Instituto Superior Técnico

Lic. em Matemática Aplicada e Computação

Mestrado Integrado em Eng. Biomédica

Elementos de Programação

4 de Fevereiro de 2019 Exame 2 Duração: 2h30

Grupo I (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 (+).

Um *sudoku* é representado por uma matriz 9×9 com entradas $1, 2, 3, \ldots, 9$, valores estes que devem ocorrer exactamente uma vez em cada linha, em cada coluna, e em cada um dos nove quadrados 3×3 da matriz (ver figura).

Defina imperativamente em *Python* uma função `sudokuQ` que dada uma matriz `m` de dimensão 9×9 com entradas $1, 2, 3, \ldots, 9$, devolve `True` se `m` representa um sudoku, e `False` caso contrário.

1	5	4	8	7	3	2	9	6
3	8	6	5	9	2	7	1	4
7	2	9	6	4	1	8	3	5
8	6	3	7	2	5	1	4	9
9	7	5	3	1	4	6	2	8
4	1	2	9	6	8	3	5	7
6	3	1	4	5	7	9	8	2
5	9	8	2	3	6	4	7	1
2	4	7	1	8	9	5	6	3

Resolução:

```
def pos(a,b):
    if a<9:
        return (a,b)
    elif a<18:
        return (b,a-9)
    else:
        return (3*((a-18)//3)+b//3,3*((a-18)%3)+b%3)

def sudokuQ(m):
    t=[]
    for n in range(10):
        t=t+[0]
    ok=True
    a=0
    while ok and a<27:
        b=0
        while ok and b<9:
            (i,j)=pos(a,b)
            x=m[i][j]
            if t[x-1]==a:
                t[x-1]=a+1
            else:
                ok=False
            b=b+1
        a=a+1
    return ok
```

Grupo II (4+4 valores) (pesa a nota do projecto)

Considere *polinómios* (numa variável x) com as seguintes operações:

- `mono(a,n)`: polinómio apenas com o monómio $\mathtt{a}x^{\mathtt{n}}$;
- `sum(p,q)`: soma `s` dos polinómios `p`, `q`, ou seja, $\mathtt{s}(x) = \mathtt{p}(x) + \mathtt{q}(x)$;
- `prod(p,q)`: produto `r` dos polinómios `p`, `q`, ou seja, $\mathtt{r}(x) = \mathtt{p}(x) \times \mathtt{q}(x)$;
- `comp(p,q)`: composição `t` dos polinómios `p`, `q`, ou seja, $\mathtt{t}(x) = \mathtt{p}(\mathtt{q}(x))$;
- `degree(p)`: grau do polinómio `p`;
- `maincoef(p)`: coeficiente do monómio de maior grau do polinómio `p`;
- `zeroQ(p)`: `True` se o polinómio `p` é nulo, e `False` caso contrário;
- `eval(p,a)`: valor do polinómio `p` no ponto $x = \mathtt{a}$, ou seja, $\mathtt{p}(\mathtt{a})$.

Em *Python*, pretende-se representar cada polinómio de grau `n` como a lista de coeficientes dos seus monómios por ordem decrescente de grau. Nomeadamente, um polinómio da forma $\mathtt{a_n}x^{\mathtt{n}} + \mathtt{a_{n-1}}x^{\mathtt{n}-1} + \cdots + \mathtt{a_2}x^2 + \mathtt{a_1}x + \mathtt{a_0}$ deve corresponder à lista $[\mathtt{a_n}, \mathtt{a_{n-1}}, \ldots, \mathtt{a_2}, \mathtt{a_1}, \mathtt{a_0}]$, garantindo-se que $\mathtt{a_n} \neq 0$ a menos que $\mathtt{n} = 0$ e o polinómio seja nulo.

a) Apresente implementações eficientes para as operações identificadas.

