Fundamentos da Programação

Solução do Exame

13 de Janeiro de 2017

09:00–11:00

1. Indique se cada uma das seguintes afirmações é verdadeira ou falsa. No caso de ser falsa, justifique de forma sucinta.

   (a) **(0.5)** Um algoritmo é uma sequência finita de instruções bem definidas e não ambíguas, cada uma das quais pode ser executada mecanicamente num período de tempo finito com uma quantidade de esforço finita.
   Resposta:
   Verdadeira.

   (b) **(0.5)** Um processo computacional dita a execução de um programa.
   Resposta:
   Falsa. O programa é que dita a evolução do processo computacional.

   (c) **(0.5)** No uso de tipos abstractos de dados (TADs) o programador não pode conhecer a representação interna dos elementos do tipo.
   Resposta:
   Falsa, o programador pode conhecer a representação interna, mas não a pode utilizar.

2. **(1.0)** Escreva uma função em Python com o nome `conta_menores` que recebe um tuplo contendo números inteiros e um número inteiro e que devolve o número de elementos do tuplo que são menores do que esse inteiro. Não é necessário verificar a validade dos argumentos. Por exemplo,

```
>>> conta_menores((3, 4, 5, 6, 7), 5)
2
>>> conta_menores((3, 4, 5, 6, 7), 2)
0
```

Resposta:

```
def conta_menores(t, num):
    res = 0
    for e in t:
        if e < num:
            res = res +1
    return res
```

3. **(1.5)** Escreva uma função com o nome `soma_divisores` que recebe um número inteiro positivo `n`, e tem como valor a soma de todos os divisores de `n`. No caso de `n` ser 0 deverá devolver 0. Por exemplo,

```
>>> soma_divisores(20)
42
>>> soma_divisores(13)
14
```

Resposta:

```
def soma_divisores(n):
    res = 0
    for i in range(1, n + 1):
        if n % i == 0:
            res = res + i
    return res
```

4. Considere a seguinte gramática em notação BNF, cujo símbolo inicial é "S"

$\langle S \rangle$ ::= $\langle A \rangle$ a

$\langle A \rangle$ ::= a $\langle B \rangle$

$\langle B \rangle$ ::= $\langle A \rangle$ a | b

(a) **(0.5)** Diga quais são os símbolos terminais e quais são os símbolos não terminais da gramática.

Símbolos terminais:

Resposta:

a, b

Símbolos não terminais:

Resposta:

$\langle S \rangle$, $\langle A \rangle$, $\langle B \rangle$

(b) **(0.5)** Descreva informalmente as frases que pertencem à linguagem.

Resposta:

As frases da linguagem começam com um número arbitrários de `a`s (mas pelo menos um), seguidas por um `b`, seguidas pelo mesmo número de `a`s que aparecem antes do `b`.

(c) **(1.5)** Escreva a função `reconhece` que recebe uma cadeia de caracteres e devolve `True` se o seu argumento corresponde a uma frase gerada pela gramática e `False` em caso contrário. A sua função deve verificar a correcção do argumento.

Resposta:

```
def reconhece(frase):
    if isinstance (frase, str):
        i, nas = 0, 0
        while i < len(frase) and frase[i] == 'a':
            nas = nas + 1
            i = i + 1
        if nas == 0:
            return False
```

```
            else:
                return frase == 'a' * nas + 'b' + 'a' * nas
        else:
            return False
```

5. **(1.5)** Escreva a função `cria_lista_multiplos` que recebe um número inteiro positivo, e devolve uma lista com os dez primeiros múltiplos desse número. A sua função deve testar se o seu argumento é um número inteiro positivo e dar uma mensagem de erro adequada em caso contrário. Considere que zero é múltiplo de todos os números. Por exemplo,

```
>>> cria_lista_multiplos(6)
[0, 6, 12, 18, 24, 30, 36, 42, 48, 54]
```

Resposta:

```
def cria_lista_multiplos(n):
    res = []
    for i in range(10):
        res = res + [i*n]
    return res
```

6. **(1.5)** Escreva a função recursiva `parte` que recebe uma lista de números e um número e que devolve uma lista de duas listas, a primeira lista contém os elementos da lista original menores que o número dado (pela mesma ordem) e a segunda lista contém os elementos da lista original maiores ou iguais que o número dado (pela mesma ordem). Não é necessário verificar a correcção dos dados de entrada. **Sugestão**: Use uma função auxiliar. Por exemplo,

```
>>> parte([3, 5, 1, 4, 5, 8, 9], 4)
[[3, 1], [5, 4, 5, 8, 9]]
```

Resposta:

```
def parte (lst, el):

    def parte_aux(lst, el, menores, maiores):
        if lst == []:
            return [menores, maiores]
        elif lst[0] < el:
            return parte_aux(lst[1:], el, menores + [lst[0]], maiores)
        else:
            return parte_aux(lst[1:], el, menores, maiores + [lst[0]])

    return parte_aux(lst, el, [], [])
```

7. **(1.0)** Escreva a função `max_div` que recebe dois inteiros positivos, `n` e `d`, e que devolve o inteiro resultante de dividir `n` pela maior potência de `d` pela qual `n` é divisível. Não é necessário verificar a correcção dos argumentos. Por exemplo,

```
>>> max_div(300, 2)
75 # a maior potência de 2 é 4
>>> max_div(75, 3)
```

```
25 # a maior potência de 3 é 3
>>> max_div(75, 2)
75 # 75 não é divisível por nenhuma potência de 2
```

Resposta:

```
def max_div(n, d):
    while n % d == 0:
        n = n//d
    return n
```

8. Os números "feios" são aqueles em que os únicos divisores primos são 2, 3 ou 5. Por convenção, o número 1 é feio. Os 10 primeiros números feios são 1, 2, 3, 4, 5, 6, 8, 9, 10 e 12. Para verificar se um número é feio, divide-se sucessivamente o número pelas maiores potências possíveis de 2, 3 e 5. Se o resultado for 1, o número é feio. Seguindo o exemplo da pergunta anterior, 300 é feio.

