Resolução de Problemas em Prolog

[1] Marcelo Ruan Moura Araújo e [2] André Ribeiro de Brito
[1 2] Universidade Federal de São Carlos - UFSCar
[1 2] Departamento de Computação
[1] Biomedical Image Processing Group - BIP Group
[1] marcelo.araujo@ufscar.br e [2] andrerbrito@ufscar.br

1. Problemas

1. Dada uma lista *Lin* que contém elementos de qualquer tipo, possivelmente com repetições,construir outra lista Lout que mostre quantas vezes cada elemento atômico (átomo, número, lista vazia) aparece na lista dada, inclusive nas sublistas. A lista *Lout* deve conter pares de elementos sendo o primeiro elemento do par um elemento atômico que aparece na lista dada e o segundo elemento do par, o número de vezes que esse elemento aparece na lista. Variáveis e estruturas devem ser descartadas e não serão usadas na contagem.
Por exemplo, dada a lista:

```
LIn = (a b z x 4.6 (a x) () (5 z x) ())
```

Deve ser construída a lista

```
Lout = ((a 2) (b 1) (z 2) (x 3) (4.6 1) (() 2) (5 1) )
```

2. Dada uma lista *Lin* com elementos de qualquer tipo, construir a codificação Lout dessa lista em que repetições consecutivas de elementos devem ser substituídas por pares da forma(N,E), onde N é o número de repetições consecutivas do elemento E.

```
LIn = (a a a a b c c a a d e e e e)
```

Deve ser construída a lista

```
Lout = ((4 a) (1 b) (2 c) (2 a) (1 d) (4 e))
```

3. Dada uma lista Lin que seja a codificação de outra lista no formato do exercício anterior, construir a decodificação dessa lista codificada.
Por exemplo, dada a lista:

```
LIn = ((4 a) (1 b) (2 c) (2 a) (1 d) (4 e))
```

Deve ser construída a lista:

```
Lout = (a a a a b c c a a d e e e e)
```

Observações:

- Podem ser utilizados, se necessário, as funções pré-definidas de LISP car, cdr, cons, append, listp, atom, member, length, list.
- Os trabalhos podem ser feitos em duplas, que deverão ser as mesmas em todos os trabalhos;
- Usar OBRIGATORIAMENTE, CLisp para implementar o trabalho;
- Entregar, via AVA (tarefa com envio de arquivo único):
- Relatório (Arquivo txt, doc ou pdf) com listagem do código fonte, explicação do funcionamento de cada um dos predicados definidos e resultados de execução de pelo menos dois exemplos de cada exercício;
- Arquivo do LISP com o código fonte do trabalho.
- **DATA DE ENTREGA: 07/06/2018**

2. Soluções

2.1. Questão 1

`planifica (lista)`: Esse predicado dado uma lista com sub-listas aninhadas remove todos os níveis mantendo apenas o primeiro nível de lista, por exemplo, dado uma lista `(a b (c (d (e) f ) g) h )` retorna uma lista `(a b c d e f g h)`, para elementos nulos o predicado retorna `nil`.

`conta (x lista)`: Dado um elemento qualquer para o parâmetro "x" e uma lista sem sub-listas aninhadas para o parâmetro "lista" retorna a quantidade de ocorrências desse elemento na lista, o fato de só aceitar listas planificadas não afeta a resolução do problema, vide da função definida acima.

`pares (lista)`: Recebe uma lista planificada e forma a saída para uma lista com "n" sub-listas, sendo cada uma delas com o elemento contado e a quantidade de ocorrências da lista de entrada, na sua chamada recursiva remove os elementos já contados para que não os contabilize novamente.

`apagarelemento (elemento lista)`: Recebe dois parâmetros sendo **`elemento`** e **`lista`**, **`elemento`**, sendo um carácter, número, etc. O parâmetro **`lista`** recebe uma lista planificada, sua função é unicamente chamar a função apaga, passados esses dois parâmetros mais uma lista vazia.

`apaga (elemento lista temp)`:Recebe três parâmetros sendo elemento e lista provindos do predicado descrito acima, **`temp`** sendo um uma lista vazia também do predicado descrito anteriormente, sendo esse parâmetro a saída do meu predicado, pois é nesse parâmetro que retorna a lista com o elemento removido.

`junta (lista)`:Análogo a função main da linguagem C, seu papel é chamar o predicado pares com a lista planificada, que por sua vez concatena o elemento e a contagem do mesmo mais a concatenação dessa resolução com uma chamada recursiva de junta, apagando a cabeça para a instância da lista chamada e retornando o corpo com o elemento excluído.

