Fundamentos da Programação

Solução do Exame

10 de Janeiro de 2019

11:30–13:30

1. Usando palavras suas e, no máximo, em três linhas explique os seguintes conceitos. Explicações dadas através de exemplos serão classificadas com zero valores.

   (a) **(0.5)** Abstração.
   Resposta:
   Consiste em ignorar certos aspectos de uma entidade, considerando apenas os aspectos relevantes.

   (b) **(0.5)** Abstracção procedimental.
   Resposta:
   Consiste em considerar o que um programa faz e ignorar o modo como o faz.

   (c) **(0.5)** Abstracção de dados.
   Resposta:
   Consiste em considerar as propriedades de um tipo de dados, ignorando o modo como este é representado.

2. **(1.0)** Escreva a função `soma_digitos` que recebe um número inteiro positivo e devolve a soma dos seus dígitos. Não pode recorrer a cadeias de caracteres nem a listas. A sua função deve verificar a correção do argumento. Por exemplo,

```
>>> soma_digitos(235)
10
>>> soma_digitos(23.5)
ValueError: O argumento deve ser inteiro positivo
```

   Resposta:

```
def soma_digitos(num):
    if isinstance(num, int) and num >= 0:
        soma = 0
        while num != 0:
            soma = soma + num % 10
            num = num // 10
        return soma
    else:
        raise ValueError ('O argumento deve ser inteiro positivo')
```

3. **(1.0)** Um número inteiro, n, diz-se *triangular* se existir um inteiro m tal que $n = 1+2+\ldots+(m-1)+m$. Escreva uma função chamada `triangular` que recebe um número inteiro positivo `n`, e cujo valor é `True` apenas se o número for triangular. No caso de `n` ser `0` deverá devolver `False`. Não é necessário verificar a correção do argumento. Por exemplo,

```
>>> triangular(6)
True
>>> triangular(8)
False
```

Resposta:

```
def triangular (n):
    soma = 0
    i = 1
    while soma < n:
        soma = soma + i
        i = i + 1
        if soma == n:
            return True
    return False
```

4. **(1.5)** Escreva a função `parte` que recebe como argumentos uma lista, `lst`, e um elemento, `e`, e que devolve uma lista de dois elementos, contendo na primeira posição a lista com os elementos de `lst` menores que `e`, e na segunda posição a lista com os elementos de `lst` maiores ou iguais a `e`. Não é necessário verificar a correção dos argumentos. Por exemplo,

```
>>> parte([2, 0, 12, 19, 5], 6)
[[2, 0, 5], [12, 19]]
>>> parte([7, 3, 4, 12], 3)
[[], [7, 3, 4, 12]]
```

Resposta:

```
def parte(lst, el):
    menores = []
    maiores = []
    for e in lst:
        if e < el:
            menores = menores + [e]
        else:
            maiores = maiores + [e]
    return [menores, maiores]
```

5. **(1.5)** Um método de codificar mensagens corresponde a dividir o alfabeto em duas partes iguais, escrevendo cada uma das partes em linhas consecutivas:

```
a b c d e f g h i j k l m
n o p q r s t u v w x y z
```

Ao codificar uma mensagem, cada letra é substituída pela letra na mesma posição que aparece na primeira linha ou na segunda linha, consoante a letra se encontra, respetivamente na segunda linha ou na primeira linha. Por exemplo `c` é transformado

em `p` e `z` é transformado em `m`. Todos os caracteres que não correspondam a letra minúsculas aparecem sem transformação. Escreva uma função que recebe uma cadeia de caracteres e codifica-a através do algoritmo indicado. Não é necessário verificar a correção do argumento. Por exemplo,

```
>>> codifica('Este exercicio nao e tao dificil como parece.')
'Efgr rkrepvpvb anb r gnb qvsvpvy pbzb cnerpr.'
```

Resposta:

```
def codifica(frase):
      res = ''
      for car in frase:
            if ord('a') <= ord(car) <= ord('m'):
                  car = chr(ord('n') + ord(car) - ord('a'))
            elif ord('n') <= ord(car) <= ord('z'):
                  car = chr(ord('a') + ord(car) - ord('n'))
            res = res + car
      return res
```

