TÉCNICO LISBOA

Fundamentos da Programação

Exame

9 de Janeiro de 2015

09:00–11:00

1. De um modo sucinto, explique o que é:

   (a) **(0.5)** Um processo computacional.
   Resposta:
   Um ente imaterial que existe dentro de um computador durante a execução de um programa, e cuja evolução ao longo do tempo é ditada pelo programa.

   (b) **(0.5)** Um algoritmo.
   Resposta:
   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.

   (c) **(0.5)** A abstracção procedimental.
   Resposta:
   A abstracção procedimental corresponde a abstrair do modo como uma função realiza o seu trabalho, considerando apenas o que ela faz.

2. O Python permite a utilização de três paradigmas de programação, a programação imperativa, a programação por objectos e a programação funcional. Apresente de uma forma sucinta as características principais de cada um destes paradigmas.

   (a) **(0.5)** Programação imperativa.
   Resposta:
   Em programação imperativa, um programa é constituido por uma série de ordens dadas ao computador. A programação imperativa depende da instrução de atribuição e da utilização de ciclos.

   (b) **(0.5)** Programação por objectos.
   Resposta:
   A programação por objectos baseia-se na utilização de objectos, entidades com estado interno associados a um conjunto de métodos que manipulam esse estado.

   (c) **(0.5)** Programação funcional.
   Resposta:
   A programação funcional baseia-se na utilização de funções que devolvem valores que são utilizados por outras funções. Em programação funcional as operações de atribuição e os ciclos podem não existir.

3. **(1.5)** Escreva em Python a função `calc_soma` que recebe um número real e um número inteiro, `x` e `n`, e que calcula o valor da soma.

$$1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!}$$

A sua função deve ter em atenção que o i-ésimo termo da soma pode ser obtido do termo na posição $i-1$, multiplicando-o por x/i. Não é necessário validar os valores dos argumentos. Por exemplo:

```
>>> calc_soma(2, 3)
6.333333333333333
```

Resposta:

```
def calc_soma(x, n):
    soma = 1
    termo = 1
    i = 1
    while i <= n:
        termo = termo * (x / i)
        soma = soma + termo
        i = i + 1
    return soma
```

4. **(1.5)** Escreva em Python a função `reduzir_por_chave` que recebe uma lista de pares (chave, valor) e retorna a lista com a soma dos valores associados a cada chave. A sua função não tem de verificar a correcção do argumento. Por exemplo, Por exemplo, se receber a lista `[('abc', 3), ('xxx', 5), ('abc', 10), ('xxx', 20)]` deve devolver a lista `[('xxx', 25), ('abc', 13)]`.

Resposta:

```
def reduzir_por_chave(lst):
   dict = {}
   for elem in lst:
      k = elem[0]
      if k in dict:
         dict[k] = dict[k]+elem[1]
      else:
         dict[k] = elem[1]

   res = []
   for i in dict:
      res = res + [(i,dict[i])]

   return(res)
```

5. **(1.5)** 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 validar o valor do argumento. Por exemplo,

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

Resposta:

```
def triangular(n):
    soma = 1
    i = 2
    while soma <= n:
        if soma == n:
            return True
        soma = soma + i
        i = i + 1
    return False
```

6. Um número *primo* é um número inteiro maior do que 1 que apenas é divisível por 1 e por si próprio. Por exemplo, 5 é primo porque apenas é divisível por si próprio e por um, ao passo que 6 não é primo pois é divisível por 1, 2, 3, e 6.

   (a) **(1.5)** Um método simples, mas pouco eficiente, para determinar se um número, n, é primo consiste em testar se n é múltiplo de algum número entre 2 e $\sqrt{n}$. Usando este processo, escreva uma função em Python chamada `primo` que recebe um número inteiro e tem o valor `True` apenas se o seu argumento for primo. Não é necessário validar o valor do argumento. Suponha que a função `sqrt` já se encontra definida. Por exemplo,

   ```
   >>> primo(23)
   True
   >>> primo(20)
   False
   ```

   Resposta:

   ```
   def primo(n):
       if n in (0, 1):
           return False
       i = 2
       raiz = sqrt(n)
       while i <= raiz:
           if n % i == 0:
               return False
           i = i + 1
       return True
   ```

   (b) **(1.5)** Um número n é o *n-ésimo primo* se for primo e existirem $n - 1$ números primos menores que ele. Usando a função `primo` do exercício anterior, escreva uma função com o nome `n_esimo_primo` que recebe como argumento um número inteiro, `n`, e devolve o `n`-ésimo número primo. Não é necessário validar o valor do argumento. Por exemplo,

   ```
   >>> n_esimo_primo(3)
   5
   ```

   Resposta:

```
def n_esimo_primo(n):
    cont = 0
    num = 1
    while cont != n:
        num = num + 1
        if primo(num):
            cont = cont + 1
    return num
```

7. Considere a seguinte gramática em notação BNF:

   ⟨frase⟩ ::= `c` ⟨meio⟩ `r`

   ⟨meio⟩ ::= ⟨letra⟩$^+$

   ⟨letra⟩ ::= `a` | `d`

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

   Símbolos terminais:
   Resposta:
   `c`, `a`, `d`, `r`
   Símbolos não terminais:
   Resposta:
   ⟨frase⟩, ⟨meio⟩, ⟨letra⟩

   (b) **(0.5)** Descreva a linguagem definida por esta gramática.
   Resposta:
   As frases começam pela letra `c`, terminam na letra `r` e entre estas dias letras podem ter qualquer número das letras `a` e `d`, existindo pelo menos uma destas duas letras.

