Universidade Federal do Paraná
Departamento de Informática

Disciplina:

Técnicas Alternativas de Programação

Alexandre I. Direne
E-mail: alexd@inf.ufpr.br
Web: http://www.inf.ufpr.br/~alexd

BIBLIOGRAFIA

- Ulf Nilsson and Jan Maluszynski. Logic, Programming and Prolog (2ed). Previously published by John Wiley & Sons Ltd. Now available free at: http://www.ida.liu.se/~ulfni/lpp/

- William F. Clocksin and C. S. Mellish. Programming in Prolog. Springer-Verlag, 1987.

- Elaine Rich, Kevin Knight. Artificial Intelligence. McGraw Hill, 1993.

- Patrick H. Winston, Artificial Intelligence, Second Edition, Addison-Wesley, 1993.

- Flávio Soares Corrêa da Silva, Marcelo Finger e Ana Cristina Vieira de Melo. Lógica para Computação. 1993. http://www.thomsonlearning.com.br/detalheLivro.do?id=104261

SOFTWARE

Compilador Prolog (Poplog). Endereço para obtenção:

http://www.cs.bham.ac.uk/research/poplog/freepoplog.html

Compilador Prolog (Swi-prolog). Endereço para obtenção:

http://www.swi-prolog.org

Téc. Altern. de Programação -- Programação Em Lógica

Lógica Proposicional ou Cálculo dos Enunciados

Enunciados (ou proposições) são expressões tomadas como verdade lógicas.

Enunciados:

- ***Simples:*** é todo enunciado que não contém nenhum outro como parte.

  Ex.: chove. (“chove” também é dito um ÁTOMO.)

- ***Compostos:*** é todo enunciado construídos a partir de enunciados simples utilizando os Conectivos Lógicos:

  Negação Lógica

  Ex.: ~ a

  NÃO chove

  Conjunção Lógica

  Ex.: a $\wedge$ b

  nublado E chove

  Disjunção Lógica

  Ex.: a $\vee$ b

  nublado OU chove (onde “OU” é um conectivo lógico)

  Implicação Lógica

  Ex.: a $\Rightarrow$ b

  SE chove ENTÃO nublado

  Equivalência Lógica

  Ex.: a $\Leftrightarrow$ b

  eleitor SE SOMENTE SE vota

Observação: É uma linguagem FORMAL ou seja, apenas FORMA (ou SINTAXE) importam.

Exemplo:

a $\Rightarrow$ a $\wedge$ b

SE chove ENTÃO chove $\wedge$ nublado.

b $\Rightarrow$ a $\wedge$ b

SE nublado ENTÃO chove $\wedge$ nublado (não retrata a realidade).

Logo, a $\Leftrightarrow$ a $\wedge$ b não retrata uma realidade. Porém, se tal enunciado constar em uma Base de Conhecimento, o mesmo será considerado como uma equivalência verdadeira, pois sua “FORMA” (ou sintaxe) está correta.

Formas Enunciativas

Tautologia: Enunciado sempre verdadeiro para qualquer variação de valor-verdade dos enunciados simples que o compõem.

Exemplo: a $\vee$ ¬a (chove OU NÃO chove)

Tabela de valores verdade:

a	**¬a**	**a $\vee$ ¬a**
V	F	V
F	V	V

Contradição: Enunciado sempre falso para qualquer variação de valor-verdade dos enunciados simples que o compõem.

Exemplo: a $\wedge$ ¬a (chove E NÃO chove)

Tabela de valores verdade:

a	**¬a**	**a $\wedge$ ¬a**
V	F	F
F	V	F

Contingência: Enunciado cujo valor-verdade só pode ser decidido mediante fatos, ou seja, os valor-verdade dos enunciados simples que o compõem.

Exemplo: a $\Rightarrow$ b (SE chove ENTÃO nublado)

Tabela de valores verdade:

a	**b**	**a $\Rightarrow$ b**
V	V	V
V	F	F
F	V	V
F	F	V

Regras de Inferência Dedutiva

Modus Ponens

$a \Rightarrow b$

a

$\therefore b$

Formas não válidas de Modus Ponens	
$a \Rightarrow b$	$a \Rightarrow b$
b	$\sim a$
$\therefore a$	$\therefore \sim b$

Exemplos:

(DADO)	$e \Rightarrow f$	SE chove ENTÃO nublado
(DADO)	$g \wedge e$	feriado E chove
(DEDUÇÃO)	f	nublado

Modus Tolens

$a \Rightarrow b$	*Exemplo:*	(DADO)	h
$\sim b$		(DADO)	$j \Rightarrow \sim h$
$\therefore \sim a$		(DEDUÇÃO)	$\sim j$

Silogismo Hipotético

$a \Rightarrow b$	*Exemplo:*	(DADO)	$\sim h \Rightarrow f$
$b \Rightarrow c$		(DADO)	$f \Rightarrow \sim e$
$\therefore a \Rightarrow c$		(DEDUÇÃO)	$\sim h \Rightarrow \sim e$

Silogismo Disjuntivo

$a \vee b$	*Exemplo:*	(DADO)	$\sim(c \Rightarrow d) \vee (h \Leftrightarrow g)$
$\sim a$		(DADO)	$c \Rightarrow d$
$\therefore b$		(DEDUÇÃO)	$h \Leftrightarrow g$

Simplificação (2 formas)

$a \wedge b$	$a \wedge b$
$\therefore a$	$\therefore b$

Exemplo:	(DADO)	$(a \wedge b) \wedge c$
	(DEDUÇÃO)	c
	(DEDUÇÃO)	$a \wedge b$
	(DEDUÇÃO)	a
	(DEDUÇÃO)	b

Conjunção

a

b

$\therefore a \wedge b$

Exemplo:	(DADO)	$a \wedge b$
	(DADO)	e
	(DADO)	c
	(DEDUÇÃO)	$e \wedge c$
	(DEDUÇÃO)	$(a \wedge b) \wedge (e \wedge c)$

Adição

a

$\therefore a \vee b$

Exemplo:	(DADO)	~h
	(DEDUÇÃO)	$\sim h \vee \sim(a \Rightarrow n)$

Lógica de Predicados

Termos : Constituem expressões que descrevem ou nomeiam entidades ou objetos.

- *Variáveis:*

  Em “x” é poeta romântico, x é uma VARIÁVEL e pode ser substituída por qualquer constante durante a avaliação de um predicado.

- *Constantes:*

  Em “Colombo descobriu a América”, “Colombo” é uma CONSTANTE.

- *Funções:*

  Em “Idade(x)”, “Idade” é uma FUNÇÃO que retorna uma valor inteiro resultante de um calculo especifico.

