Instituto Superior Técnico

Lic. em Matemática Aplicada e Computação

Mestrado Integrado em Eng. Biomédica

Elementos de Programação

3 de Fevereiro de 2017 Exame 2 Duração: 2h30

Número: _____ **Nome:** _____

Grupo I (5 valores)

Neste exercício não pode usar definições por compreensão. As únicas operações sobre listas permitidas são: lista vazia (`[]`), acesso aos elementos da lista por posição (`lis[pos]`), seccionamento da lista (`lis[pos:pos]`), concatenação (`+`) e cálculo do comprimento (`len`). Pode usar `range`.

Defina imperativamente em *Python* uma função `quants` que dada uma lista `w` de números positivos e um número alvo `k` não negativo, devolva a lista de todas as listas de quantidades naturais $[q_0, \ldots, q_{len(w)-1}]$ tais que

$$\sum_{i=0}^{len(w)-1} (q_i * w[i]) == k.$$

Nomeadamente, `quants([1,2,5],6)` deverá ser `[[6, 0, 0], [0, 3, 0], [2, 2, 0], [4, 1, 0], [1, 0, 1]]`.

Resolução:

```
def pad(w,n):
    while len(w)<n:
        w=w+[0]
    return w

def quants(vals,quant):
    res=[]
    hips=[(0,[])]
    for x in vals:
        newhips=[]
        for (tot,h) in hips:
            n=0
            while tot+n*x<=quant:
                if tot+n*x==quant:
                    res=res+[pad(h+[n],len(vals))]
                else:
                    newhips=newhips+[(tot+n*x,h+[n])]
                n=n+1
        hips=newhips
    return res
```

Grupo II (4+4 valores)

Considere filas de espera com prioridades e tolerâncias, em que cada elemento entra na fila com uma dada prioridade e tolerância (inteiros positivos).

A prioridade permite que o elemento ultrapasse na fila todos os elementos com prioridade inferior. A tolerância decrementa sempre que sai um elemento da fila e faz com que o elemento abandone a fila ao chegar a 0. Identificaram-se as seguintes operações:

- `vazia`: fila vazia;
- `entra(f,x,p,t)`: fila que resulta da entrada na fila `f` do elemento `x` com prioridade `p` e tolerância `t`;
- `prox(f)`: elemento que está no início da fila `f`;
- `sai(f)`: fila que resulta da saída do próximo elemento de `f`;
- `compr(f)`: número de elementos de `f`;
- `fptbfQ(e)`: `True` se e só se `e` é uma fila com prioridades e tolerâncias bem formada.

Note, por exemplo, que deve ser `c` o resultado de

```
prox(sai(entra(entra(entra(vazia(),c,1,10),b,4,1),a,5,3))).
```

a) Desenvolva em *Python* uma implementação eficiente deste tipo de dados, de modo a que cada fila com prioridades e tolerâncias seja representada por uma lista da forma $[(x_1, p_1, t_1), ..., (x_n, p_n, t_n)]$ onde os elementos surgem por ordem de saída, isto é, cada x_i é o i-ésimo elemento que sairá da fila (se a sua tolerância t_i o permitir).

Resolução:

In [1]:
```python
def vazia():
    return []

def entra(f,x,p,t):
    f.insert(pos(f,p),(x,p,t))
    return f

def pos(f,p):
    pos=0
    while pos<len(f) and f[pos][1]>=p:
        pos=pos+1
    return pos

def prox(f):
    assert f!=[]
    return f[0][0]

def sai(f):
    return [(x,p,t-1) for (x,p,t) in f[1:] if t>1]

def compr(f):
    return len(f)

def fptbfQ(e):
    def intpos(a):
        return (type(a) is int) and a>0
    def triplo(x):
        return (type(x) is tuple) and len(x)==3 and intpos(x[1]) and intpos(x[2])

    return (type(e) is list) and all([triplo(x) for x in e])
```

b) Desenvolva em *Python*, sobre a camada de abstracção acima desenvolvida e assegurando a independência da implementação, uma função `maxprio` que recebendo uma fila de espera com prioridades e tolerâncias não vazia `f` calcula a prioridade máxima de algum elemento na fila `f` (pode assumir que os elementos na fila são `1,2,3,...,n` onde `n` é o seu comprimento).

Resolução:

```
In [1]: def maxprio(f):
            n=compr(f)
            assert n!=0
            p=1
            cont=True
            while cont:
                fnova=entra(f,n+1,p+1,1)
                if prox(fnova)==n+1:
                    cont=False
                else:
                    p=p+1
            return p
```

Grupo III (4 valores)

Neste exercício não pode usar recursão, ciclos ou atribuições. Pode usar, sem necessitar de os definir, os combinadores `map`, `reduce`, `any`, `all`, `filter`, `nest`, `fixedpoint`, bem como definições `lambda` e por compreensão.

Implemente funcionalmente em *Python* o algoritmo de ordenação *mergesort*.

