TÉCNICO LISBOA

Fundamentos da Programação

Solução do Segundo Teste

18 de Janeiro de 2013

09:00–10:30

1. **(2.0)** Escolha a *única* resposta *incorrecta* para as seguintes questões. Cada resposta certa vale 1 valor e *cada resposta errada desconta 0.4 valores*.

   (a) A seguinte afirmação define uma característica essencial de uma função de ordem superior:

   A. É uma função cujo valor é outra função.
   B. É um objecto computacional que é considerado um cidadão de primeira classe.
   C. É uma função que recebe outras funções como parâmetros.
   D. É uma função com estado interno.

   Resposta:

   D.

   (b) Em relação à metodologia de desenvolvimento de programas pode dizer-se que:

   A. Na fase da programação da solução decide-se a representação para as estruturas de informação.
   B. A depuração lida com erros semânticos e erros sintácticos.
   C. A fase de desenvolvimento da solução recorre a uma linguagem de programação.
   D. A fase de testes nunca permite garantir a ausência completa de erros.

   Resposta:

   C.

2. **(1.0)** Explique em que consiste a abstracção de dados, usando os termos barreiras de abstracção, encapsulação da informação e anonimato da representação.

   Resposta:

   A abstracção de dados consiste em separar o modo como os dados são utilizados do modo como os dados são representados. Para isso definem-se camadas conceptuais lidando com cada um destes aspectos estando estas camadas separadas por "barreiras de abstracção" que definem o modo como os programas acima da barreira podem comunicar com os programas que se encontram abaixo da barreira. Idealmente, os programas que se encontram a um dado nível de abstracção contêm toda a informação necessária para lidar com um certo tipo de dados, a informação está "encapsulada" dentro desta camada conceptual, e escondem das restantes partes do programa o modo como a informação está representada, o que é conhecido por "anonimato da representação".

3. **(1.0)** Diga o que caracteriza a programação imperativa.

   Resposta:

   A programação imperativa baseia-se no conceito de efeito. Em programação imperativa, um programa é considerado como uma sequência de instruções, cada uma das quais produz um efeito.

4. **(1.0)** Distinga o comportamento FIFO ("first in first out") do comportamento LIFO ("last in first out"). Quais os tipos de informação que correspondem a cada um destes comportamentos?

   Resposta:

   "FIFO" designa o tipo de comportamento em que o primeiro elemento a entrar numa estrutura de informação será também o primeiro elemento a sair. As filas são a estrutura de informação que seguem este tipo de comportamento. De forma diferente, "LIFO" designa o tipo de comportamento em que o último elemento a entrar numa estrutura de informação será o primeiro elemento a sair. As pilhas são a estrutura de informação que seguem este tipo de comportamento.

5. Considere a seguinte variável:

```
teste = {'Portugal':{'Lisboa':[('Leonor', '1700-097'),
                               ('João', '1050-100')],
                     'Porto':[('Ana', '4150-036')]},
         'Estados Unidos':{'Miami':[('Nancy', '33136'),
                                    ('Fred', '33136')],
                           'Chicago':[('Cesar', '60661')]},
         'Reino Unido':{'London':[('Stuart', 'SW1H 0BD')]}}
```

   Qual o valor de cada um dos seguintes nomes? Se algum dos nomes originar um erro, explique a razão do erro.

   (a) **(0.2)** `teste['Portugal']['Porto']`
   Resposta:
   `[('Ana', '4150-036')]`

   (b) **(0.2)** `teste['Portugal']['Porto'][0][0]`
   Resposta:
   `'Ana'`

   (c) **(0.2)** `teste['Estados Unidos']['Miami'][1]`
   Resposta:
   `('Fred', '33136')`

   (d) **(0.2)** `teste['Estados Unidos']['Miami'][1][0][0]`
   Resposta:
   `'F'`

   (e) **(0.2)** `teste['Estados Unidos']['Miami'][1][1][1]`
   Resposta:
   `'3'`

6. **(1.0)** Escreva uma função *recursiva* chamada `soma_digitos_pares` que recebe como argumento um número inteiro não negativo `n`, e devolve a soma dos dígitos pares de `n`. Por exemplo,

```
>>> soma_digitos_pares(0)
0
>>> soma_digitos_pares(12426374856)
32
```

Resposta:

```
def soma_digitos_pares(num):
    if num == 0:
        return 0
    elif (num % 10) % 2 == 0:
        return soma_digitos_pares(num // 10) + (num % 10)
    else:
        return soma_digitos_pares(num // 10)
```