Predicados: Constituem nomes de propriedades e relações. No contexto do Cálculo dos Predicados, a articulação de um predicado com um termo formará a unidade significativa mínima.

Em “z é senador”, “senador” é um predicado, o qual avaliado, resulta em um valor-verdade. De outra forma, pode-se escrever:

senador (x)

Aridade de Predicados:

- chove - Aridade 0

- senador(x) - Aridade 1
- pai(x, Alberto) - Aridade 2

Instanciação: É o processo de avaliação de um predicado que consiste em substituir variáveis por constantes para a obtenção de enunciados singulares.

Por exemplo:

pai(x,y) e x = João e y = José temos a "instanciação" das variáveis x e y do predicado pai, resultando em pai (João,Jose).

Quantificação: É a aplicação de "quantificadores" ao processo de geração de enunciados a partir de funções enunciativas. Dois quantificadores são usados como enunciados no Calculo dos Predicados:

- Quantificador Universal ($\forall$)
- Quantificador Existencial ($\exists$)

Em "z é senador", "senador" é um predicado, o qual avaliado, resulta em um valor-verdade. De outra forma, pode-se escrever:

Quantificador Existencial

Significa dizer:

"existe pelo menos um x, tal que ... (..., x, ...) ..."

Exemplos:

Existe pelo menos um paulista aqui:

$\exists$ x (paulista(x))

Um dos números escolhidos é maior que 2:

$\exists$ x (x > 2)

Algum filosofo é cientista

$\exists$ x (filosofo(x) $\wedge$ cientista(x))

Existe pelo menos um indivíduo tal que este é filosofo e não é cientista (negação do enunciado acima):

$\exists$ x (filosofo(x) $\wedge$ ~cientista(x))

O enunciado existencial pode ser interpretado como:

$\exists$ x (p(x)) $\Leftrightarrow$ p(a1) $\vee$ p(a2) $\vee$ ... $\vee$ p(an)) onde U = { a1,a2, ..., an}

Quantificador Universal

Significa dizer:

"para todo x, ... (..., x, ...) ..."

Exemplos:

Tudo é material:

$\forall$ x (material(x))

Todo humano é criativo:

$\forall$ x (humano(x) $\Rightarrow$ criativo(x))

Nenhum humano é criativo (negação do enunciado acima):

$\forall$ x (humano(x) $\wedge$ ~criativo(x))

O enunciado universal pode ser interpretado como:

$\forall$ x (r(x)) $\Leftrightarrow$ r(a1) $\wedge$ r(a2) $\wedge$ ... $\wedge$ r(an)) onde U = { a1,a2, ..., an}

Observação: o conjunto U das constantes que podem ser usadas na "instanciação" da variável x deve ser finito

Alguns Enunciados:

- algo é feio
- algo não é triste
- alguns homens não são sinceros
- todo homem é mortal
- nada é incolor
- alguns escritores são realistas

Enunciados Contrários, Subcontrários e contraditórios:

Exemplos:

1. Toda criança é alegre: $\forall$ *x (criança(x)* $\Rightarrow$ *alegre(x))*
2. Alguma criança é alegre: $\exists$ *x (criança(x)* $\wedge$ *alegre(x))*
3. Nenhuma criança é alegre: $\forall$ *x (criança(x)* $\Rightarrow$ *~alegre(x))*
4. Alguma criança não é alegre: $\exists$ *x (criança(x)* $\wedge$ *~alegre(x))*

onde:

- 1 e 3 são contrários
- 1 e 4 são contraditórios
- 2 e 3 são contraditórios
- 2 e 4 são subcontrários

Problemas com Sentenças e Impactos impactos sobre Inferências na Lógica

Exemplo de aplicação "indevida" de inferência por Silogismo:

Alguns pássaros são animais que voam

$\exists$ x (pássaros(x) $\Rightarrow$ voa(x))

Alguns animais que voam são mamíferos

$\exists$ x (voa(x) $\Rightarrow$ mamíferos(x))

Dedução: Alguns pássaros são mamíferos

$\exists$ x (pássaro(x) $\Rightarrow$ mamíferos(x))

Programação em Lógica e Prolog

* Lógica
  - sobre OBJETOS
  - sobre SENTENÇAS

Conceitos inerentes a programação em lógica PROLOG:

* Método de busca em PROFUNDIDADE
* Combinado com BACKTRACKING (retroação)
* Algoritmo de UNIFICAÇÃO de predicados e seus termos (com INSTANCIAÇÃO de variáveis)
* Lógica de Predicados + Cálculo semântico (valor-de-verdade):
* FATOS (axiomas)
* REGRAS (implicação lógica: $b \Leftarrow a$)

* Forma CLAUSAL (ou de clausula):

$C1 \vee C2 \vee ... \vee Cn \Leftarrow P1 \wedge P2 \wedge ... Pm$

* Clausulas de HORN

Para n = 1 e m = 0 , resultando em FATOS.

C1

Para n = 1 e $m \geq 1$, resultando em REGRAS.

$C1 \Leftarrow P1 \wedge P2 \wedge ... Pm$

Variáveis em Prolog: São lacunas de memória que podem ou não ser instanciadas durante os ciclos do interpretador.

SINTAXE: Qualquer composição que inicie com uma letra maiúscula.

Ex.: Nome, X, Alberto

Constantes em Prolog: São porções fixas de memória que guardam valores.

SINTAXE: Qualquer composição que seja numérica, envolvida por aspas simples, ou que inicie com uma letra minúscula.

Ex.: alberto, x, 'Alberto'

Estruturas em Prolog: São elementos que, quando processados, devolvem um valor de um dado tipo (e não um valor verdade).

SINTAXE: Qualquer composição que inicie por letra maiúscula, minúscula, ou símbolos especiais, seguido de parênteses e seus COMPONENTES.

Ex.: livro('Inteligência Artificial', autor('Elaine', 'Rich'))

Predicados em Prolog: Expressam relações entre termos e sempre resultam em um valor verdade (VERDADEIRO ou FALSO).

SINTAXE: Qualquer composição que inicie por letra maiúscula, minúscula, ou símbolos especiais, seguido de parênteses e seus TERMOS.

