Instituto Superior Técnico

Lic. em Matemática Aplicada e Computação
Mestrado Integrado em Eng. Biomédica

Elementos de Programação

20 de Janeiro de 2016 Exame 1 Duração: 2h30

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 `particaoQ` que dada uma lista de números `w` devolva `True` se existe $I \subseteq \{0, \ldots, \texttt{len(w)} - 1\}$ tal que $\sum_{\texttt{i} \in I} \texttt{w[i]}$ é igual a $\sum_{\texttt{i} \notin I} \texttt{w[i]}$, e `False` caso contrário.

Nomeadamente, `particaoQ([1,2,3,3,1])` deverá ser `True` (pois `2+3` é igual a `1+3+1`), mas `particaoQ([1,2,4])` deverá ser `False`.

Resolução:

```
def tiraum(u):
    res=[]
    for i in range(len(u)):
        res=res+[u[:i]+u[i+1:]]
    return res

def somat(u):
    res=0
    for x in u:
        res=res+x
    return res

def particaoQ(w):
    alvo=somat(w)/2
    ok=False
    hips=[w]
    while not(ok) and hips!=[]:
        u=hips[0]
        if somat(u)==alvo:
            ok=True
        else:
            hips=hips[1:]+tiraum(u)
    return ok
```

Grupo II (4+4 valores)

Considere filas de espera com prioridades, em que cada elemento entra na fila com uma dada prioridade (um número inteiro) que lhe permite sair antes de todos os elementos na fila com prioridade inferior, com as seguintes operações:

- `vazia()`: fila vazia;
- `entra(f,x,p)`: fila que resulta da entrada na fila `f` do elemento `x` com prioridade `p`;
- `prox(f)`: próximo elemento que sairá da fila `f`;
- `sai(f)`: fila que resulta da saída do próximo elemento de `f`;
- `compr(f)`: número de elementos de `f`;
- `fpbfQ(e)`: `True` se e só se `e` é uma fila com prioridades bem formada.

Por exemplo, deverá ter-se `prox(entra(entra(vazia(),x,1),y,1))` igual a `x`, mas `prox(entra(entra(vazia(),x,1),y,2))` igual a `y`.

a) Desenvolva em *Python* uma implementação eficiente deste tipo de dados, de modo a que cada fila com prioridades seja representada por uma lista da forma $[(x_1, p_1), ..., (x_n, p_n)]$ onde os elementos surgem por ordem de entrada, isto é, cada x_i é o i-ésimo elemento que entrou na fila e p_i a sua prioridade.

Resolução:

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

def entra(f,x,p):
    return f+[(x,p)]

def posprox(f):
    assert f!=[]
    pos=0
    prio=f[0][1]
    for i in range(1,len(f)):
        if f[i][1]>prio:
            pos=i
            prio=f[i][1]
    return pos

def prox(f):
    return f[posprox(f)][0]

def sai(f):
    f.pop(posprox(f))
    return f

def compr(f):
    return len(f)

def fpbfQ(e):
    def parelemprio(x):
        return (type(x) is tuple) and len(x)==2 and (type(x[1]) is int)

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

b) Desenvolva em *Python*, sobre a camada de abstracção acima desenvolvida e assegurando a independência da implementação, uma função `ordena` que recebendo uma lista de números inteiros a devolve ordenada por ordem crescente (construindo uma fila de espera com prioridades adequadas).

Resolução:

```
def ordena(w):
    fila=vazia()
    for x in w:
        fila=entra(fila,x,x)
    res=[]
    while compr(fila)!=0:
        res=[prox(fila)]+res
        fila=sai(fila)
    return res
```

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.

Diz-se que uma função `f` separa dois valores `a` e `b` se `f(a)` é zero e `f(b)` não, ou vice-versa.

Implemente funcionalmente em *Python* uma função `maxseps` que dada uma lista de funções `funs` calcula o maior valor natural `n` para o qual cada par de valores distintos em `1,...,n` é separável por alguma função em `funs`.

Resolução:

```
def maxsep(funs):
    def sep(a,b):
        return any([(h(a)==0 and h(b)!=0) or (h(a)!=0 and h(b)==0) for h in funs])

    def tentaprox(n):
        if all([sep(i,n+1) for i in range(1,n+1)]):
            return n+1
        else:
            return n

    return fixedpoint(tentaprox,1)
```

Grupo IV (3 valores)

Considere o seguinte programa imperativo `PROG`.

```
r=0
while x!=1:
    r=r+1
    x=x/2
```

Demonstre que é válida a asserção

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

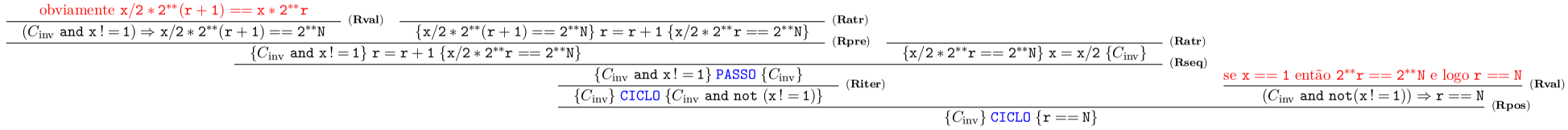
Resolução: Considera-se a estrutura de ciclo inicializado

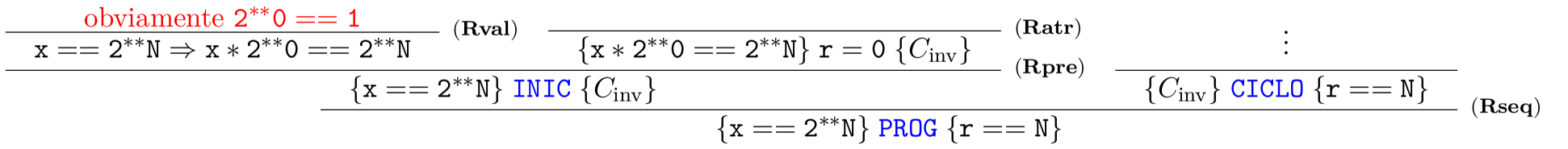
$$\texttt{PROG}\left\{\begin{array}{ll} \texttt{r} = 0 & \left.\vphantom{\texttt{r} = 0}\right\}\texttt{INIC} \\ \texttt{while x} != 1: & \left.\vphantom{\begin{array}{l}\texttt{r} = \texttt{r} + 1 \\ \texttt{x} = \texttt{x}/2\end{array}}\right\}\texttt{CICLO} \\ \quad \begin{array}{l}\texttt{r} = \texttt{r}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

e a condição invariante do ciclo

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

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


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


obviamente 2**0 == 1
x == 2**N ⇒ x * 2**0 == 2**N
(Rval)
{x * 2**0 == 2**N} r = 0 {C_inv}
(Ratr)
{x == 2**N} INIC {C_inv}
(Rpre)
{C_inv} CICLO {r == N}
(Rseq)
{x == 2**N} PROG {r == N}