As duas saídas testadas para a solução desse problema são:

```
CL-USER 38 : 1 > (junta '(a b z x 4.6 (a x) () (5 z x) ()))
((A 2) ((NIL 2) ((B 1) (# #))))
```

As hashtags são equivalentes a reticências, resume o comportamento para os demais elementos da lista. Na verdade seu comportamento completo é:

```
((A 2) ((NIL 2) ((B 1) (z 1) (x 3) (4.6 1) (5 1))
```

```
CL-USER 44 : 1 > (junta '(1 2 3 (3 2 (1))))
((1 2) ((2 2) ((3 2) NIL)))
```

```lisp
(defun planifica (lista)
  (cond
   ((null lista) nil)
   ((atom (car lista)) (cons (car lista) (planifica (cdr lista))))
   (t (append (planifica (car lista)) (planifica (cdr lista))))))


(defun conta (x lista)
  (cond
   ((null lista) 0)
   ((equal (car lista) x) (+  1 (conta x (cdr lista))))
   (t (conta x (cdr lista)))))

(defun junta (lista)
  (cond
   ((null lista) nil)
   (t (pares (planifica lista)))))

(defun pares (lista)
  (cond
   ((null lista) nil)
    (t (list (list (car lista) (conta (car lista) lista)) (pares (apagarelemento
(car lista) (cdr lista)))))))

(defun apaga (elemento lista temp)
  (cond
   ((null lista) temp)
   ((equal elemento (car lista)) (apaga elemento (cdr lista) temp))
   (t (apaga elemento (cdr lista) (cons (car lista) temp)))))

(defun apagarelemento (elemento lista)
  (apaga elemento lista '()))
```

2.2. Questão 2

`exercicio2 (lista)` : recebe uma lista testa se é nula, se não com o comando **`cons`** coloca os elementos um por vez na lista de saída, de modo que o comando seguinte concatena as sublistas do o tamanho de elementos da saída do predicado `junta(lista)`(explicado no parágrafo seguinte) e a cabeça da lista, por faz a chamada recursiva com a saída do predicado `(remov lista)` e retornar a executar até o fim da lista.

`junta (lista) : recebe uma lista e verifica se o primeiro elemento é igual ao seguinte, até não encontrar a ocorrência do mesmo elemento, seu ponto de parada é definida ao encontrar um caractere diferente do inicial e retornar uma lista os elementos repetidos, por exemplo:`
`Para uma entrada:`
`(junta '(a a a a a a b d c d d))`
Tem uma saída:
`(a a a a a a)`
Por isso, que no predicado `exercicio2 (lista)` o retorno dessa função é útil para a finalização do exercício.

`remover (lista)` : recebe uma lista e irá fazer o processo contrário de junta (lista), onde ao verificar elementos repetidos consecutivamente remove e retorna o corpo restante. Usando o mesmo exemplo do predicado anterior:
`(remover '(a a a a a a b d c d d))`
Tem uma saída
`(b d c d d)`
Dessa forma o predicado principal sempre que tiver uma quebra as repetições irá chamar sem as repetições computadas ao chamar recursivamente com o resto dos elementos, assim resolvendo problema proposto.

Exemplos de saída:

```
CL-USER 1 : 1 >(exercicio2 '(a a a a b c c a a d e e e e))
((4 A) (1 B) (2 C) (2 A) (1 D) (4 E))

CL-USER 2 : 1 > (exercicio2 '(1 1 1 2 3 3  5 1 1 9 9 2))
((3 1) (1 2) (2 3) (1 5) (2 1) (2 9) (1 2)
```

```
(defun exercicio2 (lista)
     (if (eql lista nil)
          nil
          (cons   (list   (length   (junta   lista))   (car   lista))
(exercicio2 (remover lista)))
     )
)

(defun junta (lista)
    (cond ((eql lista nil) nil)
       ((eql (cdr lista) nil) lista)
         ((equal (car lista) (cadr lista))
             (cons (car lista) (junta (cdr lista))))
         (t (list (car lista)))
    )
)

(defun remover (lista)
    (cond ((eql lista nil) nil)
       ((eql (cdr lista) nil) nil)
         ((equal (car lista) (cadr lista))
             (remover (cdr lista)))
         (t (cdr lista))))
```

2.3. Questão 3

```
(defun repete (n lista)
(if (zerop n) nil
(cons lista (repete (- n 1)lista))))

(defun exercicio3(lista)
(if (null lista) nil
(nconc (apply 'repete (car lista))(exercicio3 (cdr lista)))))
```

```
[17]> (exercicio3 '((4 a)(1 b)(2 c)(2 a)(1 d)(4 e)))
(A A A A B C C A A D E E E E)
[18]> (exercicio3 '((7 w)(1 q)(2 a)(1 w)(4 e)(1 o)))
(W W W W W W W Q A A W E E E E O)
```

```
[15]> (defun repete (n lista)
(if (zerop n) nil (cons lista (repete (- n 1) lista))))
REPETE
[16]> (defun exercicio3(lista)
(if(null lista) nil
(nconc (apply 'repete (car lista)) (exercicio3 (cdr lista)))))
EXERCICIO3
[17]> (exercicio3 '((4 a)(1 b)(2 c)(2 a)(1 d)(4 e)))
(A A A A B C C A A D E E E E)
[18]> (exercicio3 '((7 w)(1 q)(2 a)(1 w)(4 e)(1 o)))
(W W W W W W W Q A A W E E E E O)
```

`repete (n lista)`: recebe um número inteiro “n” e uma lista, verificando se n é igual a zero retorna nil, caso contrário, retorna a lista e decrementa n - 1 vezes.

`exercicio3 (lista)`: verifica se a lista é vazia, assim retornando nil, caso contrário, concatena a lista, aplica-se a função “repete” junto a uma lista de argumentos sendo ela a cabeça e chamando-se recursivamente a cauda da lista.