TÉCNICO LISBOA

FUNDAMENTOS DA PROGRAMAÇÃO

1º Teste, 19 de abril de 2013, 1h30m

1. **(1.5)** Considere a seguinte gramática em notação BNF:

```
<Morada> ::= <Nome>  <Numero>,  <Codigo>
<Nome> ::= <letra> | <letra><Nome>
<letra> ::= A | B | R | u | a | v | i | r | o
<Numero> :: = <num> | <num><Numero>
<num> :: = 1 | 2 | 3 | 4 | 5
<Codigo> ::= <num><num><num>
```

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

Solução:

Símbolos terminais:

A, B, R, u, a, v, i, r, o, 1, 2, 3, 4, 5, "," (vírgula)

   b. **(1.0)** Para cada uma das seguintes frases justifique se esta pertence ou não à linguagem definida por esta gramática:

      i. Rua 32, 123
      ii. AvWWW 45, 345
      iii. ABRu
      iv. Bairro 52, 4

Solução:

i. Pertence, pois está de acordo com a gramática.
ii. Não pertence, pois o elemento "AvWWW" não corresponde a um <Nome>, uma vez que o símbolo "W" não é uma letra admitida na gramática.
iii. Não pertence, pois não está de acordo com a sintaxe que define uma <Morada>: não tem <Numero>, nem vírgula, nem <Codigo>.
iv. Não pertence, pois o último elemento da frase não corresponde a um código válido (não corresponde a uma sequência de 3 elementos <num>)

2. **(1.5)** Escreva uma gramática em notação BNF que permita escrever as seguintes expressões:

   a. `(1 + 2)`

   b. `(3 / 9)`

   c. `(20 / 10)`

   d. `(5 * 6)`

Solução:

Uma gramática possível seria:

<expressao> ::= (<argumento> <operador> <argumento>)

<operador> ::= + | * | /

<argumento> ::= <digito>$^{+}$

<digito> ::= 1 | 2 | 3 | 5 | 6 | 9 | 0

3. **(3.25)** Diga qual é o resultado de avaliar sequencialmente os seguintes comandos no interpretador de Python. Em cada alínea indique o que é devolvido ou impresso pela execução do comando. Caso o comando não retorne ou imprima nada, deverá indicar "`-- nada --`" em frente da alínea correspondente.

```
a.    >>> 3 + 4
b.    >>> int(3.7)
c.    >>> float(3)
d.    >>> a, b = 4, 5
e.    >>> print('a =', a, 'e b =', b)
f.    >>> x = '7 + 3'
g.    >>> eval(x)
h.    >>> num = ((1, 2), 3, 4 ) + (6, 8, 10, 11)
i.    >>> num
j.    >>> print(num[:a] + num[b+1:])
k.    >>> x
```

Solução:

```
>>> 3 + 4
7
>>> int(3.7)
3
>>> float(3)
3.0
```

```
>>> a, b = 4, 5
-- nada --
>>> print('a =', a, 'e b =', b)
a = 4 e b = 5
>>> x = '7 + 3'
-- nada --
>>> eval(x)
10
>>> num = ((1, 2), 3, 4 ) + (6, 8, 10, 11)
-- nada --
>>> num
((1, 2), 3, 4, 6, 8, 10, 11)
>>> print(num[:a] + num[b+1:])
((1, 2), 3, 4, 6, 11)
>>> x
'7 + 3'
```

4. **(1.25)** Escreva a função *mult* que pede ao utilizador dois números inteiros e retorna o resultado da multiplicação desses números. A função deve pedir os argumentos ao utilizador indicando `'Introduza numero inteiro:'`.

Solução:

```
def mult():
    a = eval(input('Introduza numero inteiro: '))
    b = eval(input('Introduza numero inteiro: '))
    return a * b
```

5. **(1.75)** Escreva a função *escreve_par* que recebe como parâmetro um número inteiro, `n1`, e escreve para o ecran todos os números pares pertencentes ao intervalo `[0, n1]`. Esta função deve usar o ciclo `while`.

Solução:

```
def escreve_par(n1):
    i = 0
    while i <= n1
        print(i)
        i = i + 2
```

6. **(1.75)** Escreva a função *escreve_impares_tuplo* que recebe um tuplo de inteiros e devolve o número de elementos ímpares no tuplo. A função deve validar se o parâmetro recebido é um tuplo e gerar um erro de valor (`ValueError`) em caso negativo.

