Fundamentos da Programação

TÉCNICO LISBOA

Exame

Solução do exame de 31 de Janeiro de 2020

09:00–11:00

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 procedimental.
   Resposta:
   A utilização de uma função pensando apenas no que esta faz e não como o faz.

   (b) **(0.5)** Abstração de dados.
   Resposta:
   A utilização de dados de um dado tipo considerando apenas as suas características (operações básicas) e não como os dados são representados.

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

   (a) **(0.5)** Algoritmo e programa.
   Resposta:
   Um programa é um algoritmo escrito numa linguagem de programação.

   (b) **(0.5)** Programa e processo computacional.
   Resposta:
   Um processo computacional é uma entidade imaterial que corresponde à execução de um programa.

3. **(1.0)** Um número inteiro, n, diz-se *triangular* se existir um número inteiro m tal que $n = \sum_{i=1}^{m} i$. Escreva o predicado `triangular` que recebe um número inteiro positivo e cujo valor é verdadeiro apenas se o seu argumento for triangular. Não é necessário verificar a validade do argumento. Por exemplo,

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

Resposta:

```
def triangular (n):
    soma = 0
    for i in range(1, n//2+1):
        soma = soma + i
        if soma == n:
            return True
    return False
```

4. **(1.0)** Escreva o predicado `eh_bissexto` que recebe um número inteiro e que tem o valor verdadeiro apenas se o seu argumento for um número bissexto. Um ano é bissexto se for divisível por 4 e não for divisível por 100, a não ser que seja também divisível por 400. Não é necessário verificar a validade do argumento. Por exemplo,

```
>>> eh_bissexto(1100)
False
>>> eh_bissexto(2000)
True
>>> eh_bissexto(2020)
True
```

Resposta:

```
def eh_bissexto(a):
    return a % 4 == 0 and (a % 100 != 0 or a % 400 == 0)
```

5. **(1.0)** Escreva a função `explode` que recebe um número inteiro positivo e devolve o tuplo contendo os dígitos desse número, pela ordem em que aparecem no número. Não é necessário verificar a validade do argumento. Por exemplo,

```
>>> explode(34500)
(3, 4, 5, 0, 0)
```

Resposta:

```
def explode(n):
    res = ()
    while n != 0:
        res = (n % 10, ) + res
        n = n // 10
    return res
```

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

   ⟨S⟩ :: = ⟨A⟩ a

   ⟨A⟩ :: = a ⟨B⟩

   ⟨B⟩ :: = ⟨A⟩ a | b

   (a) **(0.5)** Indique os símbolos terminais e os símbolos não terminais da gramática.
   Símbolos terminais:
   Resposta:
   a, b
   Símbolos não terminais:
   Resposta:
   ⟨S⟩, ⟨A⟩, ⟨B⟩

(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. `aabaa`
Resposta:
Pertence.

ii. `abc`
Resposta:
Não pertence pois `c` não é um símbolo terminal.

iii. `abaa`
Resposta:
Não pertence pois falta um `a` antes do `b`.

iv. `aaaabaaaa`
Resposta:
Pertence.

v. `Aa`
Resposta:
Não pertence pois `A` não é um símbolo terminal.

(c) **(1.0)** Escreva o predicado `reconhece` que recebe uma cadeia de caracteres e cujo valor é verdadeiro apenas se o seu argumento corresponde a uma frase gerada pela gramática. Não é necessário verificar a validade do argumento.
Resposta:

```
def reconhece(frase):
    inicio = 0
    fim = len(frase) - 1
    if len(frase) < 3:
        return False
    while inicio < fim:
        if frase[inicio] == frase[fim] == 'a':
            inicio = inicio + 1
            fim = fim - 1
        else:
            return False
    if (inicio == fim) and (frase[inicio] == 'b'):
        return True
    else:
        return False
```

7. Um número diz-se perfeito se for igual à soma dos seus divisores (não contando o próprio número). Por exemplo, 6 é perfeito porque $1 + 2 + 3 = 6$.

(a) **(1.0)** Escreva o predicado `eh_perfeito` que recebe um inteiro positivo e devolve verdadeiro apenas se o seu argumento corresponder a um número perfeito. Não é necessário verificar a validade do argumento. Por exemplo,

```
>>> eh_perfeito(6)
True
>>> eh_perfeito(10)
False
```

Resposta:

```
def eh_perfeito (n):
    soma = 0
    for i in range(1, n):
```

```
            if n % i == 0:
                soma = soma + i
        return soma == n
```

(b) **(1.0)** Escreva a função, `primeiros_n_perfeitos`, que recebe um inteiro positivo, n, e devolve um tuplo contendo os n primeiros números perfeitos. Utilize o predicado da alínea anterior. Não é necessário verificar a validade do argumento. Por exemplo,

```
>>> primeiros_n_perfeitos(4)
(6, 28, 496, 8128)
```

