Instituto Superior Técnico
Lic. em Matemática Aplicada e Computação

Elementos de Programação

6 de Fevereiro de 2023 Época de Recurso Duração: 2h

Grupo I (2,5 valores)

Neste exercício não pode usar definições por compreensão nem métodos. As únicas operações sobre listas permitidas são: enumeração ([x1,...,xn]), 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 (+).

Defina imperativamente em *Python* funções `maxrep` e `maxconsec` que dada uma lista `w` devolvam, respectivamente, o número máximo de ocorrências repetidas de um elemento em `w`, e o número máximo de ocorrências repetidas consecutivas de um elemento em `w`.

Nomeadamente, `maxrep([5,5,1,1,1,5,5])` deverá ser 4 (pois o elemento que mais se repete na lista é 5, que ocorre 4 vezes), e `maxconsec([5,5,1,1,1,5,5])` deverá ser 3 (pois há 3 ocorrências consecutivas de 1, mas apenas 2 consecutivas de 5).

Resolução:

```
def maxconsec(w):
    i=0
    m=0
    while m+i<len(w):
        x=w[i]
        j=i+1
        while j<len(w) and w[j]==x:
            j=j+1
        m=max(m,j-i)
        i=j
    return m

def pertence(x,w):
    f=False
    i=0
    while not(f) and i<len(w):
        if w[i]==x:
            f=True
        else:
            i=i+1
    return f

def maxrep(w):
    vistos=[]
    falta=len(w)
    i=0
    m=0
    while falta>m:
        x=w[i]
        if not(pertence(x,vistos)):
            vistos=vistos+[x]
            c=0
            j=i
            poss=falta
            while poss>0 and c+poss>m and j<len(w):
                if w[j]==x:
                    falta=falta-1
                    c=c+1
                poss=poss-1
                j=j+1
            m=max(m,c)
        i=i+1
    return m
```

(vsff)

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

Considere listas quase-constantes (lqc), ou seja, listas de valores que são todos iguais a uma certa constante `c`, com a possível excepção de algumas posições `i1,...,ik` com valores `c1,...,ck` todos diferentes de `c`. Chamamos *núcleo* da lqc à lista `[(i1,c1),...,(ik,ck)]`.

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

- `listac(n,c)`: devolve a lista com `n` elementos todos iguais a `c`, sem excepções;
- `actual(w,i,x)`: devolve a lqc que é quase igual a `w` mas com o valor `x` na posição `i`;
- `compr(w)`: devolve o número de elementos da lqc `w`;
- `const(w)`: devolve a constante associada à lqc `w`;
- `nucleo(w)`: devolve o núcleo da lqc `w`;
- `novaconst(w,d)`: devolve a lqc com constante associada `d` que tem exactamente os mesmos elementos que a lqc `w`.

a) Desenvolva em Python implementações eficientes para as operações de modo a que cada lqc seja representada por um quádruplo da forma `(n,c,[i1,...,ik],[c1,...,ck])` onde `n` é o comprimento da lqc, `c` a constante associada, `i1,...,ik` são as posições correspondentes às excepções e `c1,...,ck` os respectivos valores.

Por exemplo, `(10,0,[0,9],[1,1])` representa a lqc de comprimento 10 com o valor `0` em todas as posições, com excepção do valor `1` no início e fim da lista.

Resolução:

```python
def posin(w,x):
    if x in w:
        return w.index(x)
    else:
        return -1

def listac(n,c):
    return (n,c,[],[])

def actual(lqc,i,x):
    (n,c,pos,val)=lqc
    p=posin(pos,i)
    if p==-1 and x!=c:
        pos=pos+[i]
        val=val+[x]
    elif p>-1 and x==c:
        pos.pop(p)
        val.pop(p)
    elif p>-1:
        val[p]=x
    return (n,c,pos,val)

def compr(lqc):
    (n,c,pos,val)=lqc
    return n

def const(lqc):
    (n,c,pos,val)=lqc
    return c

def nucleo(lqc):
    (n,c,pos,val)=lqc
    return [(pos[i],val[i]) for i in range(len(pos))]

def novaconst(lqc,d):
    (n,c,pos,val)=lqc
    xs=[i for i in range(n) if (i not in pos and c!=d) or (i in pos and val[posin(pos,i)]!=d)]
    return (n,d,xs,[(val[posin(pos,i)] if i in pos else c) for i in xs])
```

(vsff)

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

b1) A função `valor`, que recebendo uma lqc `w` e uma posição `i` devolve o valor na posição `i` de `w`.

Nomeadamente, para a lqc antes descrita, deverá ter-se `valor(lqc,0)` igual a `1`, e `valor(lqc,1)` igual a `0`.

b2) A função `mapear`, que recebendo uma função `f` e uma lqc `w` devolve a lqc que resulta de aplicar, posição a posição, a função `f` a cada um dos elementos de `w`.

Nomeadamente, para a lqc antes descrita, `mapear(lambda x:x+1,lqc)` deverá corresponder à lqc de comprimento 10 com o valor `1` em todas as posições, com excepção do valor `2` no início e fim da lista.

Resolução:

```
def valor(lqc,i):
    assert i<compr(lqc)
    nuc=nucleo(lqc)
    p=posin([j for (j,x) in nuc],i)
    if p==-1:
        return const(lqc)
    else:
        return nuc[p][1]


def mapear(f,lqc):
    fc=f(const(lqc))
    res=listac(compr(lqc),fc)
    nuc=nucleo(lqc)
    for (i,x) in nuc:
        fx=f(x)
        if fx!=fc:
            res=actual(res,i,fx)
    return res
```

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.