   (c) **(1.5)** Escreva uma função em Python, chamada `reconhece`, que recebe como argumento uma cadeia de carateres e devolve *verdadeiro* se o seu argumento corresponde a uma frase da linguagem definida pela gramática e *falso* em caso contrário. Por exemplo,

```
reconhece('cdr')
True
reconhece('cdaaaaaaddddddr')
True
reconhece('cdaaaaaaddddddra')
False
```

   Resposta:

```
def reconhece(frase):
    nbchars = len(frase)
    if frase[0] != 'c' or frase[-1] != 'r' or nbchars < 3:
        return False
    for i in range(1, nbchars-1):
        if frase[i] not in ('d', 'a'):
            return False
    return True
```

8. **(1.5)** Escreva em Python 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 validar os valores 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]
```

9. **(1.5)** Escreva uma função *recursiva*, `num_oc_lista`, que recebe uma lista e um inteiro, e devolve o número de vezes que esse inteiro ocorre na lista e nas suas sublistas, se existirem. A sua função não tem de verificar a correcção dos argumentos. Por exemplo,

```
>>> num_oc_lista([4, 5, 6],  5)
1
>>> num_oc_lista([3, [[3], 5], 7, 3, [2, 3]], 3)
4
```

Resposta:

```
def num_oc_lista(lst, num):
    if lst == []:
        return 0
    elif lst[0] == num:
        return 1 + num_oc_lista(lst[1:], num)
    elif isinstance(lst[0], list):
        return (num_oc_lista(lst[0], num) +
                num_oc_lista(lst[1:], num))
    else:
        return num_oc_lista(lst[1:], num)
```

10. **(1.5)** Usando recursão, escreva a função `num_divisores` que recebe um número inteiro positivo n, e devolve o número de divisores de n. No caso de n ser 0 deverá devolver 0. Não é necessário validar o valor do argumento. Por exemplo,

```
>>> num_divisores(20)
6
>>> num_divisores(13)
2
```

Resposta:

```
def num_divisores_rec(n):

    def num_div_aux(i):
        if i > n:
            return 0
        elif n % i == 0:
            return 1 + num_div_aux(i + 1)
        else:
            return num_div_aux(i + 1)

    return num_div_aux(1)
```

11. Suponha que desejava criar o tipo vetor em Python. Um vetor num referencial cartesiano pode ser representado pelas coordenadas da sua extremidade (x, y), estando a sua origem no ponto $(0, 0)$. Podemos considerar as seguintes operações básicas para vetores:

    - *Construtor*:
      $vetor : real \times real \mapsto vetor$
      $vetor(x, y)$ tem como valor o vetor cuja extremidade é o ponto (x, y).
    - *Seletores*:
      $abcissa : vetor \mapsto real$
      $abcissa(v)$ tem como valor a abcissa da extremidade do vetor v.
      $ordenada : vetor \mapsto real$
      $ordenada(v)$ tem como valor a ordenada da extremidade do vetor v.
    - *Reconhecedores*:
      $eh\_vetor : universal \mapsto lógico$
      $eh\_vetor(arg)$ tem valor $verdadeiro$ apenas se arg é um vetor.
      $eh\_vetor\_nulo : vetor \mapsto lógico$
      $eh\_vetor\_nulo(v)$ tem valor $verdadeiro$ apenas se v é o vetor (0, 0).
    - *Teste*:
      $vetores\_iguais : vetor \times vetor \mapsto lógico$
      $vetores\_iguais(v_1, v_2)$ tem valor verdadeiro apenas se os vetores v_1 e v_2 são iguais.

    (a) **(0.5)** Defina uma representação para vetores utilizando dicionários.
    Resposta:
    $\Re[(x, y)] =$ `{'x': x, 'y': y}`

(b) **(1.5)** Escreva em Python as operações básicas, de acordo com a representação escolhida.
Resposta:

```
def vetor(x, y):
    return {'x': x, 'y': y}

def abcissa(v):
    return v['x']

def ordenada(v):
    return v['y']

def eh_vetor(x):
    if isinstance(x, dict) and len(x) == 2 and \
       'x' in x and 'y' in x and \
       isinstance(x['x'],float) and isinstance(x['y'],float):
        return True
    else:
        return False

def eh_vector_nulo(v):
    return v['x'] == 0 and v['y'] == 0

def vectores_iguais(v1, v2):
    return v1['x'] == v2['x'] and v1['y'] == v2['y']
```

(c) **(0.5)** Usando as operações básicas, escreva uma função para determinar se um vetor é colinear com o eixo dos xx.
Resposta:

```
def colinear_x(v):
    return ordenada(v) == 0
```