TÉCNICO LISBOA

Fundamentos da Programação

Solução do Primeiro Teste

27 de Outubro de 2012

09:00–10:30

1. **(1.0)** O que é um processo computacional? Qual a relação entre um programa e um processo computacional?

   Resposta:

   Um processo computacional é 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.

2. A abstracção procedimental foi descrita como um mecanismo utilizado para dominar a complexidade de programas.

   (a) **(0.7)** Diga em que consiste a abstracção procedimental.
   Resposta:
   A abstracção procedimental corresponde a abstrair o modo como uma função realiza o seu trabalho, considerando apenas o que ela faz.

   (b) **(0.8)** Explique como esta pode ser usada no controle da complexidade de programas.
   Resposta:
   Ao desenvolver um programa, identificam-se os principais problemas que este tem que resolver, especificando-se funções que realizam esse trabalho e sem entrar nos detalhes do modo como elas realizam o seu trabalho. Depois de escrita uma primeira versão do programa recorrendo à abstracção procedimental, aborda-se o desenvolvimento de cada uma das funções especificadas utilizando o mesmo método.

   (c) **(0.5)** Como é que a abstracção procedimental é realizada em Python?
   Resposta:
   Através da definição de funções que recebem os argumentos apropriados.

3. Os métodos de passagem de parâmetros correspondem a modos de associar os parâmetros concretos com os parâmetros formais.

   (a) **(0.6)** Diga o que são os parâmetros formais e o que são os parâmetros concretos.
   Resposta:
   Os parâmetros formais são os argumentos especificados na definição de uma função e os parâmetros concretos são os valores que são usados na invocação de uma função.

(b) **(0.7)** Explique o funcionamento da passagem por valor.
Resposta:
Na passagem por valor, o parâmetro concreto é avaliado e o seu valor é associado com o respectivo parâmetro formal. A passagem por valor é um mecanismo unidireccional, do ponto de chamada para a função.

(c) **(0.7)** Explique o funcionamento da passagem por referência.
Resposta:
Na passagem por referência a localização de memória da entidade correspondente ao parâmetro concreto é fornecida ao parâmetro formal. Na passagem por referência o parâmetro concreto e o parâmetro formal partilham a mesma entidade na memória do computador.

4. **(1.0)** Escreva uma gramática em notação BNF que gera frases constituídas pelos símbolos `c`, `a`, `r`, `d`. As frases da linguagem começam pelo símbolo `c`, o qual é seguido por uma ou mais ocorrências dos símbolos `a` e `d`, e terminam no símbolo `r`. Por exemplo `caaddaar` e `cdr` são frases da linguagem, `cd` e `cdrr` não o são.

Resposta:

⟨frase⟩ ::= `c` ⟨meio⟩ `r`

⟨meio⟩ ::= ⟨letra⟩$^+$

⟨letra⟩ ::= `a` | `d`

5. **(1.0)** Considere o seguinte programa em Python:

```
numero_1 = 5
numero_2 = 10
while numero_1 > 0:
    numero_2 = numero_2 - numero_1
```

Será que este programa pode ser considerado um algoritmo? Justifique a sua resposta.

Resposta:

Este programa não pode ser considerado um algoritmo porque nunca termina. Na realidade a condição `numero_1 > 0` é sempre verdadeira, dado que `numero_1` tem o valor `5` e o corpo do ciclo `while` não altera este valor.

6. **(1.0)** Escreva um programa em Python que pede ao utilizador que lhe forneça um inteiro correspondente a um certo número de horas e que escreve um número real que traduz o número de dias correspondentes ao inteiro lido. O seu programa deve gerar uma interacção como a seguinte:

```
Eu converto horas para dias
Escreva as horas
Horas? 3465
3465 horas correspondem a  144.375 dias
```

Resposta:

```
print('Eu converto horas para dias')
h = eval(input('Escreva as horas\nHoras? '))
print(h, 'horas correspondem a', h / 24, 'dias')
```

7. **(1.0)** Diga o que é escrito pela seguinte instrução:

```
for i in range(2):
    for j in range(3, 5):
        for k in range(4, 1, -1):
            if (i + j) % 2 == 0:
                print(i, j, k)
```

Resposta:

```
0 4 4
0 4 3
0 4 2
1 3 4
1 3 3
1 3 2
```

8. **(1.0)** Escreva uma função em Python com o nome `posicoes_tuplo` que recebe um tuplo de números e um número, e devolve o tuplo de todas as posições em que o número ocorre no tuplo. Por exemplo,

```
>>> posicoes_tuplo((4, 3, 2, 2, 1, 4), 4)
(0, 5)
>>> posicoes_tuplo((4, 3, 2, 2, 1, 4), 6)
()
```

Resposta:

```
def posicoes_tuplo(t, n):
    pos = ()
    for i in range(len(t)):
        if t[i] == n:
            pos = pos + (i, )
    return pos
```

9. **(1.0)** Escreva uma função em Python com o nome `soma_quadrados` que recebe um número inteiro positivo, `n`, e tem como valor a soma dos quadrados de todos os números inteiros de 1 até `n`. Utilize um ciclo `while`.