A *união ordenada* de duas listas `u,v` também ordenadas e sem repetições é a lista ordenada e sem repetições que contém todos os valores de `u` e de `v`. Nomeadamente, a união ordenada de `[2,4,6]` e `[1,2,5,6]` é a lista `[1,2,4,5,6]`.

Uma lista `W` de listas ordenadas e sem repetições diz-se *fechada para uniões* se dadas quaisquer listas `u,v` em `W` a união ordenada de `u,v` também está em `W`. O *fecho para uniões* de uma lista `W` de listas ordenadas é a menor lista de listas que é fechada para uniões e contém todas as listas de `W`. Nomeadamente, `[[1],[2],[1,2,3]]` não é fechada para uniões pois a união ordenada das suas duas primeiras listas é a lista `[1,2]` que não está presente, enquanto `[[1],[2],[1,2],[1,2,3]]` é o correspondente fecho para uniões.

Defina funcionalmente em *Python* uma função `fechu` que, dada uma lista `W` de listas ordenadas e sem repetições devolve o seu fecho para uniões.

(vsff)

Resolução:

```
def unord(u,v):
    def fundir(i,j,r):
        if i==len(u) or j==len(v):
            return (len(u),len(v),r+u[i:]+v[j:])
        elif u[i]==v[j]:
            return (i+1,j+1,r+[u[i]])
        elif u[i]<v[j]:
            return (i+1,j,r+[u[i]])
        else:
            return (i,j+1,r+[v[j]])

    return fixedpoint(lambda t:fundir(t[0],t[1],t[2]),(0,0,[]))[-1]

def fechu(W):
    def result(new,newold):
        def update(res,u):
            if u in newold+res:
                return res
            else:
                return res+[u]

        def pairwise(i,j,res):
            if i==len(new):
                return (i,j,res)
            elif j==len(newold):
                return (i+1,i+2,res)
            else:
                return (i,j+1,update(res,unord(new[i],newold[j])))
        return fixedpoint(lambda t:pairwise(t[0],t[1],t[2]),(0,1,[]))[-1]

    def passo(old,new):
        return (old+new,result(new,new+old))

    return fixedpoint(lambda p:passo(p[0],p[1]),([],W))[0]
```

Grupo IV (1,0 valores)

Considere o seguinte programa imperativo PROG.

```
go=1
fo=0
i=0
while go!=0 and fo==0:
    if f(i)==x:
        fo=1
    elif f(i)<x:
        i=i+1
    else:
        go=0
```

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

(vsff)

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

$$\{\texttt{True}\}\ \texttt{PROG}\ \{\texttt{go==0 or f(i)==x}\}.$$

Resolução: Considere-se a condição invariante

$$C_{\text{inv}} \equiv \texttt{(fo==0 or go==0 or f(i)==x)}.$$

- C_{inv} é garantida pela inicialização:

  Na inicialização obtém-se `fo==0`.

- C_{inv} é mantida pelo passo do ciclo quando a guarda do ciclo é verdadeira:

  Iniciando o passo com C_{inv} e `go!=0 and fo==0`, há três casos a considerar.
  (i) Se é executada a primeira alternativa então tem-se `f(i)==x`, sendo apenas alterado o valor de `fo`, pelo que se terá C_{inv} no final do passo.
  (ii) Se é executada a segunda alternativa então o valor de `fo` não é alterado, sendo que a guarda do ciclo garante que se terá ainda `fo==0` e C_{inv} no final do passo.
  (iii) Se é executada a terceira alternativa então vai ter-se no final do passo `go==0`, portanto também C_{inv}.

- C_{inv} com a guarda do ciclo falsa garante o objectivo:

  Se a guarda do ciclo é falsa porque `go==0` tem-se imediatamente a condição objectivo `(go==0 or f(i)==x)`.
  Senão, tem-se `go!=0` e a guarda do ciclo é falsa porque `fo!=0`. Nesse caso, a partir de C_{inv}, conclui-se imediatamente que `f(i)==x`, pelo que também segue a condição objectivo `(go==0 or f(i)==x)`.

b) Indique, justificando, uma expressão variante do ciclo que permita demonstrar que `PROG` termina sempre que executado a partir de um estado em que `f`: $\mathbb{N}_0 \to \mathbb{N}_0$ seja uma função estritamente crescente e `x` $\in \mathbb{N}_0$.

Resolução: Considera-se a expressão variante

$$\texttt{go} - \texttt{fo} + \texttt{x} - \texttt{i}$$

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

Obviamente, sempre que o ciclo é executado e a guarda é verdadeira tem-se `go==1` e `fo==o`, pelo que o valor da expressão variante é `1+x-i`, necessariamente um número inteiro. Note-se que se `1+x-i<0` então `x<i`, e como pelas propriedades da função `f` se tem necessariamente `i<=f(i)`, resultaria que `x<f(i)` despoletando a execução da terceira alternativa do ciclo, que terminaria com `go==0`.

A expressão variante é então necessariamente um valor natural. Para garantir que o seu valor decresce por execução do passo do ciclo, temos de analisar três casos.

- Se é executada a primeira alternativa:

  O valor de `fo` passa de `0` para `1`, pelo que `go-fo+x-i` decresce uma unidade.

- Se é executada a segunda alternativa:

  O valor de `i` é incrementado, pelo que de novo `go-fo+x-i` decresce uma unidade.

- Se é executada a terceira alternativa:

  O valor de `go` passa de `1` para `0`, pelo que `go-fo+x-i` decresce, mais uma vez, uma unidade.