Ex.: corre(X,y)

Exemplos de FATOS em Prolog:

```
Pai(joao, jose).
pegou(maria, ligeirinho, centro).
```

Exemplos de REGRAS em Prolog:

```
tem_epatite(X) :-
        tem_febre(X),
        esta_amarelo(X).
```

Adicionando e Retirando Fatos

```
homem(jose).
homem(mario).
homem(alberto).
homem(fabricio).

?- asserta(homem(alex)).
yes

?- assertz(homem(rogerio)).
yes

?- listing(homem).
homem(alex).
homem(jose).
homem(mario).
homem(alberto).
homem(fabricio).
homem(rogerio).

yes

?- assert(homem(jonatas)).
yes

?- assert(homem(jonatas)).
yes

?- listing(homem).
homem(alex).
homem(jose).
homem(mario).
homem(alberto).
homem(fabricio).
homem(rogerio).
homem(jonatas).
homem(jonatas).

yes

?- retract(homem(jose)).
yes
```

```
?- listing(homem).
homem(alex).
homem(mario).
homem(alberto).
homem(fabricio).
homem(rogerio).
homem(jonatas).
homem(jonatas).

yes

?- retract(homem(jose)).
no

?- retractall(homem(_)).
yes
```

Compondo Fatos e Regras

```
homem(joao).
homem(jose).
homem(mario).
homem(alberto).
homem(fabricio).
homem(jonatas).
homem(marcos).

mulher(claudia).
mulher(silvia).
mulher(clara).
mulher(angela).
mulher(maria).
mulher(marcia).

pai(joao, jose).
pai(joao, maria).
mae(silvia, jose).
mae(silvia, maria).
pai(alberto, joao).
pai(fabricio, alberto).
pai(jonatas, fabricio).

irmao(X, Y) :-
        homem(X),
        pai(Z, X),
        pai(Z, Y),
        mae(Z1, X),
        mae(Z1, Y).

?- irmao(jose, joao).
** (1) Call : irmao(jose, joao)?
** (2) Call : homem(jose)?
** (2) Exit : homem(jose)?
** (3) Call : pai(_1, jose)?
** (3) Exit : pai(joao, jose)?
** (4) Call : pai(joao, joao)?
** (4) Fail : pai(joao, joao)?
** (3) Redo : pai(joao, jose)?
** (3) Fail : pai(_1, jose)?
** (2) Redo : homem(jose)?
** (2) Fail : homem(jose)?
```

```
** (1) Fail : irmao(jose, joao)?
no

?- irmao(jose, maria).
** (1) Call : irmao(jose, maria)?
** (2) Call : homem(jose)?
** (2) Exit : homem(jose)?
** (3) Call : pai(_1, jose)?
** (3) Exit : pai(joao, jose)?
** (4) Call : pai(joao, maria)?
** (4) Exit : pai(joao, maria)?
** (5) Call : mae(_2, jose)?
** (5) Exit : mae(silvia, jose)?
** (6) Call : mae(silvia, maria)?
** (6) Exit : mae(silvia, maria)?
** (1) Exit : irmao(jose, maria)?
yes

?- irmao(jose, jose).
** (1) Call : irmao(jose, jose)?
** (2) Call : homem(jose)?
** (2) Exit : homem(jose)?
** (3) Call : pai(_1, jose)?
** (3) Exit : pai(joao, jose)?
** (4) Call : pai(joao, jose)?
** (4) Exit : pai(joao, jose)?
** (5) Call : mae(_2, jose)?
** (5) Exit : mae(silvia, jose)?
** (6) Call : mae(silvia, jose)?
** (6) Exit : mae(silvia, jose)?
** (1) Exit : irmao(jose, jose)?
yes
irmao(X, Y) :-
        homem(X),
        pai(Z, X),
        pai(Z, Y),
        mae(Z1, X),
        mae(Z1, Y),
        not(X = Y).

?- irmao(jose, jose).
** (1) Call : irmao(jose, jose)?
** (2) Call : homem(jose)?
** (2) Exit : homem(jose)?
** (3) Call : pai(_1, jose)?
** (3) Exit : pai(joao, jose)?
** (4) Call : pai(joao, jose)?
** (4) Exit : pai(joao, jose)?
** (5) Call : mae(_2, jose)?
** (5) Exit : mae(silvia, jose)?
** (6) Call : mae(silvia, jose)?
** (6) Exit : mae(silvia, jose)?
** (6) Redo : mae(silvia, jose)?
** (6) Fail : mae(silvia, jose)?
** (5) Redo : mae(silvia, jose)?
** (5) Fail : mae(_2, jose)?
** (4) Redo : pai(joao, jose)?
** (4) Fail : pai(joao, jose)?
** (3) Redo : pai(joao, jose)?
** (3) Fail : pai(_1, jose)?
** (2) Redo : homem(jose)?
** (2) Fail : homem(jose)?
```

```
** (1) Fail : irmao(jose, jose)?
no

irma(X, Y) :-
        mulher(X),
        pai(Z, X),
        pai(Z, Y),
        mae(Z1, X),
        mae(Z1, Y),
        not(X = Y).
```

Um predicado recursivo (aplicando conceitos de indução - passo e base):

```
gerou(X, Y) :-
        pai(X, Y).
gerou(X, Y) :-
        mae(X, Y).

ancestral(X, Y) :-
        gerou(X, Y).

ancestral(X, Y) :-
        gerou(Z, Y),
        ancestral(X, Z).

?- ancestral(fabricio,alberto).
** (1) Call : ancestral(fabricio, alberto)?
** (2) Call : gerou(fabricio, alberto)?
** (3) Call : pai(fabricio, alberto)?
** (3) Exit : pai(fabricio, alberto)?
** (2) Exit : gerou(fabricio, alberto)?
** (1) Exit : ancestral(fabricio, alberto)?
yes

?- ancestral(fabricio,joao).
** (1) Call : ancestral(fabricio, joao)?
** (2) Call : gerou(fabricio, joao)?
** (3) Call : pai(fabricio, joao)?
** (3) Fail : pai(fabricio, joao)?
** (4) Call : mae(fabricio, joao)?
** (4) Fail : mae(fabricio, joao)?
** (2) Fail : gerou(fabricio, joao)?
** (5) Call : gerou(_1, joao)?
** (6) Call : pai(_1, joao)?
** (6) Exit : pai(alberto, joao)?
** (5) Exit : gerou(alberto, joao)?
** (7) Call : ancestral(fabricio, alberto)?
** (8) Call : gerou(fabricio, alberto)?
** (9) Call : pai(fabricio, alberto)?
** (9) Exit : pai(fabricio, alberto)?
** (8) Exit : gerou(fabricio, alberto)?
** (7) Exit : ancestral(fabricio, alberto)?
** (1) Exit : ancestral(fabricio, joao)?
yes

?- ancestral(fabricio,silvia).
** (1) Call : ancestral(fabricio, silvia)?
** (2) Call : gerou(fabricio, silvia)?
** (3) Call : pai(fabricio, silvia)?
** (3) Fail : pai(fabricio, silvia)?
```

```
** (4) Call : mae(fabricio, silvia)?
** (4) Fail : mae(fabricio, silvia)?
** (2) Fail : gerou(fabricio, silvia)?
** (5) Call : gerou(_1, silvia)?
** (6) Call : pai(_1, silvia)?
** (6) Fail : pai(_1, silvia)?
** (7) Call : mae(_1, silvia)?
** (7) Fail : mae(_1, silvia)?
** (5) Fail : gerou(_1, silvia)?
** (1) Fail : ancestral(fabricio, silvia)?
no

?- ancestral(fabricio,maria).
** (1) Call : ancestral(fabricio, maria)?
** (2) Call : gerou(fabricio, maria)?
** (3) Call : pai(fabricio, maria)?
** (3) Fail : pai(fabricio, maria)?
** (4) Call : mae(fabricio, maria)?
** (4) Fail : mae(fabricio, maria)?
** (2) Fail : gerou(fabricio, maria)?
. . .

** (14) Call : gerou(fabricio, alberto)?
** (15) Call : pai(fabricio, alberto)?
** (15) Exit : pai(fabricio, alberto)?
** (14) Exit : gerou(fabricio, alberto)?
** (13) Exit : ancestral(fabricio, alberto)?
** (7) Exit : ancestral(fabricio, joao)?
** (1) Exit : ancestral(fabricio, maria)?
yes
```