Solução alternativa:

```
def codifica(frase):

    def posicao(l, linha):
        i = 0
        while i < len(linha):
            if l == linha[i]:
                return i
            i = i + 1
        return -1

    linha_sup = 'abcdefghijklm'
    linha_inf = 'nopqrstuvwxyz'
    res = ''
    for l in frase:
        if 'a' <= l <= 'z':
            pos = posicao(l, linha_sup)
            if pos >= 0: # a letra está na linha superior
                res= res + linha_inf[pos]
            else:  # a letra está na linha inferior
                pos = posicao(l, linha_inf)
                res = res + linha_sup[pos]
        else:
            res = res + l
    return res
```

6. Escreva a função `soma_n_vezes` que recebe três argumentos, `a`, `b` e `n`, e que devolve o valor de somar `n` vezes `a` a `b`, isto é, b + a + a + ... + a, `n` vezes. Não é necessário verificar a correção dos argumentos. A sua função não pode usar a operação `*`.

   (a) **(1.0)** Usando um processo iterativo.
   Resposta:

```
def soma_n_vezes_iter(a, b, n):
    res = b
    for i in range(n):
        res = res + a
    return res
```

(b) **(1.0)** Usando recursão com operações adiadas.
Resposta:

```
def soma_n_vezes(a, b, n):

    if n == 0:
        return b
    else:
        return a + soma_n_vezes(a, b, n - 1)
```

(c) **(1.0)** Usando recursão de cauda.
Resposta:

```
def soma_n_vezes_rc(a, b, n):

    def soma_n_vezes_rc_aux(n, res):
        if n == 0:
            return res
        else:
            return soma_n_vezes_rc_aux(n - 1, res + a)

    return soma_n_vezes_rc_aux(n , b)
```

7. **(1.5)** Usando um ou mais dos funcionais sobre listas (`filtra`, `transforma`, `acumula`, ou as funções embutidas `filter`, `map`, `reduce`), escreva uma função que devolve a soma dos quadrados dos elementos de uma lista. Não é necessário verificar que o argumento é uma lista. A sua função não pode usar a função `sum` e o corpo da sua função apenas pode ter uma instrução, a instrução `return`.

Resposta:

```
def soma_quadrados_lista(lst):
    return acumula(transforma(lst, quadrado), \
                   lambda x, y: x + y)
```

8. Considere a seguinte gramática em notação BNF, na qual o símbolo inicial é ⟨prim⟩:

⟨prim⟩ ::= a ⟨seg⟩ a | a ⟨prim⟩ a

⟨seg⟩ ::= b ⟨seg⟩ b | b ⟨ter⟩ b

⟨ter⟩ ::= c | c ⟨ter⟩

(a) **(0.5)** Indique os símbolos terminais e os símbolos não terminais da gramática.
Símbolos terminais:
Resposta:
a, b, c
Símbolos não terminais:
Resposta:
⟨prim⟩, ⟨seg⟩, ⟨ter⟩

(b) **(0.5)** Indique, justificando no caso de não pertencer, quais das seguintes expressões pertencem ou não pertencem ao conjunto de frases da linguagem definida pela gramática:

i. `abbcba`
Resposta:
Não pertence pois falta um `b` depois do `c`.

ii. `abbbbbbcccccccbbbbbba`
Resposta:
Pertence.

iii. `abc`
Resposta:
Não pertence pois falta `ba` depois do `c`.

iv. `primsegter`
Resposta:
Não pertence pois `primsegter` não são símbolos terminais.

v. `aabbccbbaa`
Resposta:
Pertence.