Resolução:

```
def mono(a,n):
    if a==0 and (n>0):
        return [0]
    else:
        return [a]+[0 for i in range(n)]

def clean(p):
    while len(p)>1 and p[0]==0:
        p=p[1:]
    return p

def sum(p,q):
    if len(p)<len(q):
        p,q=q,p
    q=[0 for i in range(len(p)-len(q))]+q
    return clean(list(map(lambda x,y:x+y,p,q)))

def prod(p,q):
    r=[]
    k=degree(p)+degree(q)
    for n in range(k+1):
        x=0
        for i in range(n+1):
            j=n-i
            if i<len(p) and j<len(q):
                x=x+p[-i-1]*q[-j-1]
        r=[x]+r
    return r
```

(vsff)

```
def comp(p,q):
    powq=[[1]]
    n=len(p)-1
    for i in range(n):
        powq=powq+[prod(q,powq[-1])]
    r=[0]
    for i in range(n+1):
        r=sum(r,prod([p[n-i]],powq[i]))
    return r

def degree(p):
    return len(p)-1

def maincoef(p):
    return p[0]

def zeroQ(p):
    return p==[0]

def eval(p,a):
    r=p[0]
    for i in range(1,len(p)):
        r=r*a+p[i]
    return r
```

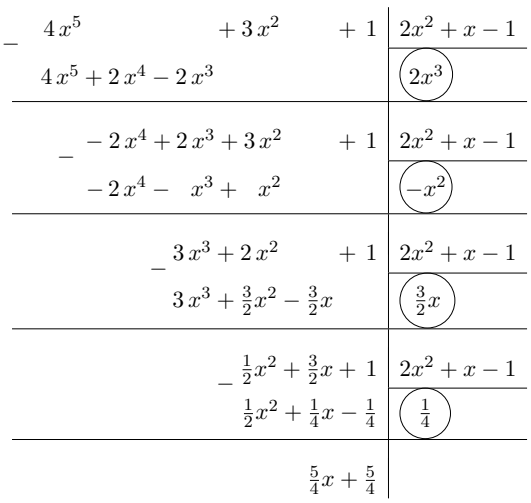
b) Desenvolva, sobre a camada de abstracção obtida acima e assegurando a independência da implementação, as seguintes funções:

b1) `derivative`, que dado um polinómio `p` devolve o polinómio `d` correspondente à sua derivada, i.e., $\mathtt{d}(x) = \mathtt{p}'(x)$;

Resolução:

```
def derivative(p):
    n=degree(p)
    r=mono(0,0)
    s=mono(-1,0)
    for i in range(n-1,-1,-1):
        if degree(p)==i+1:
            a=maincoef(p)
            r=sum(r,mono(a*(i+1),i))
            p=sum(p,prod(s,mono(a,i+1)))
    return r
```

b2) `division`, que dados polinómios `p,q` devolve o par `(d,r)` tal que $\mathtt{d}(x)$ é o quociente da divisão de $\mathtt{p}(x)$ por $\mathtt{q}(x)$, e $\mathtt{r}(x)$ o resto da divisão.



A figura ilustra a divisão de $\mathtt{p}(x) = 4x^5 + 3x^2 + 1$ por $\mathtt{q}(x) = 2x^2 + x - 1$. O quociente da divisão é $\mathtt{d}(x) = 2x^3 - x^2 + \frac{3}{2}x + \frac{1}{4}$ e o resto é $\mathtt{r}(x) = \frac{5}{4}x + \frac{5}{4}$. Como seria de esperar, tem-se $\mathtt{p}(x) = \mathtt{q}(x) \times \mathtt{d}(x) + \mathtt{r}(x)$. A divisão é obtida subtraindo repetidamente ao dividendo ($\mathtt{p}(x)$, ou o que dele restar) o produto do divisor $\mathtt{q}(x)$ por um monómio escolhido de forma a igualar os monómios de maior grau de ambos. O processo termina quando o grau do dividendo é inferior ao grau do divisor, caso em que o dividendo se denomina o resto da divisão. O quociente da divisão obtém-se somando os vários monómios usados ao longo da divisão (dentro de círculos, na figura).

