U.PORTO
FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Professor Vitor Costa

Programação em Lógica

Mini-Trabalhos

Junho de 2022

Trabalho realizado por:

Pedro Leite - 201906697

1. Matrizes

1.1. Concatenar Listas

```
term([X|XS], Result):- term(XS, R1),append(X, R1, Result).
term([], Result):- Result=[].

> term([[00, 01, 02], [ 10, 11, 12]], R).
R = [0, 1, 2, 10, 11, 12]
```

O "term" recebe uma lista de listas que representa uma matriz e devolve uma lista com todos os elementos das listas que pertencem à lista principal.

Para fazer isto vamos fazendo "append" da cabeça da lista que recebemos (que é uma lista), à nossa lista "Result" e chama-mos recursivamente o resto da lista.

Se a lista que recebemos estiver vazia, o "Result" também vai ser uma lista vazia.

1.2. Obter Elemento, Linha e Coluna

```
%Número de colunas e de linhas
length_of(N, L) :- length(L, N).
size(M, R, C) :- length(M, R), maplist(length_of(C), M).

> size([[1, 2, 3],[1, 2, 3]], R, C).
C = 3, R = 2
```

O "size" recebe uma lista de listas, que representa uma matriz e devolve o número de colunas e de linhas, respetivamente.

Para fazer isto é chamada a função "length" da matriz, que devolve o número de listas da lista principal, que é o número de linhas. Para determinar o número de colunas aplica-mos a função "length_of" a todas as colunas da matriz.

```
%obter elemento
elem(L, Row, Col, Res):- size(L, _, C), Pos is Row*C+Col, term(L, Result), nth0(Pos, Result, Res).
elem([], _, _, []).

> elem([[1, 2, 3],[4, 5, 6]], 1, 1, R).
R = 5
```

A "elem" recebe uma lista de listas, que representa uma matriz, uma linha e uma coluna, que representam uma posição da matriz e devolvem o valor nessa posição da matriz.

Para fazer isto é chamada a função "size", para retornar apenas o número de colunas. Definimos uma variável "pos" que faz: linha*número de colunas+coluna. Chamamos a função "term", para transformas a lista de listas, numa lista apenas. E dentro dessa lista retornamos o valor na "pos", através da função "nth0".

Se a lista que recebemos estiver vazia, o "R" também vai ser uma lista vazia.

```
%obter linha
line(L, Row, Result) :- nth1(Row, L, Result).
line([], _, []).

> line([[1, 2], [3, 4], [5, 6]], 2, R).
R = [3, 4]
```

A "line" recebe uma lista de listas, que representa uma matriz e uma linha, esta função devolve o conteúdo dessa linha na matriz, na forma de lista. Para tal utilizamos a função auxiliar "nth1".

Se a lista que recebemos estiver vazia, o "Result" também vai ser uma lista vazia.

```
%obter coluna
column([X|XS], Col, [X2|XS2]):- line(X, Col, X2), column(XS,Col,XS2).
column([], _, []).

> column([[1, 2], [3, 4], [5, 6]], 1, R).
R = [1, 3, 5]
```

A "column" recebe uma lista de listas, que representa uma matriz e uma coluna, esta função devolve o conteúdo dessa coluna na matriz, na forma de lista. Para tal chamamos a função "line" para cada linha da matriz e concatenamos o valor obtido ao "R" e chamamos recursivamente a função para o resto das linhas da matriz.

Se a lista que recebemos estiver vazia, o "R" também vai ser uma lista vazia.

1.3. Somar

```
sum(X1, X2, X3) :- X3 is X1+X2.
addmat(M1, M2, M3) :- maplist(maplist(sum), M1, M2, M3).
addmat([], M2, M2).
addmat(M1, [], M1).
addmat([], [], Result) :- Result=[].

> addmat([[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]], R).
R = [[8, 10, 12], [14, 16, 18]]
```

A "addmat" recebe duas listas de listas que representam matrizes e devolve uma terceira lista de listas que também representa uma matriz resultado da soma das matrizes inicias. Para tal aplicamos a função "sum" a todos os elementos das matrizes, colocando o resultado dessa soma na terceira matriz.

Se uma das listas estiver vazia, o "R" vai ser igual à lista não vazia. Se ambas as listas estiverem vazias, o "R" também vai ser vazio.