(c) **(1.5)** Escreva a função `reconhece`, que recebe como argumento uma cadeia de caracteres e devolve verdadeiro se o seu argumento corresponde a uma frase da linguagem definida pela gramática e falso em caso contrário. Não é necessário validar os argumentos.
Resposta:

```
def reconhece(cad):

      def eh_ter(cad):
            return len(cad) >= 1 and cad == len(cad) * 'c'

      def eh_seg(cad):
            return len(cad) >= 3 and cad[0] == cad[-1] == 'b' and \
                   (eh_ter(cad[1 : -1]) or eh_seg(cad[1 : -1]))

      def eh_prim(cad):
            return len(cad) >= 5 and cad[0] == cad[-1] == 'a' and \
                   (eh_seg(cad[1 : -1]) or eh_prim(cad[1 : -1]))

      return eh_prim(cad)
```

Solução alternativa:

```
def reconhece(f):
    num_as = 0
    i = 0
    while i < len(f) and f[i] == 'a':
        num_as = num_as + 1
        i = i + 1
    if i == len(f) or num_as == 0:
        return False
    else:
        num_bs = 0
        while  i < len(f) and f[i] == 'b':
            num_bs = num_bs + 1
            i = i + 1
```

```
            if i == len(f) or num_bs == 0:
                return False
            else:
                num_cs = 0
                while i < len(f) and f[i] == 'c':
                    num_cs = num_cs + 1
                    i = i + 1
                if i == len(f) or num_cs == 0:
                    return False
                else:
                    while num_bs !=0 and i < len(f) and f[i] == 'b':
                        numb_bs = num_bs - 1
                        i = i + 1
                    if i == len(f) or f[i] != 'a':
                        return False
                    else:
                        while num_as != 0 and i < len(f) and f[i] == 'a':
                            num_as = num_as - 1
                            i = i + 1
                        return i == len(f)
```

9. **(1.5)** Escreva a função `multiplos_filtrados` que recebe um natural, `n` e um predicado `pred` e que devolve o dicionário cujas chaves são os naturais inferiores ou iguais a `n` que satisfazem o predicado `pred` e cujos valores associados às chaves são os `n` primeiros múltiplos da chave. Não é necessário validar os argumentos. Por exemplo,

```
>>> multiplos_filtrados(4, lambda x: x % 2 == 0)
{2: [2, 4, 6, 8], 4: [4, 8, 12, 16]}
```

Resposta:

```
def multiplos_filtrados(n, pred):
    res = {}
    for i in range (1, n+1):
        if pred(i):
            lst = []
            for j in range(1, n+1):
                lst = lst + [i * j]
            res[i] = lst
    return res
```

10. Suponha que quer representar o tipo *tempo*, dividindo-o em horas e minutos. No tipo *tempo* o número de minutos está compreendido entre 0 e 59, e o número de horas apenas está limitado inferiormente a zero. Por exemplo 546:37 é um *tempo* válido.

    Considere as operações básicas do tipo *tempo*:

    - *Construtor*:
      - $cria_tempo : \mathbb{N}_0 \times \mathbb{N}_0 \mapsto tempo$
        $cria_tempo(h, m)$, em que $h \geq 0$ e $0 \leq m \leq 59$ tem como valor o tempo $h : m$.

- *Seletores*:
  - $horas$: *tempo* $\mapsto \mathbb{N}_0$
    $horas(t)$ tem como valor as horas do tempo t.
  - $minutos$: *tempo* $\mapsto \mathbb{N}_0$
    $minutos(t)$ tem como valor os minutos do tempo t.
- *Reconhecedores*:
  - *eh_tempo* : $universal \mapsto$ *lógico*
    eh_tempo(arg) tem o valor $verdadeiro$ apenas se arg é um tempo.