Uma definição incompleta de FATORIAL(N):

```
fatorial(1, 1).

fatorial(X, FatX) :-
        Y is X - 1,
        fatorial(Y, FatY),
        FatX is X * FatY.


?- fatorial(1, Z).
** (1) Call : fatorial(1, _1)?
** (1) Exit : fatorial(1, 1)?
Z = 1 ?
yes

?- fatorial(2, Z).
** (1) Call : fatorial(2, _1)?
** (2) Call : fatorial(1, _2)?
** (2) Exit : fatorial(1, 1)?
** (1) Exit : fatorial(2, 2)?
Z = 2 ?
yes

?- fatorial(3, Z).
** (1) Call : fatorial(3, _1)?
** (2) Call : fatorial(2, _2)?
** (3) Call : fatorial(1, _3)?
** (3) Exit : fatorial(1, 1)?
** (2) Exit : fatorial(2, 2)?
```

```
** (1) Exit : fatorial(3, 6)?
Z = 6 ?
yes


?- fatorial(4, Z).
** (1) Call : fatorial(4, _1)?
** (2) Call : fatorial(3, _2)?
** (3) Call : fatorial(2, _3)?
** (4) Call : fatorial(1, _4)?
** (4) Exit : fatorial(1, 1)?
** (3) Exit : fatorial(2, 2)?
** (2) Exit : fatorial(3, 6)?
** (1) Exit : fatorial(4, 24)?
Z = 24 ?
yes

?- fatorial(5, Z).
** (1) Call : fatorial(5, _1)?
** (2) Call : fatorial(4, _2)?
** (3) Call : fatorial(3, _3)?
** (4) Call : fatorial(2, _4)?
** (5) Call : fatorial(1, _5)?
** (5) Exit : fatorial(1, 1)?
** (4) Exit : fatorial(2, 2)?
** (3) Exit : fatorial(3, 6)?
** (2) Exit : fatorial(4, 24)?
** (1) Exit : fatorial(5, 120)?
Z = 120 ?
yes
```

Listas Dinâmicas em Prolog

EXEMPLOS DE LISTAS:

```
[ banana, laranja, abacaxi ]
[ cavalo, X, carneiro ]
[ data(3,dezembro,1960), data(11,agosto,1958) ]
[ [a, b], [c, d] ]
partes_de(carro, [ lataria, chassi, interior ] ).
partes_de(lataria, [ paralamas, portas, capo, porta_mala ] ).
[ banana, laranja | X ]
[ X | Y ]
[ Cabeca | Cauda ]
[ Primeiro | Resto ]
```

Manipulação de Listas com Qualquer Número de Elementos

```
?- X is 2 + 5 * 3.0/4.0.
X = 5.75 ?
yes

numero_de_elementos([], 0).

numero_de_elementos([X | Y], N) :-
        numero_de_elementos(Y, NY),
        N is 1 + NY.

?- numero_de_elementos([a,b,c,d], X).
** (1) Call : numero_de_elementos([a, b, c, d], _1)?
```

```
** (2) Call : numero_de_elementos([b, c, d], _2)?
** (3) Call : numero_de_elementos([c, d], _3)?
** (4) Call : numero_de_elementos([d], _4)?
** (5) Call : numero_de_elementos([], _5)?
** (5) Exit : numero_de_elementos([], 0)?
** (4) Exit : numero_de_elementos([d], 1)?
** (3) Exit : numero_de_elementos([c, d], 2)?
** (2) Exit : numero_de_elementos([b, c, d], 3)?
** (1) Exit : numero_de_elementos([a, b, c, d], 4)?
X = 4 ?
yes

somatoria([], 0).

somatoria([X | Y], S) :-
        somatoria(Y, SY),
        S is X + SY.

?- somatoria([1, 2, 3, 4], X).
** (1) Call : somatoria([1, 2, 3, 4], _1)?
** (2) Call : somatoria([2, 3, 4], _2)?
** (3) Call : somatoria([3, 4], _3)?
** (4) Call : somatoria([4], _4)?
** (5) Call : somatoria([], _5)?
** (5) Exit : somatoria([], 0)?
** (4) Exit : somatoria([4], 4)?
** (3) Exit : somatoria([3, 4], 7)?
** (2) Exit : somatoria([2, 3, 4], 9)?
** (1) Exit : somatoria([1, 2, 3, 4], 10)?
X = 10 ?
yes

        media(Lista, Media) :-
                numero_de_elementos(Lista, Num_elem),
                somatoria(Lista, S),
                Media is S / Num_elem.

?- media([1, 2, 3, 4], X).
* (1) Call : media([1, 2, 3, 4], _1)?
** (2) Call : numero_de_elementos([1, 2, 3, 4], _2)?
** (3) Call : numero_de_elementos([2, 3, 4], _3)?
** (4) Call : numero_de_elementos([3, 4], _4)?
** (5) Call : numero_de_elementos([4], _5)?
** (6) Call : numero_de_elementos([], _6)?
** (6) Exit : numero_de_elementos([], 0)?
** (5) Exit : numero_de_elementos([4], 1)?
** (4) Exit : numero_de_elementos([3, 4], 2)?
** (3) Exit : numero_de_elementos([2, 3, 4], 3)?
** (2) Exit : numero_de_elementos([1, 2, 3, 4], 4)?
** (7) Call : somatoria([1, 2, 3, 4], _7)?
** (8) Call : somatoria([2, 3, 4], _8)?
** (9) Call : somatoria([3, 4], _9)?
** (10) Call : somatoria([4], _10)?
** (11) Call : somatoria([], _11)?
** (11) Exit : somatoria([], 0)?
** (10) Exit : somatoria([4], 4)?
** (9) Exit : somatoria([3, 4], 7)?
** (8) Exit : somatoria([2, 3, 4], 9)?
** (7) Exit : somatoria([1, 2, 3, 4], 10)?
** (1) Exit : media([1, 2, 3, 4], 2.5)?
X = 2.5 ?
yes
```