   (a) **(1.0)** Recorrendo à função `max_div` da pergunta anterior, escreva a função, `e_feio` que recebe um inteiro positivo, `n`, e devolve `True` se `n` for feio e `False` em caso contrário. Não é necessário verificar a correcção do argumento. Por exemplo,

   ```
   >>> e_feio(5)
   True
   >>> e_feio(11)
   False
   ```

   Resposta:

   ```
   def e_feio(n):
       for d in (2, 3, 5):
           n = max_div(n, d)
       return n == 1
   ```

   (b) **(1.0)** Recorrendo à função `e_feio` da alínea anterior, escreva a função `n_esimo_feio` que recebe um inteiro positivo, `n`, e que devolve o `n`-ésimo número feio. Não é necessário verificar a correcção do argumento. Por exemplo,

   ```
   >>> n_esimo_feio(1)
   1
   >>> n_esimo_feio(10)
   12
   ```

   Resposta:

   ```
   def n_esimo_feio(n):
       i = 1
       cont = 1
       while cont < n:
           i = i + 1
           if e_feio(i):
               cont = cont + 1
       return i
   ```

9. **(1.0)** Utilizando alguns (ou todos) os funcionais sobre listas (`filtra`, `transforma`, `acumula`) escreva uma função que devolve a soma dos quadrados dos elementos de uma lista. O corpo da sua função apenas pode ter uma instrução `return`.

   Resposta:

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

10. **(1.0)** Escreva uma função em Python que recebe um dicionário cujos valores associados às chaves correspondem a listas de inteiros e que devolve o dicionário que se obtém "invertendo" o dicionário recebido, no qual as chaves são os inteiros que correspondem aos valores do dicionário original e os valores são as chaves do dicionário original às quais os valores estão associados. Por exemplo,

```
>>> inverte_dic({'a': [1, 2], 'b': [1, 5], 'c': [9], 'd': [4]})
{1: ['a', 'b'], 2: ['a'], 4: ['d'], 5: ['b'], 9: ['c']}
```

Resposta:

```
def inverte_dic(d):
    res = {}
    for e in d:
        for v in d[e]:
            if v in res:
                res[v] = res[v] + [e]
            else:
                res[v] = [e]
    return res
```

11. Suponha que quer representar o 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.

As operações básicas do tipo *tempo* são as seguintes:

- *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 em Python as operações básicas, de acordo com a representação escolhida.

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('os minutos devem estar compreendidos entre
        else:
            raise ValueError ('as horas devem ser um número positivo')
    else:
        raise ValueError ('os componentes do tempo devem ser 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

def escreve_tempo(t):
    h = str(t[0])
    if t[1] < 10:
        m = '0' + str(t[1])
    else:
        m = str(t[1])
    print(h + ':' + m)
```

(c) **(1.0)** Com base nas operações básicas do tipo *tempo*, escreva a função 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.

Resposta:

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

12. Considere a seguinte função definida para inteiros não negativos:

$$f(n) = \begin{cases} n & \text{se } 0 \leq n < 3 \\ f(n-1) + 2 \cdot f(n-2) + 3 \cdot f(n-3) & \text{se } n \geq 3 \end{cases}$$

(a) **(1.0)** Escreva em Python uma função recursiva que calcula o valor desta função. A sua função deve testar, de modo eficiente, a correcção do argumento. Por exemplo,

```
>>> f(20)
10771211
>>> f(-2)
ValueError: f: o argumento deve ser um inteiro não negativo
```

Resposta:

```
def f(n):

    def f_aux(n):
        if n < 3:
            return n
        else:
            return f_aux(n-1) + 2 * f_aux(n-2) + 3 * f_aux(n-3)

    if isinstance(n, int) and n >= 0:
        return f_aux(n)
    else:
        raise ValueError('f: o argumento deve ser ' + \
                         'um inteiro não negativo')
```

(b) **(0.5)** Diga, justificando, qual o tipo de processo gerado por esta função.
Resposta:
É um processo recursivo em árvore pois a versão chama três versões de si próprio.

(c) **(1.5)** Como pode verificar, a sua função calcula várias vezes o mesmo valor quando chamada com um argumento superior a 3. Para evitar este problema, podemos definir uma classe, `mem_f`, cujo estado interno contém informação sobre os valores de `f` já calculados, apenas calculando um novo valor quando este ainda não é conhecido. Esta classe possui um método `val` que calcula o valor de `f` para o inteiro que é seu argumento e um método `mem` que mostra os valores memorizados. Por exemplo,

```
>>> fm = mem_f()
>>> fm.val(8)
1082
>>> fm.mem()
{0: 0, 1: 1, 2: 2, 3: 7, 4: 17, 5: 52, 6: 137, 7: 397, 8: 1082}
```

Resposta:

```
class mem_f:

    def __init__(self):
        self.f = {0: 0, 1: 1, 2: 2}

    def val(self, n):

        def val_aux(n):
            if n in self.f:
                return self.f[n]
            else:
                f_n_1 = val_aux(n-1)
                f_n_2 = val_aux(n-2)
                f_n_3 = val_aux(n-3)
                self.f[n] = f_n_1 + 2 * f_n_2 + 3 * f_n_2
                return self.f[n]
```

```
        if isinstance(n, int) and n >= 0:
            return val_aux(n)
        else:
            raise ValueError('f: o argumento deve ser' + \
                             'um inteiro não negativo')

    def mem(self):
        return self.f
```