Instituto Superior Técnico
Lic. em Matemática Aplicada e Computação

Elementos de Programação

18 de Novembro de 2022 Época Normal Duração: 2h

Grupo I (2,5 valores)

Neste exercício não pode usar definições por compreensão nem métodos. As únicas operações sobre listas permitidas são: lista vazia ([]), acesso aos elementos da lista por posição (lista[posição]), seccionamento da lista (lista[posição:posição]), comparação com a lista vazia (==[]), cálculo do comprimento (`len`) e concatenação (+).

Uma forma simples de comprimir uma lista de valores com blocos de valores repetidos consecutivos é contar quantas vezes o valor se repete nesses blocos. Usando esta ideia, uma lista como `w1=[1,1,1,1,1,7,7,1,1,1]` pode ser comprimida para `w2=[1,5,7,2,1,3]` (`1` cinco vezes, depois `7` duas vezes, e depois `1` três vezes). A compressão de uma lista é portanto uma lista da forma `[x1,q1,x2,q2,...,xn,qn]` para uma lista com `q1+q2+...+qn` elementos onde cada `qi` é um inteiro positivo e indica quantas vezes se repete consecutivamente o valor `xi`.

Defina imperativamente em *Python* funções `comprime` e `descomprime` tais que, `comprime` devolve a compressão de uma lista dada, e `descomprime` inverte o processo e recalcula a lista original a partir da sua compressão.

Nomeadamente, `comprime(w1)` deve ser `w2`, e `descomprime(w2)` deve ser `w1`.

Resolução:

```python
def comprime(w):
    r=[]
    i=0
    while i<len(w):
        x=w[i]
        q=1
        i=i+1
        while i<len(w) and w[i]==x:
            q=q+1
            i=i+1
        r=r+[x,q]
    return r

def descomprime(w):
    r=[]
    i=0
    while i<len(w):
        x=w[i]
        q=w[i+1]
        for k in range(q):
            r=r+[x]
        i=i+2
    return r
```

(vsff)

Grupo II (2,5+2,0 valores)

Considere funções quase-nulas (fqn) sobre os naturais, ou seja, funções $f : \mathbb{N}_0 \to \mathbb{N}_0$ tais que $E_f = \{x \in \mathbb{N}_0 : f(x) \neq 0\}$ é um conjunto finito, cujos elementos se denominam excepções. Se $E_f = \{x_1, \ldots, x_k\}$ e sabemos que $f(x_1) = y_1, f(x_2) = y_2 \ldots, f(x_k) = y_k$, tal função pode definir-se por casos na forma:

$$f(n) = \begin{cases} y_1 & \text{se } n = x_1 \\ y_2 & \text{se } n = x_2 \\ & \ldots \\ y_k & \text{se } n = x_k \\ 0 & \text{caso contrário} \end{cases} .$$

As operações relevantes sobre este tipo de dados são as seguintes:

- `nula()`: devolve a função totalmente nula, sem excepções;
- `defval(f,x,y)`: devolve a fqn que é quase igual a `f` mas que vale `y` no ponto `x`;
- `valor(f,x)`: devolve o valor da fqn `f` no ponto `x`;
- `numexc(f)`: devolve o número de excepções da fqn `f`;
- `soma(f,g)`: devolve a fqn que resulta de somar ponto a ponto as fqns `f` e `g`;
- `mult(f,g)`: devolve a fqn que resulta de multiplicar ponto a ponto as fqns `f` e `g`.

a) Desenvolva em Python implementações eficientes para as operações de modo a que cada fqn seja representada por uma lista de pares da forma `[(x1,y1),...,(xk,yk)]` onde `x1,...,xk` são as excepções e `y1,...,yk` os valores não nulos que lhes correspondem.

Por exemplo, a lista `[(5,10),(10,1)]` representa a fqn h com duas excepções, tal que $h(5) = 10$ e $h(10) = 1$.