Concatenação de 2 Listas com Qualquer Número de Elementos

```
concatenadas([], L, L).

concatenadas([X | Cauda1], L2, [X | Cauda3]) :-
        concatenadas(Cauda1, L2, Cauda3).

?- concatenadas([o1, o2, o3], [o4, o5, o6], X).
** (1) Call : concatenadas([o1, o2, o3], [o4, o5, o6], _1)?
** (2) Call : concatenadas([o2, o3], [o4, o5, o6], _2)?
** (3) Call : concatenadas([o3], [o4, o5, o6], _3)?
** (4) Call : concatenadas([], [o4, o5, o6], _4)?
** (4) Exit : concatenadas([], [o4, o5, o6], [o4, o5, o6])?
** (3) Exit : concatenadas([o3], [o4, o5, o6], [o3, o4, o5, o6])?
** (2) Exit : concatenadas([o2, o3], [o4, o5, o6], [o2, o3, o4, o5, o6])?
** (1) Exit : concatenadas([o1, o2, o3], [o4, o5, o6], [o1, o2, o3, o4, o5, o6])?
X = [o1, o2, o3, o4, o5, o6] ?
yes

?- concatenadas(X,Y, [a,b,c,d,e]).
X = []
Y = [a, b, c, d, e] ? y

X = [a]
Y = [b, c, d, e] ? y

X = [a, b]
Y = [c, d, e] ? y

X = [a, b, c]
Y = [d, e] ? y

X = [a, b, c, d]
Y = [e] ? y

X = [a, b, c, d, e]
Y = [] ? y
no

?- concatenadas(X,X, [a,b,c,d,e]).
no

?- concatenadas(X,X, [a,b,c,d,e]).
** (1) Call : concatenadas(_1, _1, [a, b, c, d, e])?
** (2) Call : concatenadas(_2, [a | _2], [b, c, d, e])?
** (3) Call : concatenadas(_3, [a, b | _3], [c, d, e])?
** (4) Call : concatenadas(_4, [a, b, c | _4], [d, e])?
** (5) Call : concatenadas(_5, [a, b, c, d | _5], [e])?
** (6) Call : concatenadas(_6, [a, b, c, d, e | _6], [])?
** (6) Fail : concatenadas(_6, [a, b, c, d, e | _6], [])?
** (5) Fail : concatenadas(_5, [a, b, c, d | _5], [e])?
** (4) Fail : concatenadas(_4, [a, b, c | _4], [d, e])?
** (3) Fail : concatenadas(_3, [a, b | _3], [c, d, e])?
** (2) Fail : concatenadas(_2, [a | _2], [b, c, d, e])?
** (1) Fail : concatenadas(_1, _1, [a, b, c, d, e])?
no
```

Manipulação de Listas com Qualquer Número de Elementos

```
pertence_a(X, [X | Cauda]).
```

```
pertence_a(X, [Cabeca | Cauda]) :-
        pertence_a(X, Cauda).

?- pertence_a(5, [5, 1, 2, 3, 4, 6]).
** (1) Call : pertence_a(5, [5, 1, 2, 3, 4, 6])?
** (1) Exit : pertence_a(5, [5, 1, 2, 3, 4, 6])?
yes

?- pertence_a(5, [1, 2, 3, 4, 5, 6]).
** (1) Call : pertence_a(5, [1, 2, 3, 4, 5, 6])?
** (2) Call : pertence_a(5, [2, 3, 4, 5, 6])?
** (3) Call : pertence_a(5, [3, 4, 5, 6])?
** (4) Call : pertence_a(5, [4, 5, 6])?
** (5) Call : pertence_a(5, [5, 6])?
** (5) Exit : pertence_a(5, [5, 6])?
** (4) Exit : pertence_a(5, [4, 5, 6])?
** (3) Exit : pertence_a(5, [3, 4, 5, 6])?
** (2) Exit : pertence_a(5, [2, 3, 4, 5, 6])?
** (1) Exit : pertence_a(5, [1, 2, 3, 4, 5, 6])?
yes

?- pertence_a(5, [1, 2, 3, 4, 6, 5]).
** (1) Call : pertence_a(5, [1, 2, 3, 4, 6, 5])?
** (2) Call : pertence_a(5, [2, 3, 4, 6, 5])?
** (3) Call : pertence_a(5, [3, 4, 6, 5])?
** (4) Call : pertence_a(5, [4, 6, 5])?
** (5) Call : pertence_a(5, [6, 5])?
** (6) Call : pertence_a(5, [5])?
** (6) Exit : pertence_a(5, [5])?
** (5) Exit : pertence_a(5, [6, 5])?
** (4) Exit : pertence_a(5, [4, 6, 5])?
** (3) Exit : pertence_a(5, [3, 4, 6, 5])?
** (2) Exit : pertence_a(5, [2, 3, 4, 6, 5])?
** (1) Exit : pertence_a(5, [1, 2, 3, 4, 6, 5])?
yes
?- pertence_a(5, [1, 2, 3, 4, 6]).
** (1) Call : pertence_a(5, [1, 2, 3, 4, 6])?
** (2) Call : pertence_a(5, [2, 3, 4, 6])?
** (3) Call : pertence_a(5, [3, 4, 6])?
** (4) Call : pertence_a(5, [4, 6])?
** (5) Call : pertence_a(5, [6])?
** (6) Call : pertence_a(5, [])?
** (6) Fail : pertence_a(5, [])?
** (5) Fail : pertence_a(5, [6])?
** (4) Fail : pertence_a(5, [4, 6])?
** (3) Fail : pertence_a(5, [3, 4, 6])?
** (2) Fail : pertence_a(5, [2, 3, 4, 6])?
** (1) Fail : pertence_a(5, [1, 2, 3, 4, 6])?
no
```