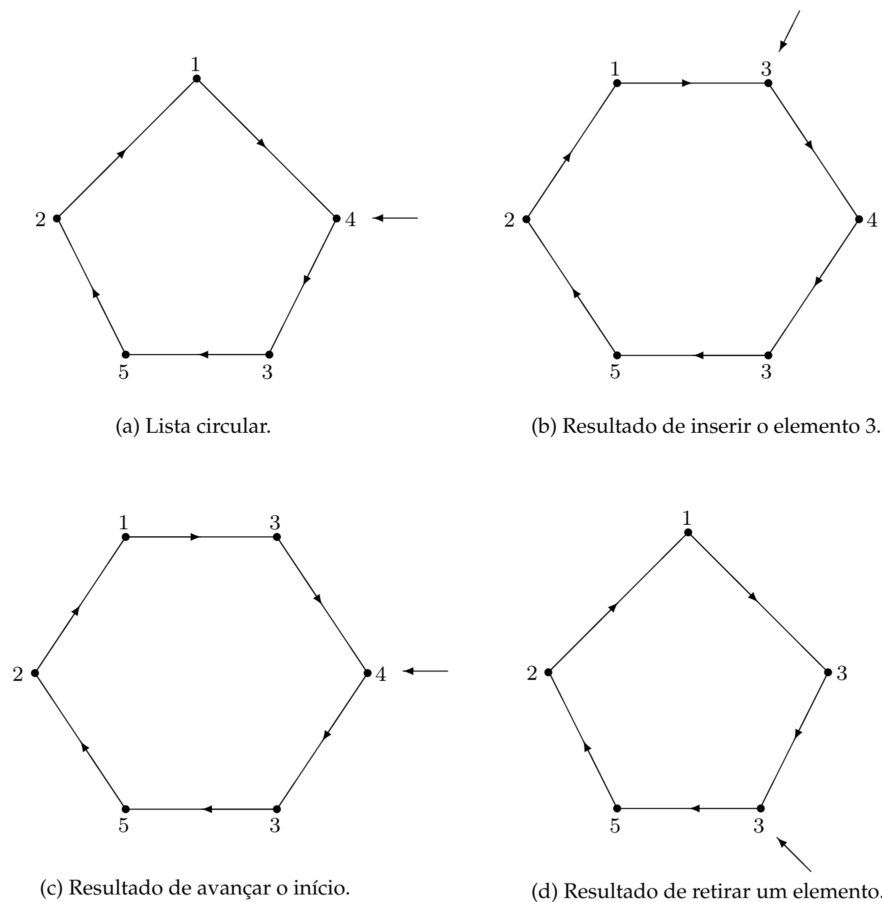
  Teste:
  - $tempos_iguais$: *tempo* $\times$ *tempo* $\mapsto$ *lógico*
    $tempos_iguais(t_1, t_2)$ tem o valor verdadeiro apenas se os tempos t_1 e t_2 são iguais.

(a) **(0.5)** Escolha uma representação para o tipo *tempo*.
Resposta:
Dado que um tempo é uma constante, usamos tuplos para o representar
$Re[h : m]$ = `(h, m)`

(b) **(1.0)** Escreva as operações básicas para a representação escolhida. Apenas o construtor deve verificar a correção dos argumentos.
Resposta:

```
def cria_tempo(h, m):
    if isinstance(h, int) and isinstance(m, int):
        if h >= 0:
            if 0 <= m <= 59:
                return (h, m)
            else:
                raise ValueError('cria_tempo: minutos < 0 ou minutos > 59')
        else:
            raise ValueError ('cria_tempo:  horas negativas')
    else:
        raise ValueError ('cria_tempo:  componentes não inteiros')

def horas(t):
    return t[0]

def minutos(t):
    return t[1]

def eh_tempo(x):
    return isinstance(x, tuple) and len(x) == 2 and \
           isinstance(x[0], int) and isinstance(x[1], int) and \
           x[0] >= 0 and 0 <= x[1] <= 59

def tempos_iguais(t1, t2):
    return t1 == t2
```

(c) **(1.5)** Com base nas operações básicas do tipo *tempo*, escreva o predicado de alto nível
$depois$: *tempo* $\times$ *tempo* $\mapsto$ *lógico*
$depois(t_1, t_2)$ tem o valor $verdadeiro$ apenas se t_1 corresponder a um instante de tempo posterior a t_2. Não é necessário validar a correção dos argumentos.
Resposta:



(a) Lista circular.

(b) Resultado de inserir o elemento 3.

(c) Resultado de avançar o início.

(d) Resultado de retirar um elemento.

Figure 1: Operações sobre listas circulares.

```
def depois(t1, t2):
    return horas(t1)* 60 + minutos(t1) > horas(t2) * 60 + minutos(t2)
```

(d) **(0.5)** Que alterações teria que fazer na função `depois` se fosse alterada a representação interna do tempo? Justifique a sua resposta.