Resposta:

```
def soma_quadrados(n):
    soma = 0
    while n != 0:
        soma = soma + n * n
        n = n - 1
    return soma
```

10. **(1.5)** Escreva uma função em Python com o nome `numero_algarismos` que recebe um inteiro positivo, `n`, e devolve o número de algarismos de `n`. Por exemplo:

```
>>> numero_algarismos(0)
1
>>> numero_algarismos(23464321)
8
```

Resposta:

```
def numero_algarismos(num):
    algarismos = 1 # Um número tem pelo menos 1 algarismo
    while num > 9:
        algarismos = algarismos + 1
        num = num // 10
    return algarismos
```

11. **(1.5)** Suponha que a operação de multiplicação não existia em Python, existindo as operações de adição, +, de subtracção, −, e o operador relacional >. Escreva em Python uma função com o nome `vezes`, que efectua a multiplicação de dois inteiros positivos. O seu programa não tem que verificar se os inteiros são positivos. Pode usar instruções correspondentes a ciclos.

Resposta:

```
def vezes(m, n):
    res = 0
    while n > 0:
        res = res + m
        n = n - 1
    return res
```

12. **(2.0)** Escreva uma função em Python com o nome `remove_pares` que recebe uma lista não vazia de números, e altera essa lista de modo a remover todos os números pares que esta contém. A sua função deve testar se o argumento é uma lista não vazia de números e produzir uma mensagem de erro adequada em caso contrário. Por exemplo,

```
>>> lst = [3, 5, 6, 2, 7, 10, 2]
>>> remove_pares(lst)
>>> lst
[3, 5, 7]
>>> remove_pares([])
builtins.ValueError: remove_pares: arg não é lista de números
```

Resposta:

```
def remove_pares(lst):
    if not(e_lista_numeros(lst)):
        raise ValueError('remove_pares: arg não é lista de números')
    else:
        for i in range(len(lst), 0, -1):
            if lst[i-1] % 2 == 0:
                del(lst[i-1])

def e_lista_numeros(lst):
    if not(isinstance(lst, list)) or lst == []:
        return False
    else:
        for el in lst:
            if not(isinstance(el, (int, float))):
                return False
    return True
```

13. (a) **(3.0)** Utilizando a representação de caracteres, escreva funções em Python, com os nomes `codifica` e `descodifica`, que recebem uma mensagem (uma cadeia de caracteres) e, respectivamente, codificam e descodificam essa mensagem, utilizando a cifra de Atbash. Uma mensagem é codificada através da cifra de Atbash, substituindo cada uma das suas letras pela letra que se encontra à mesma distância do fim do alfabeto que a distância da letra original ao início do alfabeto. Com a cifra de Atbash originamos a seguinte correspondência entre as letras do alfabeto:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ZYXWVUTSRQPONMLKJIHGFEDCBA
```

A sua cifra apenas considera letras maiúsculas, sendo qualquer símbolo que não corresponda a uma letra maiúscula ignorado. Os espaços entre as palavras mantêm-se como espaços. A seguinte interacção ilustra o funcionamento destas funções:

```
>>> codifica('ESTA E UMA PERGUNTA BOA PARA O TESTE DE FP')
'VHGZ V FNZ KVITFMGZ YLZ KZIZ L GVHGV WV UK'
>>> descodifica('VHGZ V FNZ KVITFMGZ YLZ KZIZ L GVHGV WV UK')
'ESTA E UMA PERGUNTA BOA PARA O TESTE DE FP'
```

Resposta:

```
def codifica(f):
    codif = ''
    for c in f:
        if c == ' ':
           codif = codif + c
        elif 'A' <= c <= 'Z':
           codif = codif + \
                   chr(ord('Z') - (ord(c) - ord('A')))
    return codif

def descodifica(f):
    desc = ''
    for c in f:
        if c == ' ':
           desc = desc + c
        elif 'A' <= c <= 'Z':
           desc = desc + \
                  chr(ord('A') + (ord('Z') - ord(c)))
    return desc
```

Ou, alternativamente, considerando que o `codifica` e o `descodifica` na realidade realizam a mesma operação, podemos simplificar o `descodifica`.

```
def descodifica(f):
    return codifica(f)
```

(b) **(1.0)** Escreva um programa em Python que utiliza as funções definidas na alínea anterior, interaccionando com este através da solicitação de frases para codificar ou descodificar. Parta do princípio que as funções `codifica` e `descodifica` da alínea anterior já estão definidas. A seguinte interacção ilustra o funcionamento do seu programa:

```
>>> codificador()
```

```
Introduza uma mensagem
Use a palavra FIM para terminar
-> AQUI VAI UM EXEMPLO
C(odificar), D(escodificar)?
-> C
A mensagem codificada é:
 ZJFR EZR FN VCVNKOL
Introduza uma mensagem
Use a palavra FIM para terminar
-> FIM
```

Resposta:

```
def codificador():

    print('Introduza uma mensagem\n', \
          'Use a palavra FIM para terminar')
    original = input('-> ')

    while original != 'FIM':
        tipo = input('C(odificar), D(escodificar)?\n-> ')
        if tipo == 'C' :
             print('A mensagem codificada é:\n',
                   codifica(original))
        elif tipo == 'D' :
             print('A mensagem descodificada é:\n',
                   descodifica(original))
        else:
             print('Oops .. não sei o que fazer')
        print('Introduza uma mensagem\n', \
              'Use a palavra FIM para terminar')
        original = input('-> ')
```