7. **(1.0)** Escreva uma função chamada `todos_lista` que recebe uma lista e um predicado de um argumento, e devolve *verdadeiro* caso todos os elementos da lista satisfaçam o predicado e *falso* em caso contrário. Por exemplo:

```
>>> todos_lista([4, 5, 6], lambda x: x > 5)
False
>>> todos_lista([4, 5, 6], lambda x: x >= 4)
True
```

Resposta:

```
def todos_lista(lst, pred):
    for el in lst:
        if not(pred(el)):
            return False
    return True
```

8. **(1.5)** Escreva uma função em Python que que recebe uma cadeia de caracteres, que contém o nome de um ficheiro, lê esse ficheiro, linha a linha, e calcula quantas vezes aparece cada uma das vogais. A sua função deve devolver um dicionário cujas chaves são as vogais e os valores associados correspondem ao número de vezes que a vogal aparece no ficheiro. Apenas conte as vogais que são letras minúsculas. Por exemplo,

```
>>> conta_vogais('testevogais.txt')
{'a': 36, 'u': 19, 'e': 45, 'i': 16, 'o': 28}
```

Resposta:

```
def conta_vogais(fich):
    f = open(fich, 'r', encoding='UTF-16')
    res = {'a':0, 'e':0, 'i':0, 'o':0, 'u':0}
    linha = f.readline()
    while linha != '':
        for c in linha:
            if c in res:
                res[c] = res[c] + 1
        linha = f.readline()
    f.close()
    return res
```

9. **(1.0)** Escreva em Python a função *recursiva* `junta_ordenadas` que recebe como argumentos duas listas ordenadas contendo números, em que cada lista não contém repetições, e que devolve uma lista ordenada, sem elementos repetidos, correspondente à junção das duas listas. Por exemplo:

```
>>> junta_ordenadas([1, 2, 3], [1, 5, 6, 7, 8])
[1, 2, 3, 5, 6, 7, 8]
```

Resposta:

```
def junta_ordenadas(lst1, lst2):

    if lst1 == []:
        return lst2
    elif lst2 == []:
        return lst1
    elif lst1[0] > lst2[0]:
        return [lst2[0]] + \
               junta_ordenadas(lst1, lst2[1:])
    elif lst1[0] < lst2[0]:
        return [lst1[0]] + \
               junta_ordenadas(lst1[1:], lst2)
    else:
        return [lst1[0]] + \
               junta_ordenadas(lst1[1:], lst2[1:])
```

10. (a) **(1.0)** Especifique as operações básicas do tipo abstracto de informação *carta* o qual é caracterizado por um naipe (espadas, copas, ouros e paus) e por um valor (A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K).
Resposta:
    i. *Construtor:*
    $cria\_carta : \{espadas, copas, ouros, paus\} \times \{A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K\} \mapsto carta$
    ii. *Selectores:*
    $naipe : carta \mapsto \{espadas, copas, ouros, paus\}$
    $valor : carta \mapsto \{A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K\}$
    iii. *Reconhecedor:*
    $carta : universal \mapsto lógico$
    iv. *Teste:*
    $cartas\_iguais : carta^2 \mapsto lógico$

    (b) **(1.0)** Defina a classe `carta` correspondente ao tipo da alínea anterior, incluindo o transformador de saída.
Resposta:

```
class carta:

    def __init__(self, naipe, valor):
        if naipe in ['espadas', 'copas', 'ouros', 'paus']:
            if valor in ['A', '2', '3', '4', '5', '6', '7', \
                         '8', '9', '10', 'J', 'Q', 'K']:
                self.n = naipe
                self.v = valor
            else:
```

```
                raise ValueError ('valor nao especificado')
            else:
                raise ValueError ('naipe nao especificado')

    def naipe(self):
        return self.n

    def valor(self):
        return self.v

    def __eq__(self, outra):
        return self.n == outra.naipe() and \
               self.v == outra.valor()

    def __repr__(self):
        return '(' + str(self.n) + ', ' + str(self.v) + ')'
```

(c) **(1.0)** Usando o tipo *carta*, defina uma função em Python que devolve uma lista em que cada elemento corresponde a uma carta de um baralho.
Resposta:

```
def todas_cartas():
    res = []
    for n in ['espadas', 'copas', 'ouros', 'paus']:
        for v in ['A', '2', '3', '4', '5', '6', '7', \
                  '8', '9', '10', 'J', 'Q', 'K']:
            res = res + [carta(n, v)]
    return res
```