1.4. Somar Elementos

```
addelem(L, Row1, Col1, Row2, Col2, Result) :- elem(L, Row1, Col1, R1), elem(L, Row2, Col2, R2), Result is R1+R2.
addelem([], _, _, _, _, 0).

> addelem([[1, 2, 3], [4, 5, 6]], 1, 1, 0, 0, R).
R = 6
```

A "addelem" recebe uma lista de listas que representa uma matriz, duas linhas e duas colunas que representam duas posições da matriz. E devolve a soma dessas duas posições. Para tal chamamos a função "elem", para ambas as posições e devolvemos o resultado.

Se a lista que recebemos estiver vazia, o "Result" vai ser 0.

1.5. Mapear

```
%adicionar um valor a todos os elementos da matriz
addaux(_, [], []) :- !.
addaux(A, [X|XS], Result):- F is X+A, addaux(A, XS, Result2), append([F], Result2, Result), !.

> addaux(2, [1, 2, 3], R), !.
R = [3, 4, 5]

%subtrair um valor a todos os elementos da matriz
subaux(_, [], []) :- !.
subaux(A, [X|XS], Result):- F is X-A, subaux(A, XS, Result2), append([F], Result2, Result), !.

> subaux(1, [1, 2, 3], R), !.
```

```
R = [0, 1, 2]

%multipicar um valor a todos os elementos da matriz
mulaux(_, [], []) :- !.
mulaux(A, [X|XS], Result):- F is X*A, mulaux(A, XS, Result2),
append([F], Result2, Result), !.

> mulaux(5, [1, 2, 3], R), !.
R = [5, 10, 15]

%dividir um valor a todos os elementos da matriz
divaux(_, [], []) :- !.
divaux(A, [X|XS], Result):- F is X/A, divaux(A, XS, Result2), append([F],
Result2, Result), !.

> divaux(2, [2, 4, 6], R), !.
R = [1, 2, 3]
```

As funções "addaux", "subaux", "mulaux", "divaux", recebem um valor e uma lista e devolve outra lista, resultado da operação do valor com todos os elementos da lista inicial. Para tal é usada uma variável auxiliar que faz a operação do valor com a cabeça da lista, dá "append" desse valor a uma lista nova e aplicamos a função recursivamente ao resto da lista. Quando chegar ao final da lista, devolve a lista final.

```
%mapear (usar addaux, subaux, mulaux, divaux)
mapmat(F, [X|XS], Result) :- call(F, X, Result2), mapmat(F, XS, Result3),
append([Result2], Result3, Result), !.
mapmat(_, [], []) :- !.
```

```
> mapmat(addaux(2), [[1, 2, 3], [4, 5, 6]], R).
```

A “mapmat” recebe uma função (“addaux”, “subaux”, “mulaux" ou “divaux") com um valor, uma lista de listas que representa uma matriz e devolva outra lista de listas que também representa uma matriz, que é o resultado da aplicação da função a todas as listas das lista fornecida. Para tal aplicamos a função à lista inicial, damos “append” ao resultado numa lista nova e aplicamos a função recursivamente às outras listas. Quando não houver mais listas, devolvemos a lista de listas.

1.6. Reduzir

```
%somar 2 valores
addaux2(L1, L2, Result) :- Result is L1+L2.

> addaux2(1, 2, R)
R = 3

%subtrair 2 valores
subaux2(L1, L2, Result) :- Result is L1-L2.

> subaux2(5, 2, R)
R = 3

%multiplicar 2 valores
mulaux2(L1, L2, Result) :- Result is L1*L2.

> mulaux2(3, 2, R)
R = 6

%dividir 2 valores
```

```
divaux2(L1, L2, Result) :- Result is L1/L2.

> divaux2(6, 2, R)
R = 3
```

As funções “addaux2”, “subaux2”, “mulaux2”, “divaux2”, recebem dois valores e retornam a operação com esses dois valores.

```
reducemat(F, Matrix, Reduced) :- reduceleftlist(F, Matrix, List),
reducelist(F, List, Reduced).

reduceleftlist(_, [], []).
reduceleftlist(F, [R1|R2], Matrix):- reducelist(F, R1, Nlist), reduceleftlist(F,
R2, X), append([Nlist], X, Matrix).

reducelist(_, [A], A) :- !.
reducelist(F, [A, B|R1], List) :- call(F, A, B, Nelm), reducelist(F, [Nelm|R1],
List), !.

> reducemat(divaux2, [[1, 2, 3], [4, 5, 6], [7, 8, 9]], R), !.
R = 12.85…
```