Resolução:

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

def valor(f,x):
    xcpt=[(a,b) for (a,b) in f if a==x]
    if xcpt==[]:
        return 0
    else:
        return xcpt[0][1]

def defval(f,x,y):
    z=valor(f,x)
    if z==y:
        return f
    elif z==0:
        return f+[(x,y)]
    elif y==0:
        return [(a,b) for (a,b) in f if a!=x]
    else:
        return [(a,b) for (a,b) in f if a!=x]+[(x,y)]
```

(vsff)

```
def numexc(f):
    return len(f)

def soma(f,g):
    xcpt=[a for (a,b) in f]
    xcpt=xcpt+[a for (a,b) in g if a not in xcpt]
    s=[]
    for x in xcpt:
        s=s+[(x,valor(f,x)+valor(g,x))]
    return s

def mult(f,g):
    xcpt=[a for (a,b) in f]
    m=[]
    for x in xcpt:
        y=valor(f,x)*valor(g,x)
        if y!=0:
            m=m+[(x,y)]
    return m
```

b) Desenvolva, sobre a camada de abstracção obtida acima e assegurando a independência da implementação, as seguintes funções.

b1) A função `lista_exc`, que recebendo uma fqn devolve a lista das suas excepções.

Nomeadamente, para a fqn antes descrita, deverá ter-se `lista_exc(h)` igual a `[5,10]`, em particular para `h=defval(defval(nula(),5,10),10,1)`.

Resolução:

```
def lista_exc(f):
    w=[]
    n=numexc(f)
    x=0
    while n!=0:
        if valor(f,x)!=0:
            w=w+[x]
            n=n-1
        x=x+1
    return w
```

b2) A função `solucaoQ`, que recebendo uma fqn `f` e um natural `y` devolve `True` se existe um natural `x` tal que o valor de `f` no ponto `x` é igual a `y`, e `False` caso contrário.

Nomeadamente, `solucaoQ(nula(),1)` deverá ser `False`, e `solucaoQ(h,1)` deverá ser `True`.

Resolução:

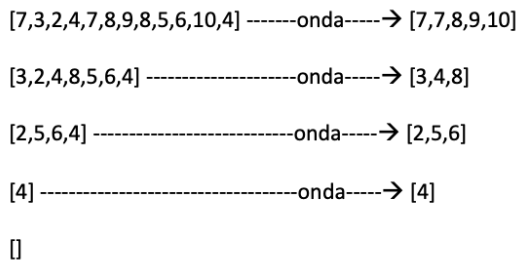
```
def solucaoQ(f,y):
    if y==0:
        return True
    else:
        n=numexc(f)
        x=0
        found=False
        while not(found) and n!=0:
            z=valor(f,x)
            if z==y:
                found=True
            elif z!=0:
                n=n-1
            x=x+1
        return found
```

Grupo III (2,0 valores)

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

Dada uma lista de valores não vazia, pode extrair-se dela uma *onda* crescente (ou melhor, não decrescente) que consiste em tomar o primeiro elemento da lista, depois o próximo elemento da lista que não lhe é inferior, e o próximo elemento que não é inferior a esse, e assim sucessivamente.

O algoritmo de ordenação *wavesort* consiste na extracção sucessiva de ondas a partir da lista original, como se ilustra, ondas essas que devem ser fundidas (tal como no algoritmo *mergesort*) numa só lista ordenada.



Implemente funcionalmente em *Python* uma função `wavesort` que implemente este algoritmo de ordenação.

Resolução:

```
def merge(u1,u2):
    def pick(i,j,r):
        if i<len(u1) and ((j<len(u2) and u1[i]<u2[j]) or j==len(u2)):
            return (i+1,j,r+[u1[i]])
        else:
            return (i,j+1,r+[u2[j]])
    return nest(lambda tr:pick(tr[0],tr[1],tr[2]),len(u1)+len(u2),(0,0,[]))[2]

def wave(w):
    def one(x,remain,done):
        if done!=[] and x<done[-1]:
            return (remain+[x],done)
        else:
            return (remain,done+[x])
    return reduce(lambda p,x:one(x,p[0],p[1]),w,([],[]))

def wavesort(w):
    def step(p,r):
        return (p[0],merge(r,p[1]))
    return fixedpoint(lambda p:step(wave(p[0]),p[1]),(w,[]))[1]
```

(vsff)

Grupo IV (1,0 valores)

Considere o seguinte programa imperativo `PROG1`.

