TÉCNICO LISBOA

Fundamentos da Programação

Exame 1

Primeiro Semestre — 8 de Janeiro de 2016

2015/2016 — 2 horas

Esta prova, individual e sem consulta, tem 11 perguntas. A cotação de cada pergunta está assinalada entre parêntesis.

Escreva o seu número em todas as folhas da prova. O tamanho das respostas deve ser limitado ao espaço fornecido para cada questão. O corpo docente reserva-se o direito de não considerar a parte das respostas que excedam o espaço indicado.

Pode responder utilizando lápis.

Em cima da mesa devem apenas estar o enunciado, caneta ou lápis e borracha e cartão de aluno. Não é permitida a utilização de folhas de rascunho, telemóveis, calculadoras, etc.

---

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

1.1. (0.5) Um algoritmo é uma sequência finita de instruções bem definidas e não ambíguas, cada uma das quais pode ser executada mecanicamente utilizando eventualmente recursos ilimitados.

Solução: Falsa, as instruções de um algoritmo têm de ser executadas num período de tempo finito com uma quantidade de esforço finita.

1.2. (0.5) 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.

Solução: Verdadeira.

1.3. (0.5) No uso de tipos abstractos de dados (TADs) é indispensável que o programador conheça a representação interna dos elementos do tipo.

Solução: Falsa, os tipos abstractos de dados (TADs) visam precisamente garantir a independência entre as funções básicas do tipo e a representação dos elementos do tipo.

---

2. (1.5) Escreva em Python a função `digitos` que, dado um número inteiro positivo n, e verificando a correcção do argumento, devolve um tuplo de inteiros com os dígitos de n pela ordem que aparecem no mesmo. Caso o argumento não seja um inteiro positivo, deverá ser emitido um erro.

Solução:

```
def digitos(n):
```

```
    if not isinstance(n, int) and n > 0:
        raise ValueError ('int2digitos: argumento invalido')
    res = ()
    while n != 0:
        res = (n % 10, ) + res
        n = n // 10
    return res
```

---

3. Assuma que a função `digitos` descrita na questão anterior está definida.

3.1. (1.0) Defina uma função que implemente o algoritmo que permite determinar o número inteiro que resulta de remover todos os dígitos pares de um outro número inteiro dado como parâmetro. Por exemplo, dado o número 340512, a função deverá retornar o número 351.

Solução:

```
def sem_pares(n):
    res = 0
    k = 0
    while n > 0:
        if n%2 == 1:
            res += n%10 * 10**k
            k += 1
        n = n//10
    return res
```

3.2. (1.0) Defina novamente uma função que implemente o algoritmo anterior usando funcionais sobre listas. Deverá incluir a definição dos funcionais que utilizar.

Solução:

```
def acumulador(f, lst):
    res = lst[0]
    for x in lst[1:]:
        res = f(res, x)
    return res

def filtra(p, lst):
    res = []
    for x in lst:
        if p(x):
            res.append(x)
    return res

def sem_pares(n):
    return acumulador(lambda x,y: 10*x + y, \
               filtra(lambda x : x%2, digitos(n)))
```

4. (2.0) Usando recursão de cauda, escreva a função `soma_divisores` que recebe um número inteiro positivo n, e que devolve a soma de todos os divisores de n.

Solução:

```
def soma_divisores(n):
    def f_aux(d, r):
        if d > n:
            return r
        if n%d == 0:
            return f_aux(d+1, r + d)
        else:
            return f_aux(d+1, r)
    return f_aux(1, 0)
```

---

5. (1.5) Considere a passagem de argumentos em Python, que é por referência dos objectos, e a seguinte interacção.

```
>>> def adiciona_a_todos(t, x):
...     for l in t:
...         l.append(x)
...     t = ()
...
>>> t = ([], [])
>>> adiciona_a_todos(t, 2)
>>> adiciona_a_todos(t, 3)
>>> adiciona_a_todos(t, 5)
>>> t
```

Indique o resultado que deverá aparecer após a última instrução.

Solução: `([2,3,5],[2,3,5])`

---

6. (1.5) Considere a seguinte interacção em Python:

```
>>> def somatorio(l_inf, l_sup, f, passo):
...     resultado = 0
...     while l_inf <= l_sup:
...         resultado += f(l_inf)
...         l_inf = passo(l_inf)
...     return resultado
...
>>> somatorio(1, 10, A, B)
165
>>>
```

Sabendo que o valor 165 corresponde à soma dos quadrados dos números ímpares entre 1 e 10, indique as expressões pelas quais devemos substituir `A` e `B` para produzir o resultado esperado.

Solução: Podemos substituir por `lambda x:x*x` e `lambda x:x+2`, respectivamente.