Monitorando o predicado concatenadas para os dois primeiros termos livres:

```
?- concatenadas(X,Y, [a,b,c,d,e]).
** (1) Call : concatenadas(_1, _2, [a, b, c, d, e])?
** (1) Exit : concatenadas([], [a, b, c, d, e], [a, b, c, d, e])?
X = []
Y = [a, b, c, d, e] ? y

** (1) Redo : concatenadas([], [a, b, c, d, e], [a, b, c, d, e])?
** (2) Call : concatenadas(_3, _2, [b, c, d, e])?
** (2) Exit : concatenadas([], [b, c, d, e], [b, c, d, e])?
```

```
** (1) Exit : concatenadas([a], [b, c, d, e], [a, b, c, d, e])?
X = [a]
Y = [b, c, d, e] ? y

** (1) Redo : concatenadas([a], [b, c, d, e], [a, b, c, d, e])?
** (2) Redo : concatenadas([], [b, c, d, e], [b, c, d, e])?
** (3) Call : concatenadas(_4, _2, [c, d, e])?
** (3) Exit : concatenadas([], [c, d, e], [c, d, e])?
** (2) Exit : concatenadas([b], [c, d, e], [b, c, d, e])?
** (1) Exit : concatenadas([a, b], [c, d, e], [a, b, c, d, e])?
X = [a, b]
Y = [c, d, e] ? y

** (1) Redo : concatenadas([a, b], [c, d, e], [a, b, c, d, e])?
** (2) Redo : concatenadas([b], [c, d, e], [b, c, d, e])?
** (3) Redo : concatenadas([], [c, d, e], [c, d, e])?
** (4) Call : concatenadas(_5, _2, [d, e])?
** (4) Exit : concatenadas([], [d, e], [d, e])?
** (3) Exit : concatenadas([c], [d, e], [c, d, e])?
** (2) Exit : concatenadas([b, c], [d, e], [b, c, d, e])?
** (1) Exit : concatenadas([a, b, c], [d, e], [a, b, c, d, e])?
X = [a, b, c]
Y = [d, e] ? y

** (1) Redo : concatenadas([a, b, c], [d, e], [a, b, c, d, e])?
** (2) Redo : concatenadas([b, c], [d, e], [b, c, d, e])?
** (3) Redo : concatenadas([c], [d, e], [c, d, e])?
** (4) Redo : concatenadas([], [d, e], [d, e])?
** (5) Call : concatenadas(_6, _2, [e])?
** (5) Exit : concatenadas([], [e], [e])?
** (4) Exit : concatenadas([d], [e], [d, e])?
** (3) Exit : concatenadas([c, d], [e], [c, d, e])?
** (2) Exit : concatenadas([b, c, d], [e], [b, c, d, e])?
** (1) Exit : concatenadas([a, b, c, d], [e], [a, b, c, d, e])?
X = [a, b, c, d]
Y = [e] ? y

** (1) Redo : concatenadas([a, b, c, d], [e], [a, b, c, d, e])?
** (2) Redo : concatenadas([b, c, d], [e], [b, c, d, e])?
** (3) Redo : concatenadas([c, d], [e], [c, d, e])?
** (4) Redo : concatenadas([d], [e], [d, e])?
** (5) Redo : concatenadas([], [e], [e])?
** (6) Call : concatenadas(_7, _2, [])?
** (6) Exit : concatenadas([], [], [])?
** (5) Exit : concatenadas([e], [], [e])?
** (4) Exit : concatenadas([d, e], [], [d, e])?
** (3) Exit : concatenadas([c, d, e], [], [c, d, e])?
** (2) Exit : concatenadas([b, c, d, e], [], [b, c, d, e])?
** (1) Exit : concatenadas([a, b, c, d, e], [], [a, b, c, d, e])?
X = [a, b, c, d, e]
Y = [] ? y
no

?- concatenadas(X,X, [a,b,c,a,b,c]).
** (1) Call : concatenadas(_1, _1, [a, b, c, a, b, c])?
** (2) Call : concatenadas(_2, [a | _2], [b, c, a, b, c])?
** (3) Call : concatenadas(_3, [a, b | _3], [c, a, b, c])?
** (4) Call : concatenadas(_4, [a, b, c | _4], [a, b, c])?
** (4) Exit : concatenadas([], [a, b, c], [a, b, c])?
** (3) Exit : concatenadas([c], [a, b, c], [c, a, b, c])?
** (2) Exit : concatenadas([b, c], [a, b, c], [b, c, a, b, c])?
** (1) Exit : concatenadas([a, b, c], [a, b, c], [a, b, c, a, b, c])?
```

```
X = [a, b, c] ? y

** (1) Redo : concatenadas([a, b, c], [a, b, c], [a, b, c, a, b, c])?
** (2) Redo : concatenadas([b, c], [a, b, c], [b, c, a, b, c])?
** (3) Redo : concatenadas([c], [a, b, c], [c, a, b, c])?
** (4) Redo : concatenadas([], [a, b, c], [a, b, c])?
** (5) Call : concatenadas(_5, [a, b, c, a | _5], [b, c])?
** (6) Call : concatenadas(_6, [a, b, c, a, b | _6], [c])?
** (7) Call : concatenadas(_7, [a, b, c, a, b, c | _7], [])?
** (7) Fail : concatenadas(_7, [a, b, c, a, b, c | _7], [])?
** (6) Fail : concatenadas(_6, [a, b, c, a, b | _6], [c])?
** (5) Fail : concatenadas(_5, [a, b, c, a | _5], [b, c])?
** (4) Fail : concatenadas(_4, [a, b, c | _4], [a, b, c])?
** (3) Fail : concatenadas(_3, [a, b | _3], [c, a, b, c])?
** (2) Fail : concatenadas(_2, [a | _2], [b, c, a, b, c])?
** (1) Fail : concatenadas(_1, _1, [a, b, c, a, b, c])?

no
```

Exercícios Sobre Manipulção de Lista em PROLOG

1. Construir um predicado em Prolog, denominado "listificada", cujo comportamento é o expresso abaixo:

```
listificada([], []).

listificada([X | L1], [ [X] | L2 ]) :-
        listificada(L1, L2).

?- listificada([], Y).
** (1) Call : listificada([], _1)?
** (1) Exit : listificada([], [])?
Y = [] ?
yes

?- listificada([a], Y).
** (1) Call : listificada([a], _1)?
** (2) Call : listificada([], _2)?
** (2) Exit : listificada([], [])?
** (1) Exit : listificada([a], [[a]])?
Y = [[a]] ?
yes

?- listificada([a, b, c, d], Y).
** (1) Call : listificada([a, b, c, d], _1)?
** (2) Call : listificada([b, c, d], _2)?
** (3) Call : listificada([c, d], _3)?
** (4) Call : listificada([d], _4)?
** (5) Call : listificada([], _5)?
** (5) Exit : listificada([], [])?
** (4) Exit : listificada([d], [[d]])?
** (3) Exit : listificada([c, d], [[c], [d]])?
** (2) Exit : listificada([b, c, d], [[b], [c], [d]])?
** (1) Exit : listificada([a, b, c, d], [[a], [b], [c], [d]])?
Y = [[a], [b], [c], [d]] ?
yes
```