Solução:

```
def escreve_impares_tuplo(t):
    if not isinstance(t, tuple):
        raise ValueError('escreve_impares_tuplo: parametro nao e tuplo')
    cont = 0
    for i in t:
        if i % 2 != 0:
            cont = cont + 1
    return cont
```

7. **(1.0)** Escreva a função *imprime_tuplo* que recebe um tuplo de inteiros e escreve no ecran o índice de cada elemento no tuplo e o respetivo valor da seguinte forma, para o tuplo `(2, 4, 7)`:

```
posicao 0 = 2
posicao 1 = 4
posicao 2 = 7
```

Solução:

```
def imprime_tuplo(t):
    for i in range(len(t)):
        print('posicao', i, '=', t[i])
```

8. **(2.75)** Escreva a função *altera_posicao* que tem como argumentos um tuplo `t` de caracteres, um inteiro `i` e um carácter `c` e devolve um novo tuplo, igual a `t`, mas em que a posição `i` se encontra preenchida com o carácter fornecido. Caso o inteiro fornecido não corresponda a uma *posição válida* no tuplo, a função deverá gerar uma mensagem de erro do tipo (`IndexError`). Uma *posição válida* é uma posição dentro dos limites dos tuplo e esteja preenchida pela cadeia de carateres `' '`. Deve também validar se `c` é do tipo cadeia de carateres e gerar uma mensagem de erro do tipo (`ValueError`) em caso negativo.

Solução:

```
def altera_posicao(t, i, c):
    if not isinstance(c, str):
        raise ValueError('altera_posicao: nao e caracter')
    if not (0 <= i < len(t) and t[i] == ' '):
        raise IndexError('altera_posicao: posicao invalida')
    return t[:i] + tuple(c) + t[i+1:]
```

9. **(1.75)** Escreva a função *monta_frase* que recebe como argumento uma lista com elementos e retorna uma cadeia de carateres que corresponde à concatenação apenas dos elementos da lista que correspondem a cadeias de caracteres. A função deve validar se a lista está vazia e em caso afirmativo em caso afirmativo deve devolver uma cadeia de carateres com a mensagem `'Lista vazia'`.

Solução:

```
def monta_frase(lst):
    res = 'Lista vazia'
    if lst != []:
        res = ''
        for el in lst:
            if isinstance(el, str):
                res = res + el
    return res
```

10. **(2.5)** Escreva uma função que recebe como argumentos uma lista `lst` e um tuplo `z`:
    a. **(1.5)** A função deve procurar se o tuplo `z` se encontra na lista `lst`. Caso este elemento esteja presente uma ou mais vezes, a função deve retornar a lista sem esse elemento, caso contrário deve imprimir a cadeia de caracteres `'Elemento nao presente.'` Esta função não retorna nada.

Solução:

```
1 def procura_lista(lst, z):
2     if z not in lst:
3         print('Elemento nao presente.')
4     else:
5         for i in range(len(lst)-1, -1, -1):
6             if lst[i] == z:
7                 del(lst[i])
8         return lst
```

    b. **(1.0)** Justifique como é que o tempo de execução da função, cresce em função da dimensão da lista `lst`. Identifique no código da alínea anterior qual a linha que mais condiciona o tempo de execução da função.

Solução:

No pior caso, será necessário percorrer a lista toda. Por essa razão, o tempo de execução da função anterior cresce *linearmente* com o comprimento da lista. Esta dependência é visível na linha 5 da função.

11. **(1.0)** Considere a seguinte lista de números inteiros.

[3, 6, 1, 7, 0]

Considere que esta lista é ordenada por um algoritmo de ordenação da sua preferência. Nomeie o algoritmo utilizado e represente o conteúdo da lista após cada passagem do algoritmo. Após a execução do algoritmo a lista deve ficar ordenada.

Solução:

```
Ordenação por borbulhamento:
  [3, 6, 1, 7, 0]
  [3, 1, 6, 0, 7]
  [1, 3, 0, 6, 7]
  [1, 0, 3, 6, 7]
  [0, 1, 3, 6, 7]
Ordenação shell:
 (intervalo = 2)
  [3, 6, 1, 7, 0]
  [1, 6, 0, 7, 3]
  [0, 6, 1, 7, 3]
 (intervalo = 1)
  [0, 1, 6, 3, 7]
  [0, 1, 3, 6, 7]
Ordenação por selecção:
  [3, 6, 1, 7, 0]
  [0, 6, 1, 7, 3]
  [0, 1, 6, 7, 3]
  [0, 1, 3, 7, 6]
  [0, 1, 3, 6, 7]
```