```
stop=0
s=0
i=0
while i!=len(w) and stop==0:
    s=s+w[i]
    if s==0:
        stop=1
    else:
        i=i+1
```

Neste grupo deve resolver apenas uma das alíneas, indicando-a claramente.

a) Indique, justificando, uma condição invariante do ciclo que permita demonstrar que é válida a asserção de correcção parcial

$$\{\texttt{True}\}\ \texttt{PROG}\ \{\texttt{stop} == 0\ \texttt{or}\ \sum_{\texttt{j}\leq\texttt{i}} \texttt{w[j]} == 0\}.$$

Resolução: Considere-se a condição invariante

$$C_{\text{inv}} \equiv (\texttt{stop} == 0\ \texttt{and}\ \texttt{s} == \sum_{\texttt{j}<\texttt{i}} \texttt{w[j]})\ \texttt{or}\ (\texttt{stop} == 1\ \texttt{and}\ \sum_{\texttt{j}\leq\texttt{i}} \texttt{w[j]} == 0).$$

- C_{inv} é garantida pela inicialização:

  Na inicialização obtém-se `stop==s==i==0` e portanto, como $\sum_{\texttt{j}<\texttt{i}} \texttt{w[j]} == 0$, obtém-se $(\texttt{stop} == 0\ \texttt{and}\ \texttt{s} == \sum_{\texttt{j}<\texttt{i}} \texttt{w[j]})$.
- C_{inv} é mantida pelo passo do ciclo quando a guarda do ciclo é verdadeira:

  Iniciando o passo com C_{inv} e `i!=len(w) and stop==0`, há dois casos a considerar.
  Se após o passo se tem `stop==1`, então o valor de `i` fica inalterado, e o novo valor de `s` é zero, o que pela C_{inv}, significa que $\sum_{\texttt{j}<\texttt{i}} \texttt{w[j]} + \texttt{w[i]} == \sum_{\texttt{j}\leq\texttt{i}} \texttt{w[j]} == 0$.
  Se após o passo se mantém `stop==0`, então o valor de `i` é incrementado, e para o novo valor de `s`, pela C_{inv}, tem-se $\texttt{s} == \sum_{\texttt{j}<\texttt{i}} \texttt{w[j]} + \texttt{w[i]} == \sum_{\texttt{j}<\texttt{i}+1} \texttt{w[j]}$.
- C_{inv} com a guarda do ciclo falsa garante o objectivo:

  De C_{inv}, se `stop!=0` então `stop==1`, e portanto tem-se $\sum_{\texttt{j}\leq\texttt{i}} \texttt{w[j]} == 0$.
  Senão tem-se imediatamente `stop==0`.

b) Indique, justificando, uma expressão variante do ciclo que permita demonstrar que `PROG` termina sempre que executado a partir de um estado que satisfaça

$$\texttt{type(w)} == \texttt{list}.$$

Resolução: Considera-se a expressão variante

$$\texttt{len(w)} - \texttt{i} - \texttt{stop}$$

e a ordem usual de $\mathbb{N}_0$.

Obviamente, sempre que o ciclo é executado e a guarda é verdadeira tem-se `stop==0` e portanto `len(w)-i` é um número natural. Para garantir que o passo do ciclo, de facto, garante que a expressão variante diminui temos de analisar dois casos.

(vsff)

– Se na execução do passo se tem `s==0`:

  O valor de `stop` passa de `0` para `1`, ficando `i` inalterado, pelo que o novo valor de $\texttt{len(w)} - \texttt{i} - \texttt{stop} == \texttt{len(w)} - \texttt{i} - \texttt{1} < \texttt{len(w)} - \texttt{i} - \texttt{0}$ (que é o anterior valor da expressão variante).

– Se na execução do passo se tem `s!=0`:

  O valor de `stop` mantém-se e `i` é incrementado, pelo que o novo valor da expressão variante é $\texttt{len(w)} - (\texttt{i} + \texttt{1}) - \texttt{stop} < \texttt{len(w)} - \texttt{i} - \texttt{stop}$.