2. Construir um predicado em Prolog, denominado "lista_dobro", cujo comportamento é o expresso abaixo:

```
lista_dobro([], []).

lista_dobro([X | L1], [ Y | L2 ]) :-
        Y is 2 * X,
        lista_dobro(L1, L2).

?- lista_dobro([], Z).
** (1) Call : lista_dobro([], _1)?
** (1) Exit : lista_dobro([], [])?
Z = [] ?
yes

?- lista_dobro([7], Z).
** (1) Call : lista_dobro([7], _1)?
** (2) Call : lista_dobro([], _2)?
** (2) Exit : lista_dobro([], [])?
** (1) Exit : lista_dobro([7], [14])?
Z = [14] ?
yes

?- lista_dobro([2, 4, 6], Z).
** (1) Call : lista_dobro([2, 4, 6], _1)?
** (2) Call : lista_dobro([4, 6], _2)?
** (3) Call : lista_dobro([6], _3)?
** (4) Call : lista_dobro([], _4)?
** (4) Exit : lista_dobro([], [])?
** (3) Exit : lista_dobro([6], [12])?
** (2) Exit : lista_dobro([4, 6], [8, 12])?
** (1) Exit : lista_dobro([2, 4, 6], [4, 8, 12])?
Z = [4, 8, 12] ?
yes
```

3. Construir um predicado em Prolog, denominado "num_ocorrencias", cujo comportamento é o expresso abaixo:

```
num_ocorrencias(Palavra, [], 0).
num_ocorrencias(Palavra, [Palavra | Cauda], Num_ocorrencias) :-
        num_ocorrencias(Palavra, Cauda, Num_cauda),
        Num_ocorrencias is Num_cauda + 1.

num_ocorrencias(Palavra, [Cabeca | Cauda], Num_ocorrencias) :-
        num_ocorrencias(Palavra, Cauda, Num_ocorrencias).

?- num_ocorrencias(a, [], N).
** (1) Call : num_ocorrencias(a, [], _1)?
** (1) Exit : num_ocorrencias(a, [], 0)?
N = 0 ?
yes

?- num_ocorrencias(a, [a], N).
** (1) Call : num_ocorrencias(a, [a], _1)?
** (2) Call : num_ocorrencias(a, [], _2)?
** (2) Exit : num_ocorrencias(a, [], 0)?
** (1) Exit : num_ocorrencias(a, [a], 1)?
N = 1 ?
yes

?- num_ocorrencias(a, [a, menina], N).
** (1) Call : num_ocorrencias(a, [a, menina], _1)?
** (2) Call : num_ocorrencias(a, [menina], _2)?
** (3) Call : num_ocorrencias(a, [], _2)?
```

```
** (3) Exit : num_ocorrencias(a, [], 0)?
** (2) Exit : num_ocorrencias(a, [menina], 0)?
** (1) Exit : num_ocorrencias(a, [a, menina], 1)?
N = 1 ?
yes

?- num_ocorrencias(a, [a, menina, viu, a, uva], N).
** (1) Call : num_ocorrencias(a, [a, menina, viu, a, uva], _1)?
** (2) Call : num_ocorrencias(a, [menina, viu, a, uva], _2)?
** (3) Call : num_ocorrencias(a, [viu, a, uva], _2)?
** (4) Call : num_ocorrencias(a, [a, uva], _2)?
** (5) Call : num_ocorrencias(a, [uva], _3)?
** (6) Call : num_ocorrencias(a, [], _3)?
** (6) Exit : num_ocorrencias(a, [], 0)?
** (5) Exit : num_ocorrencias(a, [uva], 0)?
** (4) Exit : num_ocorrencias(a, [a, uva], 1)?
** (3) Exit : num_ocorrencias(a, [viu, a, uva], 1)?
** (2) Exit : num_ocorrencias(a, [menina, viu, a, uva], 1)?
** (1) Exit : num_ocorrencias(a, [a, menina, viu, a, uva], 2)?
N = 2 ?
yes

?- num_ocorrencias(menino, [o, menino, espantou, o, gato], N).
** (1) Call : num_ocorrencias(menino, [o, menino, espantou, o, gato], _1)?
** (2) Call : num_ocorrencias(menino, [menino, espantou, o, gato], _1)?
** (3) Call : num_ocorrencias(menino, [espantou, o, gato], _2)?
** (4) Call : num_ocorrencias(menino, [o, gato], _2)?
** (5) Call : num_ocorrencias(menino, [gato], _2)?
** (6) Call : num_ocorrencias(menino, [], _2)?
** (6) Exit : num_ocorrencias(menino, [], 0)?
** (5) Exit : num_ocorrencias(menino, [gato], 0)?
** (4) Exit : num_ocorrencias(menino, [o, gato], 0)?
** (3) Exit : num_ocorrencias(menino, [espantou, o, gato], 0)?
** (2) Exit : num_ocorrencias(menino, [menino, espantou, o, gato], 1)?
** (1) Exit : num_ocorrencias(menino, [o, menino, espantou, o, gato], 1)?
N = 1 ?
yes
```

4. Construir um predicado em Prolog, denominado "todas_antes", o qual relaciona uma lista de palavras com uma sub-lista de todas as palavras da lista que precedem uma dada palavra. Seu comportamento é o expresso abaixo:

```
todas_antes(Palavra, [], []).

todas_antes(Palavra, [ Palavra | _ ], []).

todas_antes(Palavra, [ Cabeca | Cauda ], [Cabeca | Sub_Lista_1]) :-
        todas_antes(Palavra, Cauda, Sub_Lista_1).


?- todas_antes(casaco, [], L).
** (1) Call : todas_antes(casaco, [], _1)?
** (1) Exit : todas_antes(casaco, [], [])?
L = [] ?
yes
?- todas_antes(casaco, [ casaco ], L).
** (1) Call : todas_antes(casaco, [casaco], _1)?
** (1) Exit : todas_antes(casaco, [casaco], [])?
L = [] ?
```

```
yes

?- todas_antes(casaco, [ casaco, de, pele ], L).
** (1) Call : todas_antes(casaco, [casaco, de, pele], _1)?
** (1) Exit : todas_antes(casaco, [casaco, de, pele], [])?
L = [] ?
yes

?- todas_antes(abriu, [o, homem, abriu, a, porta], L).
** (1) Call : todas_antes(abriu, [o, homem, abriu, a, porta], _1)?
** (2) Call : todas_antes(abriu, [homem, abriu, a, porta], _2)?
** (3) Call : todas_antes(abriu, [abriu, a, porta], _3)?
** (3) Exit : todas_antes(abriu, [abriu, a, porta], [])?
** (2) Exit : todas_antes(abriu, [homem, abriu, a, porta], [homem])?
** (1) Exit : todas_antes(abriu, [o, homem, abriu, a, porta], [o, homem])?
L = [o, homem] ?
yes

?- todas_antes(casaco, [o, homem, abriu, a, porta], L).
** (1) Call : todas_antes(casaco, [o, homem, abriu, a, porta], _1)?
** (2) Call : todas_antes(casaco, [homem, abriu, a, porta], _2)?
** (3) Call : todas_antes(casaco, [abriu, a, porta], _3)?
** (4) Call : todas_antes(casaco, [a, porta], _4)?
** (5) Call : todas_antes(casaco, [porta], _5)?
** (6) Call : todas_antes(casaco, [], _6)?
** (6) Exit : todas_antes(casaco, [], [])?
** (5) Exit : todas_antes(casaco, [porta], [porta])?
** (4) Exit : todas_antes(casaco, [a, porta], [a, porta])?
** (3) Exit : todas_antes(casaco, [abriu, a, porta], [abriu, a, porta])?
** (2) Exit : todas_antes(casaco, [homem, abriu, a, porta], [homem, abriu, a, porta])?
** (1) Exit : todas_antes(casaco, [o, homem, abriu, a, porta], [o, homem, abriu, a, porta])?
L = [o, homem, abriu, a, porta] ?
yes
```