Resposta:

```
def primeiros_n_perfeitos (n):
    res = ()
    i = 1
    cont_perfeitos = 0
    while cont_perfeitos != n:
        if eh_perfeito(i):
            res = res + (i,)
            cont_perfeitos = cont_perfeitos + 1
        i = i + 1
    return res
```

8. **(1.5)** Escreva a função `junta` que recebe dois dicionários, cujos valores associados às chaves correspondem a listas, e devolve o dicionário que contém todas as chaves contidas em pelo menos um dos dicionários e o valor associado a cada chave corresponde à lista obtida pela "união" (no sentido de conjuntos) das listas correspondendo às chaves existentes nos dicionários. A sua função não pode modificar nenhum dos dicionários recebidos, nem as listas associadas às suas chaves. Não é necessário verificar a validade do argumento. Por exemplo,

```
>>> ml = [3,4]
>>> junta({'a': [1,2], 'b': ml}, {'b': [4,5], 'c': [6,7]})
{'a': [1, 2], 'b': [3, 4, 5], 'c': [6, 7]}
>>> ml
[3, 4]
```

Resposta:

```
def junta(d1, d2):

    res = {}
    # para não destruir d1 nem nenhuma das suas listas
    for c in d1:
        l = []
        for e in d1[c]:
            l = l + [e]
        res[c] = l

    for c in d2:
        if c in res:
            lst = res[c]
            for el in d2[c]:
                if el not in lst:
                    lst = lst + [el]
            res[c] = lst
```

```
        else:
            res[c] = d2[c]
    return res
```

9. Considere a função `num_occ` que recebe uma lista contendo números inteiros e um inteiro e que devolve o número de vezes que o inteiro aparece na lista. Esta função não verifica a validade do argumento. Por exemplo,

```
>>> num_occ([1, 2, 3, 4, 3], 3)
2
>>> num_occ([2, 3, 4, 3], 1)
0
```

Escreva a função `num_occ`:

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

```
def num_occ(lst, n):
    res = 0
    for el in lst:
        if el == n:
            res = res + 1
    return res
```

(b) **(1.0)** Usando recursão com operações adiadas. NOTA: não pode usar atribuição nem ciclos.
Resposta:

```
def num_occ(lst, n):
    if lst == []:
        return 0
    elif lst[0] == n:
        return 1 + num_occ(lst[1:], n)
    else:
        return num_occ(lst[1:], n)
```

(c) **(1.0)** Usando recursão de cauda. NOTA: não pode usar atribuição nem ciclos.
Resposta:

```
def num_occ(lst, n):

    def num_occ_aux(lst, res):
        if lst == []:
            return res
        elif lst[0] == n:
            return num_occ_aux(lst[1:], res + 1)
        else:
            return num_occ_aux(lst[1:], res)

    return num_occ_aux(lst, 0)
```

(d) **(1.0)** Usando um ou mais dos funcionais sobre listas (`filtra`, `transforma`, `acumula`). O corpo da sua função apenas pode ter uma instrução, a instrução `return`.
Resposta:

```
def num_occ(lst, n):
    return len(filtra(lambda x: x==n, lst))
```

10. Suponha que desejava criar o tipo *data*. Uma *data* é caracterizada por um dia (um inteiro entre 1 e 31), um mês (um inteiro entre 1 e 12) e um ano (um inteiro que pode ser positivo, nulo ou negativo). Para cada *data*, deve ser respeitado o limite de dias de cada mês, incluindo o caso de Fevereiro nos anos bissextos.

    O tipo *data* tem as seguintes operações básicas:

    - *Construtores*:
      - $cria\_data : \mathbb{N} \times \mathbb{N} \times \mathbb{Z} \mapsto data$
        $cria\_data(d, m, a)$ tem como valor a data com dia d, mês m e ano a.
    - *Seletores*:
      - $dia : data \mapsto \mathbb{N}$
        $dia(dt)$ tem como valor o dia da data dt.
      - $mes : data \mapsto \mathbb{N}$
        $mes(dt)$ tem como valor o mês da data dt.
      - $ano : data \mapsto \mathbb{Z}$
        $ano(dt)$ tem como valor o ano da data dt.
    - *Reconhecedores*:
      - $eh\_data : universal \mapsto lógico$
        $eh\_data(arg)$ tem o valor $verdadeiro$ se arg é uma *data* e tem o valor $falso$ em caso contrário.
    - *Testes*:
      - $mesma\_data : data^2 \mapsto lógico$
        $mesma\_data(d_1, d_2)$ tem o valor $verdadeiro$ se d_1 e d_2 correspondem à mesma data e tem o valor $falso$ em caso contrário.