Resolução:

```
def division(p,q):
    d=mono(0,0)
    s=mono(-1,0)
    while degree(p)>=degree(q):
        m=mono(maincoef(p)/maincoef(q),degree(p)-degree(q))
        d=sum(d,m)
        p=sum(p,prod(s,prod(m,q)))
    return (d,p)
```

Grupo III (4 valores)

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

Um *problema de cobertura* é representado por uma matriz quadrada `m` com entradas numéricas. Dado um tal problema, uma *cobertura* de `m` consiste de uma lista `c` com valores em $0, \ldots, \texttt{len}(\texttt{m}) - 1$ que satisfaça a seguinte condição:

- para cada entrada não nula `m[i][j]` de `m` tem-se que `i` ocorre em `c`, ou `j` ocorre em `c`, ou ambos.

Por exemplo, `[0,2]` é uma cobertura da matriz `[[0,1,0],[1,0,1],[0,1,0]]`. Essa cobertura não é mínima, no entanto, pois `[1]` é também uma cobertura do mesmo problema.

Implemente funcionalmente em *Python* uma função `mincover` que dado um problema de cobertura devolve uma sua cobertura de tamanho mínimo.

Resolução:

```
def coverQ(m,c):
    return all([(i in c) or (j in c) for i in range(len(m)) for j in range(len(m)) if m[i][j]!=0])

def mincover(m):
    def test(hyps):
        if coverQ(m,hyps[0]):
            return hyps
        else:
            return hyps[1:]+[hyps[0]+[i] for i in range(len(m)) if not(i in hyps[0])]

    return fixedpoint(test,[[]])[0]
```

Grupo IV (3 valores)

Considere o seguinte programa imperativo PROG.

```
r=[]
a=w[0]
i=1
while i!=len(w):
    r=r+[w[i]-a]
    a=w[i]
    i=i+1
```

Demonstre que é válida a asserção

$$\{\texttt{len(w)} > 0\}\ \texttt{PROG}\ \{\texttt{w[len(w)} - 1] == \texttt{w[0]} + \sum_{\texttt{x in r}} \texttt{x}\}.$$

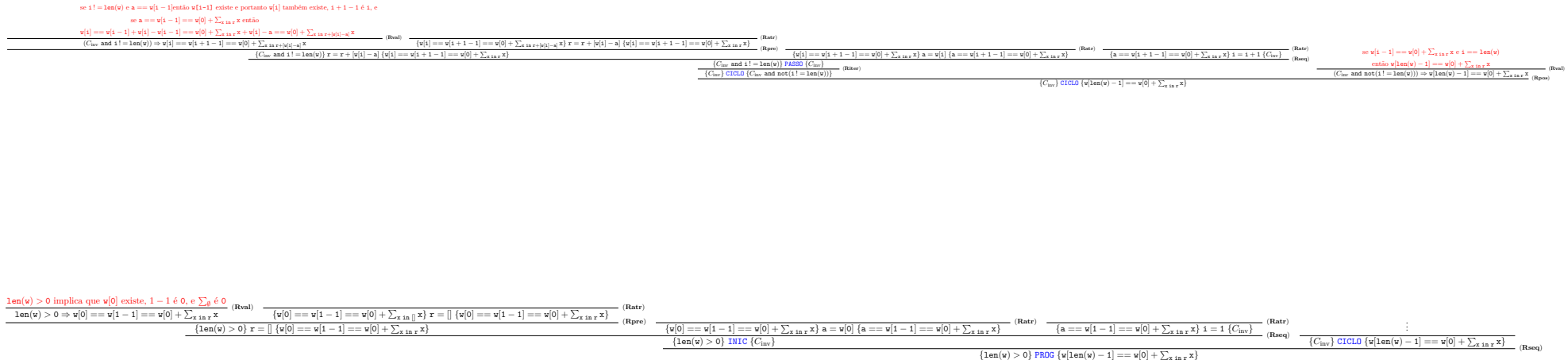
Resolução: Considera-se a estrutura de ciclo inicializado