---

7. (2.0) Escreva a função `conta_linhas` que dada uma cadeia de caracteres com o nome de um ficheiro, retorna o número de linhas que ocorrem no ficheiro e que não estão em branco, ou seja, apenas com o caractere fim de linha.

Solução:

```
def conta_linhas(nome):
    ficheiro = open(nome, 'r')
    linhas = ficheiro.readlines()
    ficheiro.close()
    n = 0
    for linha in linhas:
        if len(linha) > 1:
            n += 1
    return n
```

---

8. (2.0) Considere a função

```
1  def desconhecido(lst):
2      for i in range(len(lst)):
3          k = i
4          for j in range(i + 1, len(lst)):
5              if lst[j] < lst[k]:
6                  k = j
7          lst[i], lst[k] = lst[k], lst[i]
```

e a seguinte interacção:

```
>>> lst = [8, 9, 10, 2, 3, 1, 11, 6]
>>> desconhecido(lst)
```

Indique o conteúdo da lista `lst` após a segunda iteração do ciclo `for` na linha 2.

Solução: `[1,2,10,9,3,8,11,6]`.

---

9. (2.0) Implemente a função `agrupa_por_chave` que dada uma lista de pares chave valor (k, v) (representados por tuplos de dois elementos), retorna um dicionário que a cada chave k associa uma lista com os valores v para essa chave encontrados na lista passada como argumento. Exemplo:

```
>>> agrupa_por_chave([('a', 8), ('b', 9), ('a', 3)])
{'a': [8, 3], 'b': [9]}
```

Solução:

```
def agrupa_por_chave(pares):
    res = {}
    for par in pares:
        if par[0] not in res:
```

```
            res[par[0]] = []
        res[par[0]].append(par[1])
    return res
```

---

10. (2.0) Considere a seguinte assinatura para o TAD pilha (mutável) com comportamento LIFO:

```
pilha_nova: {} --> pilha
e_pilha: universal --> booleano
pilha_vazia: pilha --> booleano
pilha_topo: pilha --> universal
pilha_retira: pilha --> pilha
pilha_empurra: pilha x universal --> pilha
pilha_igual: pilha x pilha --> booleano
```

Proponha uma representação interna para este TAD e implemente as operações básicas descritas na assinatura. As operações `pilha_retira` e `pilha_empurra` devem modificar e retornar a pilha modificada.

Solução: Vamos utilizar uma lista como representação interna da pilha, em que o topo da pilha é a última posição da lista.

```
def pilha_nova():
    return []

def e_pilha(x):
    return isinstance(x, list)

def pilha_vazia(p):
    return p == []

def pilha_topo(p):
    if not pilha_vazia(p):
        return p[-1]

def pilha_retira(p):
    if not pilha_vazia(p):
        del(p[-1])
    return p

def pilha_empurra(p, x):
    p.append(x)
    return p

def pilha_igual(p, q):
    return p == q
```

---

11. (2.0) Considere ainda a assinatura para o TAD pilha descrita na questão 10 e implemente a função `calculadora` que deve simular o comportamento de uma calculadora de pilha,

```
calculadora: pilha x int|float|str --> {}
```

em que a memória da calculadora é representada por uma pilha de números (inteiros ou reais) e que aceita as operações aritméticas binárias `+,-,*,/` e a instrução `=` para mostrar/imprimir o resultado. A ideia é colocar na pilha os operandos, e substitui-los pelo resultado da aplicação das operações introduzidas na calculadora. Esta função deve produzir interacções idênticas à seguinte:

```
>>> memoria = pilha_nova()
>>> calculadora(memoria, 5)
>>> calculadora(memoria, 6)
>>> calculadora(memoria, 3)
>>> calculadora(memoria, '+')
>>> calculadora(memoria, '=')
9
>>> calculadora(memoria, '*')
>>> calculadora(memoria, '=')
45
>>> calculadora(memoria, '*')
Erro: Argumentos em falta!
>>> calculadora(memoria, 2)
>>> calculadora(memoria, '*')
>>> calculadora(memoria, '=')
90
```

Note que deve respeitar as barreiras de abstracção.

Solução:

```
def calculadora(memoria, x):
    if isinstance(x, (int, float)):
        pilha_empurra(memoria, x)
    elif x in ('+', '-', '*', '/'):
        if pilha_vazia(memoria):
            print('Erro: Argumentos em falta!')
            return memoria
        u = pilha_topo(memoria)
        memoria = pilha_retira(memoria)
        if pilha_vazia(memoria):
            print('Erro: Argumentos em falta!')
            pilha_empurra(memoria, u)
            return
        v = pilha_topo(memoria)
        pilha_retira(memoria)
        pilha_empurra(memoria, eval(str(u) + x + str(v)))
    elif x == '=':
        if pilha_vazia(memoria):
            print('Erro: Argumentos em falta!')
        else:
            print(pilha_topo(memoria))
```