A “reducemat” recebe uma função (“addaux2”, “subaux2”, “mulaux2” ou “divaux2”) e uma lista dele listas que representa uma matriz e devolve outra lista de listas que também representa uma matriz, resultado da redução da função utilizada como argumento na matriz. Para tal é chamada a função “reduceleftlist” que chama a função “reduceclist” que aplica a função aos dois primeiros argumentos da lita e chama recursivamente o resto da lista. Aplicamos a função “reducelist” a todos as listas da lista de listas do argumento. No final devolvemos o resultado.

1.7. Transpor

```
transpose([X|XS], Result) :- foldl(transpose2, X, Result, [X|XS], _).
transpose([], []).
transpose2(_, X, List1, List2) :- maplist(listfirstrest, List1, X, List2).
listfirstrest([X|XS], X, XS).
```

```
> transpose([[a, b, c], [d, e, f]], R).
R = [[a, d], [b, e], [c, f]]
```

A "transpose" recebe umas lista de listas que representa uma matriz e devolve outra lista de listas que representa matriz transposta da matriz inicial. Para tal com a função "foldl" aplicamos a função "transpose2" a todas as listas da lista de listas fornecida como argumento, que vai mapear essas listas com a função "listfirstrerst" que devolve a cabeça da lista e a cauda.

2. Árvores Binárias

2.1. Calcular Máximo e Mínimo

```
%calcular máximo
max(t(X,_,nil),Ans):-
   Ans=X.
max(t(_,_,B),Ans):-
   max(B,Ans).
```

```
> max(t(4,t(2,t(1,nil,nil),t(3,nil,nil)),t(5,nil,t(7,t(6,nil,nil),nil))), R).
R = 7
```

O "max" recebe uma árvore binária e retorna o máximo dessa árvore, que é último elemento (o elemento mais à direita da árvore). A função devolve recursivamente a árvore mais à direita e assim que não houver mais árvores à direita, devolvemos essa árvore.

```
%calcular mínimo
min(t(X,nil,_),Ans):-
   Ans=X.
min(t(_,A,_),Ans):-
   min(A,Ans).

> min(t(4,t(2,t(1,nil,nil),t(3,nil,nil)),t(5,nil,t(7,t(6,nil,nil),nil))), R).
R = 1
```

O "min" recebe uma árvore binária e retorna o mínimo dessa árvore, que é primeiro elemento (o elemento mais à esquerda da árvore). A função devolve recursivamente a árvore mais à esquerda e assim que não houver mais árvores à esquerda, devolvemos essa árvore.

2.2. Converter numa Lista

```
converttolist(nil, []).
converttolist(t(X, L, R), Result) :- converttolist(L, L2), converttolist(R, L3), append(L2, [X], Aux), append(Aux, L3, Result).

> converttolist(t(a,t(b,t(d,nil,nil),t(e,nil,nil)),t(c,nil,t(f,t(g,nil,nil),nil))), R).
R = [d, b, e, a, c, g, f]
```

O "convertolist" recebe uma árvore binária e retorna a sua conversão em lista. Aplica-se recursividade à árvore da esquerda e à árvore da direita (separadamente), e vai dando "append" à lista.

2.3. Inserir

```
insert(Elem, nil, t(Elem, nil, nil)).
insert(Elem, t(Elem, L, R), t(Elem, L, R)).
```

```
insert(Elem, t(X, L, R), t(X, L2, R)) :- Elem < X, insert(Elem, L, L2), !.
insert(Elem, t(X, L, R), t(X, L, R2)) :- Elem > X, insert(Elem, R , R2), !.

> insert(8, t(4,t(2,t(1,nil,nil),t(3,nil,nil)),t(5,nil,t(7,t(6,nil,nil),nil))), R).
R = t(4,t(2,t(1,nil,nil),t(3,nil,nil)),t(5,nil,t(7,t(6,nil,nil),t(8,nil,nil))))
```

O “insert" recebe um valor e uma árvore binária e retorna a árvore com esse valor inserido. Se o nosso valor for menor que a raíz começamos por percorrer a sub-árvore da esquerda, se for maior, começamos por percorrer a sub-árvore da direita. Fazemos esta procura recursivamente até chegarmos ao termo vazio que é onde colocamos o nosso valor.