Resposta:

Não teria que fazer qualquer alteração, pois sendo esta uma função de alto nível é independente da representação

11. **(1.5)** O tipo de dados *lista circular* corresponde a uma lista na qual, excepto no caso de ser uma lista vazia, ao último elemento segue-se o primeiro. A Figura 1 (a) mostra esquematicamente uma lista circular em que o primeiro elemento é 4, o segundo, 3, o terceiro, 5, o quarto, 2 e o quinto (e último) é 1. Uma lista circular tem um elemento que se designa por primeiro elemento ou elemento do início da lista. Na lista da Figura 1 (a), esse elemento é 4. As listas circulares aparecem em várias aplicações, sendo uma delas a representação de vídeos que se repetem ao chegar ao fim.

Com listas circulares, para além das operações habituais de um tipo de dados, podemos efetuar as seguintes operações, as quais estão exemplificadas na Figura 1.

- *insere_circ*, que insere um elemento na lista. Com esta operação, o elemento inserido passa a ser o primeiro da lista resultante, o primeiro elemento da lista

original passa a ser o segundo da nova lista, e assim sucessivamente.

- *primeiro_circ*, que devolve o primeiro elemento da lista, sem a alterar.
- *retira_circ*, que retira um elemento da lista. Com esta operação, o elemento retirado é sempre o do início da lista, passando o início da lista resultante a ser o segundo elemento (se este existir) da lista original.
- *avanca_circ*, a qual avança o início da lista para o elemento seguinte. Esta operação não altera os elementos da lista, apenas altera o início da lista, que passa a ser o segundo elemento da lists original, se esta tiver pelo menos dois elementos; se apenas tiver um elemento, nada se altera; se a lista circular for vazia, esta operação tem um valor indefinido.
- *el_n_circ*, a qual recebe um inteiro, n e devolve o elemento na n-ésima posição da lista. Note-se que como a lista é circular, n pode ser superior ao número dos elementos da lista. Considerando a lista da Figura 1(a), o elemento na posição zero é 4 e o elemento na posição sete é 5.

Supondo que a lista circular apresentada na Figura 1(a) é representada externamente por `@4, 3, 5, 2, 1@`, defina a classe `lista_circular` (os seus métodos não necessitam de validar a correção dos argumentos). Permitindo a interação:

```
>>> lc = lista_circ()
>>> lc.insere_circ(1)
@1@
>>> lc.insere_circ(2)
@2, 1@
>>> lc.insere_circ(5)
@5, 2, 1@
>>> lc.insere_circ(3)
@3, 5, 2, 1@
>>> lc.insere_circ(4)
@4, 3, 5, 2, 1@
>>> lc.primeiro_circ()
4
>>> lc.el_n_circ(11)
3
>>> lc.insere_circ(3)
@3, 4, 3, 5, 2, 1@
>>> lc.avanca_circ()
@4, 3, 5, 2, 1, 3@
>>> lc.retira_circ()
@3, 5, 2, 1, 3@
```

Resposta:

```
class lista_circ:

    def __init__(self):
        self.lc = []

    def insere_circ(self, elem):
        self.lc = [elem] + self.lc
```

```
        return self

    def primeiro_circ(self):
        if self.lc != []:
            return self.lc[0]
        else:
            raise ValueError('primeiro_circ: lista vazia')

    def el_n_circ(self, n):
        if self.lc != []:
            return self.lc[n % len(self.lc)]
        else:
            raise ValueError('pos_n_circ: lista vazia')

    def retira_circ(self):
        if self.lc != []:
            self.lc = self.lc[1:]
            return self
        else:
            raise ValueError('retira_circ: lista vazia')

    def avanca_circ(self):
        if self.lc != []:
            self.lc = self.lc[1:] + [self.lc[0]]
            return self
        else:
            raise ValueError('avanca_circ: lista vazia')

    def eh_lista_circ_vazia(self):
        return self.lc == []

    def listas_circ_iguais(self, outro):
        return self.lc == outro.lc_para_lista()

    def lc_para_lista(self):
        return self.lc

    def __repr__(self):
        return '@' + str(self.lc)[1 : -1] + '@'
```