    (a) **(0.5)** Escolha uma representação interna para o tipo *data* usando tuplos.
    Resposta:
    $\Re[d/m/a]$ = `(d, m, a)`.

    (b) **(1.5)** Escreva as operações básicas para a representação escolhida. Apenas o construtor deve verificar a validade do argumento. NOTA: Recorra a abstração procedimental para utilizar a função `eh_bissexto` do Exercício 5.
    Resposta:

```
def cria_data(d, m, a):
    if isinstance(d, int) and isinstance(m, int) and isinstance(a, int) \
       and dias_e_meses_corretos(d, m, a):
        return (d, m, a)
    else:
        raise ValueError('cria_data: dados incorretos')

def dias_e_meses_corretos(d, m, a):
    return 1 <= m <= 12 and \
           ((m in (1, 3, 5, 7, 8, 10, 12) and 1 <= d <= 31) or \
            (m in (4, 6, 9, 11) and 1 <= d <= 30) or \
            (m == 2 and eh_bissexto(a) and 1 <= d <= 29) or \
            (m == 2 and not(eh_bissexto(a)) and 1 <= d <= 28))

def dia(data):
```

```
    return data[0]

def mes(data):
    return data[1]

def ano(data):
    return data[2]

def eh_data(arg):
    return isinstance(arg, tuple) and len(arg) == 3 and \
           isinstance(arg[0], int) and isinstance(arg[1], int) and \
           isinstance(arg[2], int) and \
           dias_e_meses_corretos(arg[0], arg[1], arg[2])

def mesma_data(d1, d2):
    return d1 == d2
```

(c) **(0.5)** Supondo que a representação externa para um elemento do tipo *data* é `dd/mm/aaaa ee` (em que `dd` representa o dia, `mm` o mês, `aaaa` o ano e `ee` representa a era, a qual é omitida se o ano for maior ou igual a 0 e é escrita `AC` se o ano for menor que zero), escreva o transformador de saída para o tipo *data*. Por exemplo,

```
>>> escreve_data(cria_data(30, 1, 2020))
'30/01/2020'
>>> escreve_data(cria_data(7, 8, -12))
'07/08/0012 AC'
```

Resposta:

```
def escreve_data(data):

    def n_digitos(n, d):
        return '0'*(d-len(str(n))) + str(n)

    def ac(n):
        if n < 0:
            return ' AC'
        else:
            return ''

    return  n_digitos(dia(data), 2) + '/' + \
            n_digitos(mes(data), 2) + '/' + \
            n_digitos(abs(ano(data)), 4) + ac(ano(data))
```

(d) **(1.0)** Defina a função de alto nível, `dias_entre`, que recebe como argumentos duas *datas* e devolve o número de dias que decorreram entre a primeira (sem contar com a primeira data) e a segunda (contando com a segunda data). Por uma questão de simplificação, considere que as duas datas pertencem ao mesmo ano. Se a primeira *data* for igual à segunda, a função devolve 0; se a primeira *data* for anterior à segunda, a função devolve -1. Não é necessário verificar a validade dos argumentos. Por exemplo:

```
>>> dias_entre(cria_data(1, 1, 2020),cria_data(12, 3, 2020))
71
```

Resposta:

```
def dias_entre(d1, d2):
    """
    Devolve o número de dias entre a data d1 (sem contar com d1)
    e a data posterior d2 (contando com d2)
    """

    def eh_depois(d1, d2):
        '''
        Devolve verdadeiro apenas se a data d2 for
        posterior à data d1
        '''
        if mes(d2) > mes(d1):
            return True
        elif mes(d2) == mes(d1):
            if dia(d2) > dia(d1):
                return True
            else:
                return False
        else:
            return False

    def dias_ate_fim_mes(dt):
        return dias_do_mes(mes(dt), ano(dt)) - dia(dt)

    def dias_entre_meses(dt1, dt2):
        dias = 0
        for m in range(mes(dt1)+1, mes(dt2)):
            dias = dias + dias_do_mes(m, ano(dt2))
        return dias

    def dias_do_mes(m, a):
        '''
        Devolve o número de dias do mês m do ano a
        '''
        if m in (1, 3, 5, 7, 8, 10, 12):
            return 31
        elif m in (4, 6, 9, 11):
            return 30
        else:
            if eh_bissexto(a):
                return 29
            else:
               return 28

    if eh_depois(d1, d2):
        if mes(d2) > mes(d1):
            return dias_ate_fim_mes(d1) + \
                   dias_entre_meses(d1, d2) + dia(d2)
        else:
            return dia(d2) - dia(d1)
    elif mesma_data(d1, d2):
        return 0
    else:
        return -1
```

11. **(2.0)** Defina a classe `reservatorio_com_fuga` que simula o comportamento de

um reservatório de líquidos quem tem uma certa capacidade e que tem uma fuga pela qual perde um certo número de litros por dia.

