/=prim(f2)) then
   r=.false.
else
   call saida(f1)
   call saida(f2)
   call compara(f1,f2,p,r)
end if
end subroutine compara
```

A subrotina pretendida usa as duas anteriores, começando por apagar os elementos indesejados de f1 e de f2, e comparando as filas resultantes.

```
subroutine sub(f1,f2,p,r)
type(fecp), intent(inout) :: f1,f2
integer, intent(in) :: p
logical, intent(out) :: r

call apagaprimeiros(f1,p)
call apagaprimeiros(f2,p)
call compara(f1,f2,p,r)
end subroutine sub
```

Grupo III

a) Descreva sucintamente o método de programação em camadas e baseado em objectos. Indique as vantagens do método.

Resolução:

O método de programação por camadas baseadas em objectos que consiste em:

1. Identificar a camada dos tipos de objectos relevantes.
2. Desenvolver o programa abstracto supondo que a camada existe.
3. Implementar a camada de modo eficiente sobre os tipos primitivos da linguagem F.
4. Integrar o programa abstracto e a implementação da camada no programa operacional.

As vantagens deste método são inegáveis, incluindo:

- Separação entre os aspectos procedimentais e a estruturação dos objectos de trabalho o que nos permite concentrar em cada um dos subproblemas separadamente ou que sejam programadores diferentes encarregados do programa abstracto e da implementação da camada.
- Independência da implementação: o programa abstracto é independente dos detalhes de implementação da camada, o que permite ulteriormente alterar a implementação da camada (por exemplo, para a optimizar) sem ser necessário modificar o programa. Para tal, é importante que a linguagem de programação utilizada permita esconder os detalhes da implementação de uma camada. A construção `module` da linguagem F permite declarar como public / private os elementos da camada que pretendemos que sejam visíveis / invisíveis no exterior. Assim, o programador responsável pelo programa abstracto nem inadvertidamente poderá utilizar os detalhes irrelevantes da implementação.
- Reutilização de código: uma camada pode ser utilizada em muitos outros programas pois os objectos tendem a ser muito mais universais que as soluções procedimentais.

b) Considere a função *g* seguinte, definida em F. Descreva a funcionalidade desta função. Defina uma versão recursiva desta função, equivalente a *g*

```
function g(v) result(y)
integer, dimension(:), intent(in) :: v
integer :: y, i

i=1
y=0
do i=1,size(v)
   y=y+v(i)*i
end do
end function g
```

Resolução:

O programa recebe como argumento um vector de tamanho arbitrário, percorre-o elemento a elemento, recorrendo a um ciclo do e calcula a soma do produto de cada elemento pela respectiva posição.

A versão recursiva recorre a uma função auxiliar:

```
function gR(v) result(y)
integer, dimension(:), intent(in) :: v
integer :: y

y=gAux(V,1)
end function gR

recursive function gAux(v,i) result(y)
integer, dimension(:), intent(in) :: v
integer, intent(in) :: i
integer :: y

if (i>size(v)) then
   y=0
else
   y=v(i)*i+gAux(v,i+1)
end if
end function gAux
```