Resolução:

```
In [1]: def mergesort(w):
            def merge(duas):
                def maux(tres):
                    if len(tres[1])==0 and len(tres[2])==0:
                        return tres
                    elif len(tres[1])==0 or len(tres[2])==0:
                        return [tres[0]+tres[1]+tres[2],[],[]]
                    elif tres[1][0]<=tres[2][0]:
                        return [tres[0]+[tres[1][0]],tres[1][1:],tres[2]]
                    else:
                        return [tres[0]+[tres[2][0]],tres[1],tres[2][1:]]

                return fixedpoint(maux,[[]]+duas)[0]

            def step(lista):
                if len(lista)<2:
                    return lista
                else:
                    return [merge(lista[2*i:2*i+2]) for i in range(len(lista)//2)]+([] if len(lista)%2==0 else [lista[-1]])

            if len(w)==0:
                return []
            else:
                return fixedpoint(step,[[x] for x in w])[0]
```

Grupo IV (3 valores)

Considere o seguinte programa imperativo `PROG`.

```
r=0
y=0
while x!=y:
    r=r+2*y+1
    y=y+1
```

Demonstre que é válida a asserção

$$\{\texttt{True}\}\ \texttt{PROG}\ \{\texttt{r} == \texttt{x}^{**}2\}.$$

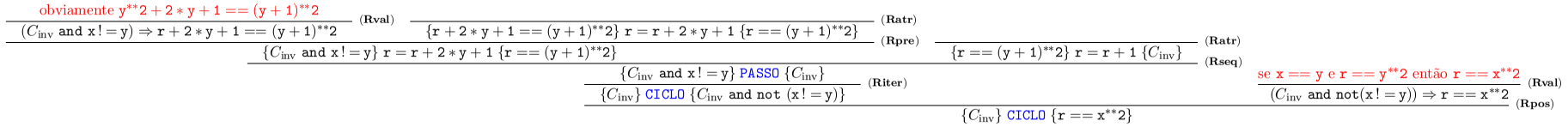
Resolução: Considera-se a estrutura de ciclo inicializado

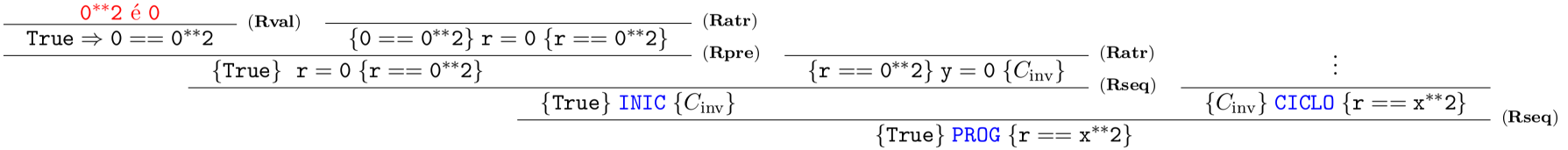
$$\texttt{PROG}\left\{\begin{array}{ll} \left.\begin{array}{l}\texttt{r} = 0\\ \texttt{y} = 0\end{array}\right\}\texttt{INIC} & \\ \texttt{while x} \,!= \texttt{y}: & \\ \quad \left.\begin{array}{l}\texttt{r} = \texttt{r} + 2 * \texttt{y} + 1\\ \texttt{y} = \texttt{y} + 1\end{array}\right\}\texttt{PASSO} & \left.\begin{array}{l}\\ \\ \\ \end{array}\right\}\texttt{CICLO}\end{array}\right.$$

e a condição invariante do ciclo

$$C_{\text{inv}} \equiv (\texttt{r} == \texttt{y}^{**}2).$$

A demonstração da asserção segue abaixo, onde surgem a vermelho justificações para a validade das condições Booleanas.


obviamente y**2 + 2 * y + 1 == (y + 1)**2
(Cinv and x ! = y) ⇒ r + 2 * y + 1 == (y + 1)**2
(Rval)
{r + 2 * y + 1 == (y + 1)**2} r = r + 2 * y + 1 {r == (y + 1)**2}
(Ratr)
{Cinv and x ! = y} r = r + 2 * y + 1 {r == (y + 1)**2}
(Rpre)
{r == (y + 1)**2} r = r + 1 {Cinv}
(Ratr)
{Cinv and x ! = y} PASSO {Cinv}
{Cinv} CICLO {Cinv and not (x ! = y)}
(Riter)
{Cinv} CICLO {r == x**2}
(Rseq)
se x == y e r == y**2 então r == x**2
(Cinv and not(x ! = y)) ⇒ r == x**2
(Rval)
(Rpos)


0**2 é 0
True ⇒ 0 == 0**2
(Rval)
{0 == 0**2} r = 0 {r == 0**2}
(Ratr)
{True} r = 0 {r == 0**2}
(Rpre)
{r == 0**2} y = 0 {Cinv}
(Ratr)
{True} INIC {Cinv}
(Rseq)
⋮
{Cinv} CICLO {r == x**2}
{True} PROG {r == x**2}
(Rseq)