Por exemplo, suponha que o reservatório tem uma capacidade de 100 litros e que perde líquido à taxa de 0,5 litros por dia. Se o reservatório é cheio no dia 11/1/2020, no dia 13/1/2020 (ou seja passados dois dias), o seu conteúdo será 100 - 2 * 0,5 = 99 litros. As instâncias desta classe são criadas indicando a capacidade do reservatório em litros (o reservatório é criado vazio), quantos litros perde por dia e a data da sua criação. Para lidar com datas use o tipo *data* da pergunta anterior; se não respondeu à pergunta anterior, use abstração procedimental relativamente às operações indicadas na pergunta anterior.

As operações disponíveis são as seguintes:

- $enche : \mathbb{N}_0 \times data \mapsto \mathbb{N}_0$
  $enche(l, d)$, introduz l litros no reservatório na *data* d. Se l mais a quantidade de líquido existente no reservatório na *data* d, o reservatório fica cheio. Esta função *não* verifica a legalidade dos seus argumentos. Devolve a quantidade de litros existentes no reservatório. Se a *data* d for anterior à data em que se efetuou uma operação de `enche` ou de `tira`, esta operação gera uma mensagem e a operação não se realiza.
- $tira : \mathbb{N}_0 \times data \mapsto \mathbb{R}_0$
  $tira(l, d)$, remove l litros no reservatório na *data* d. Se l for superior à quantidade de líquido existente no reservatório na *data* d, o reservatório fica vazio. Esta função *não* verifica a legalidade dos seus argumentos. Devolve a quantidade de litros existentes no reservatório. Se a *data* d for anterior à data em que se efetuou uma operação de `enche` ou de `tira`, esta operação gera uma mensagem e a operação não se realiza.
- $mostra\_nivel : data \mapsto \mathbb{N}_0$
  $mostra\_nivel(d)$, devolve o número de litros existentes no reservatório na *data* d. Esta função *não* verifica a legalidade dos seus argumentos. Se a *data* d for anterior à data em que se efetuou uma operação de `enche` ou de `tira`, esta operação gera uma mensagem e a operação não se realiza.

Por exemplo,

```
>>> res = reservatorio_com_fuga(100, 0.5, cria_data(1, 1, 2020))
>>> res.mostra_nivel(cria_data(10, 1, 2020))
0
>>> res.enche(40, cria_data(11, 1, 2020))
40
>>> res.enche(40, cria_data(2, 1, 2020))
'data anterior à ultima operação: 11/01/2020'
>>> res.mostra_nivel(cria_data(13, 1, 2020))
39.0
>>> res.tira(18, cria_data(19, 1, 2020))
18.0
>>> res.mostra_nivel(cria_data(28, 1, 2020))
13.5
>>> res.mostra_nivel(cria_data(22, 10, 2020))
0
```

Resposta:

```
class reservatorio_com_fuga:

    def __init__(self, capacidade, perda, data):
        self.cap = float(capacidade)
        self.prd = perda
        self.nivel = 0
        self.ult_mod = data

    def muda_nivel(self, niv_inic, dias, perda, variacao):
        nivel_atual = niv_inic - dias * perda
        if nivel_atual < 0:
            nivel_atual = 0
        return nivel_atual + variacao

    def enche(self, vol, d):
        tempo_passado = dias_entre(self.ult_mod, d)
        if tempo_passado >= 0:
            self.ult_mod = d
        else:
            return 'data anterior à ultima operação: ' + \
                   str(escreve_data(self.ult_mod))
        self.nivel = self.muda_nivel(self.nivel, tempo_passado, \
                                     self.prd, vol)
        if self.nivel > self.cap:
            self.nivel = self.cap
        return self.nivel

    def tira(self, vol, d):
        tempo_passado = dias_entre(self.ult_mod, d)
        if tempo_passado >= 0:
            self.ult_mod = d
        else:
            return 'data anterior à ultima operação: ' + \
                   str(escreve_data(self.ult_mod))
        self.nivel = self.muda_nivel(self.nivel, tempo_passado, \
                                           self.prd, -vol)
        if self.nivel < 0:
            self.nivel = 0
        return self.nivel

    def mostra_nivel(self, d):
        if self.nivel == 0:
            return 0
        else:
            tempo_passado = dias_entre(self.ult_mod, d)
            nivel_atual = self.nivel - self.prd * tempo_passado
            if nivel_atual < 0:
                nivel_atual = 0
            return nivel_atual

    def __repr__(self):
        return str(self.cap) + ' ' + str(self.prd) + ' ' + \
               str(self.nivel) + ' ' + str(self.ult_mod)
```

(continuação da resposta da pergunta 12)

RASCUNHO