$$\texttt{PROG}\begin{cases} \left.\begin{array}{l} \texttt{r} = [] \\ \texttt{a} = \texttt{w[0]} \\ \texttt{i} = 1 \end{array}\right\}\ \texttt{INIC} \\ \texttt{while i} \mathrel{!}= \texttt{len(w)}: \\ \quad \left.\begin{array}{l} \texttt{r} = \texttt{r} + [\texttt{w[i]} - \texttt{a}] \\ \texttt{a} = \texttt{w[i]} \\ \texttt{i} = \texttt{i} + 1 \end{array}\right\}\ \texttt{PASSO} \end{cases}\ \texttt{CICLO}$$

e a condição invariante do ciclo

$$C_{\text{inv}} \equiv (\texttt{a} == \texttt{w[i} - 1] == \texttt{w[0]} + \sum_{\texttt{x in r}} \texttt{x}).$$

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


se i ! = len(w) e a == w[i − 1]então w[i−1] existe e portanto w[i] também existe, i + 1 − 1 é i, e
se a == w[i − 1] == w[0] + ∑x in r x então
w[i] == w[i − 1] + w[i] − w[i − 1] == w[0] + ∑x in r x + w[i] − a == w[0] + ∑x in r+|w[i]−a| x
(Cinv and i ! = len(w)) ⇒ w[i] == w[i + 1 − 1] == w[0] + ∑x in r+|w[i]−a| x
(Rval)
{w[i] == w[i + 1 − 1] == w[0] + ∑x in r+|w[i]−a| x} r = r + [w[i] − a] {w[i] == w[i + 1 − 1] == w[0] + ∑x in r x}
(Ratr)
(Cinv and i ! = len(w)) r = r + [w[i] − a] {w[i] == w[i + 1 − 1] == w[0] + ∑x in r x}
(Rpre)
{w[i] == w[i + 1 − 1] == w[0] + ∑x in r x} a = w[i] {a == w[i + 1 − 1] == w[0] + ∑x in r x}
(Ratr)
{a == w[i + 1 − 1] == w[0] + ∑x in r x} i = i + 1 {Cinv}
(Ratr)
(Cinv and i ! = len(w)) PASSO {Cinv}
(Rseq)
{Cinv} CICLO {Cinv and not(i ! = len(w))}
(Ritar)
se w[i − 1] == w[0] + ∑x in r x e i == len(w)
então w[len(w) − 1] == w[0] + ∑x in r x
(Cinv and not(i ! = len(w))) ⇒ w[len(w) − 1] == w[0] + ∑x in r x
(Rval)
{Cinv} CICLO {w[len(w) − 1] == w[0] + ∑x in r x}
(Rpos)
len(w) > 0 implica que w[0] existe, 1 − 1 é 0, e ∑∅ é 0
len(w) > 0 ⇒ w[0] == w[1 − 1] == w[0] + ∑x in r x
(Rval)
{w[0] == w[1 − 1] == w[0] + ∑x in [] x} r = [] {w[0] == w[1 − 1] == w[0] + ∑x in r x}
(Ratr)
{len(w) > 0} r = [] {w[0] == w[1 − 1] == w[0] + ∑x in r x}
(Rpre)
{w[0] == w[1 − 1] == w[0] + ∑x in r x} a = w[0] {a == w[1 − 1] == w[0] + ∑x in r x}
(Ratr)
{a == w[1 − 1] == w[0] + ∑x in r x} i = 1 {Cinv}
(Ratr)
{len(w) > 0} INIC {Cinv}
(Rseq)
{Cinv} CICLO {w[len(w) − 1] == w[0] + ∑x in r x}
(Rseq)
{len(w) > 0} PROG {w[len(w) − 1] == w[0] + ∑x in r x}