(d) **(1.0)** Usando o tipo *carta* e recorrendo à função `random()`, a qual produz um número aleatório no intervalo $[0, 1[$, escreva uma função, `baralha`, que recebe uma lista correspondente a um baralho de cartas e baralha aleatoriamente essas cartas, devolvendo a lista que corresponde às cartas baralhadas.
SUGESTÃO: percorra sucessivamente as cartas do baralho trocando cada uma delas por uma outra carta seleccionada aleatoriamente.
Resposta:

```
def baralha(b):
    n_cartas = len(b)
    for i in range(n_cartas):
        al = int(random()*n_cartas) + 1
        b[i], b[al] = b[al], b[i]
    return b
```

11. Considere a função g, definida para inteiros não negativos do seguinte modo:

$$g(n) = \begin{cases} 0 & \text{se } n = 0 \\ n - g(g(n-1)) & \text{se } n > 0 \end{cases}$$

(a) **(1.0)** Escreva uma função recursiva em Python para calcular o valor de $g(n)$.
Resposta:

```
def g(n):
    if n == 0:
        return 0
    else:
        return n - g(g(n-1))
```

(b) **(1.0)** Siga o processo gerado por `g(3)`, indicando todos os cálculos efectuados.
Resposta:

```
g(3)
3-g(g(2))
3-g(2-g(g(1)))
3-g(2-g(1-g(g(0))))
3-g(2-g(1-g(0)))
3-g(2-g(1-0))
3-g(2-g(1))
3-g(2-g(1)))
3-g(2-(1-g(g(0))))
3-g(2-(1-g(0)))
3-g(2-(1-0))
3-g(2-(1))
3-g(1)
3-(1-g(g(0)))
3-(1-g(0))
3-(1-0)
3-(1)
2
```

(c) **(1.0)** Que tipo de processo é gerado por esta função?
Resposta:

A função gera um processo recursivo em árvore pois existem múltiplas fases de expansão e de contracção geradas pela dupla recursão em `g(g(n-1))`.

(d) **(1.0)** Como pode verificar na alínea (b), a sua função calcula várias vezes o mesmo valor. Para evitar este problema, podemos definir uma classe, `mem_g`, cujo estado interno contém informação sobre os valores de `g` já calculados, apenas calculando um novo valor quando este ainda não é conhecido. Esta classe possui um método `val` que calcula o valor de `g` para o inteiro que é seu argumento. Por exemplo,

```
>>> g = mem_g()
>>> g.val(0)
0
>>> g.val(12)
8
```

Defina a classe `mem_g`.
Resposta:

```
class mem_g:

    def __init__(self):
        self.g = {0:0}

    def val(self, n):
        if n in self.g:
            return self.g[n]
        else:
            g_n = self.val(n-1)
            g_g_n = self.val(g_n)
            self.g[n] = n - g_g_n
        return self.g[n]
```

12. **(1.5)** Considere os tipos abstractos de informação `Posição` e `Caminho`, como descritos no enunciado do projecto. O tipo `Posição` tem as operações `posicao`, `posicao_linha`, `posicao_coluna` e `posicao_igual`; o tipo `Caminho` tem as operações `caminho`, `caminho_junta_posicao`, `caminho_origem`, `caminho_apos_origem`, `caminho_destino`, `caminho_antes_destino` e `caminho_comprimento`. Assuma que estas operações estão disponíveis. Escreva o predicado `caminho_contem_ciclo` pedido no projecto para o tipo abstracto de informação `Caminho`.

Resposta:

Apresentamos duas soluções alternativas.
Usando a representação interna do caminho como uma lista de posições, com a variável interna chamada `rota`:

```
def caminho_contem_ciclo(self):
    for i in range(len(self.rota) - 1):
        if self.rota[i] in self.rota[i + 1:]:
            return True
    return False
```

Recorrendo a uma operação de alto nível:

```
def caminho_contem_posicao(self, obj):
    c2 = self
    for i in range(c2.caminho_comprimento()):
        if obj.posicao_igual(c2.caminho_origem()):
            return True
        c2 = c2.caminho_apos_origem()
    return False

def caminho_contem_ciclo(self):
    c2 = self
    for i in range(c2.caminho_comprimento() - 1):
        if c2.caminho_apos_origem().\
           caminho_contem_posicao(c2.caminho_origem()):
            return True
        c2 = c2.caminho_apos_origem()
    return False
```