***** REFORMULANDO A PRIMEIRA CLAUSULA *****

```
todas_antes(Palavra, Lista, []) :-
        not(pertence_a(Palavra, Lista)).

todas_antes(Palavra, [ Palavra | _ ], []).

todas_antes(Palavra, [ Cabeca | Cauda ], [Cabeca | Sub_Lista_1]) :-
        todas_antes(Palavra, Cauda, Sub_Lista_1).

?- todas_antes(casaco, [o, homem, abriu, a, porta], L).
** (1) Call : todas_antes(casaco, [o, homem, abriu, a, porta], _1)?
** (1) Exit : todas_antes(casaco, [o, homem, abriu, a, porta], [])?
L = [] ?
yes
```

5. Construir um predicado em Prolog, denominado "todas_depois", o qual relaciona uma lista de palavras com uma sub-lista de todas as palavras da lista que se sucedem à dada palavra. Seu comportamento é o expresso abaixo:

```
todas_depois(Palavra, Lista, []) :-
        not(pertence_a(Palavra, Lista)).

todas_depois(Palavra, [ Palavra | Cauda ], Cauda).

todas_depois(Palavra, [ Cabeca | Cauda ], Sub_Lista_1) :-
        todas_depois(Palavra, Cauda, Sub_Lista_1).
```

```
?- todas_depois(casaco, [], L).
** (1) Call : todas_depois(casaco, [], _1)?
** (1) Exit : todas_depois(casaco, [], [])?
L = [] ?
yes

?- todas_depois(casaco, [ casaco ], L).
** (1) Call : todas_depois(casaco, [casaco], _1)?
** (1) Exit : todas_depois(casaco, [casaco], [])?
L = [] ?
yes

?- todas_depois(casaco, [ casaco, de, pele ], L).
** (1) Call : todas_depois(casaco, [casaco, de, pele], _1)?
** (1) Exit : todas_depois(casaco, [casaco, de, pele], [de, pele])?
L = [de, pele] ?
yes

?- todas_depois(abriu, [o, homem, abriu, a, porta], L).
** (1) Call : todas_depois(abriu, [o, homem, abriu, a, porta], _1)?
** (2) Call : todas_depois(abriu, [homem, abriu, a, porta], _1)?
** (3) Call : todas_depois(abriu, [abriu, a, porta], _1)?
** (3) Exit : todas_depois(abriu, [abriu, a, porta], [a, porta])?
** (2) Exit : todas_depois(abriu, [homem, abriu, a, porta], [a, porta])?
** (1) Exit : todas_depois(abriu, [o, homem, abriu, a, porta], [a, porta])?
L = [a, porta] ?
yes

?- todas_depois(casaco, [o, homem, abriu, a, porta], L).
** (1) Call : todas_depois(casaco, [o, homem, abriu, a, porta], _1)?
** (1) Exit : todas_depois(casaco, [o, homem, abriu, a, porta], [])?
L = [] ?
yes
```

Lista de Exercícios de PROLOG

Informações Gerais: Os exercícios abaixo não valem nota mas devem ser obrigatoriamente resolvidos por completo com o auxílio de qualquer compilador/interpretador PROLOG disponível. As soluções serão extremamente úteis para o T1.

OBS.: Vários dos predicados propostos necessitam da definição prévia de outros predicados, alguns dos quais já foram resolvidos em outras transparências.

EXERCÍCIO 1: Construir um predicado em Prolog, denominado "elem_repetidos", o qual relaciona uma lista de itens com uma sub-lista (possivelmente vazia) de todos os itens da lista original que aparecem repetidos em qualquer numero de vezes. Seu comportamento é o expresso abaixo:

```
?- elem_repetidos([a,b], Z).
Z = [] ?
yes

?- elem_repetidos([a,a,b], Z).
Z = [a] ?
yes

?- elem_repetidos([a, b, c, c, d, e, f, f, g, h, c, c, c, e, e, a], Z).
Z = [a, c, e, f] ?
yes
```

EXERCÍCIO 2: Construir um predicado em Prolog, denominado "intercalada", o qual relaciona duas listas de itens (L1 e L2 – possivelmente vazias) com uma terceira lista (L3), onde esta última contém a intercalação dos elementos das duas outras listas (L1 e L2). Caso L1 e L2 sejam de tamanhos diferentes, completar L3 com os itens não intercalados da lista de maior número de itens. Seu comportamento é o expresso abaixo:

```
?- intercalada([], [], Z).
Z = [] ?
yes

?- intercalada([a], [], Z).
Z = [a] ?
yes
?- intercalada([], [1,2], Z).
Z = [1, 2] ?
yes

?- intercalada([], [1,2,3,4], Z).
Z = [1, 2, 3, 4] ?
yes

?- intercalada([a,b], [1,2,3,4], Z).
Z = [a, 1, b, 2, 3, 4] ?
yes

?- intercalada([a,b,c,d], [1,2,3,4], Z).
Z = [a, 1, b, 2, c, 3, d, 4] ?
yes
```

EXERCÍCIO 3: Construir um predicado em Prolog, denominado "insercao_ord", o qual relaciona um item (I), uma lista ordenada (L1 – possivelmente vazia) de itens da mesma natureza de I, e uma lista ordenada (L2 – não vazia). A lista L2 contem todos os itens de L1 e ainda o item I em uma posição que mantenha L2 de forma ordenada. Seu comportamento é o expresso abaixo:

```
?- insercao_ord( 6, [], Z).
Z = [6] ?
yes

?- insercao_ord( 6, [5], Z).
Z = [5, 6] ?
yes

?- insercao_ord( 6, [3 ,4 , 5 , 8 , 9, 10], Z).
Z = [3, 4, 5, 6, 8, 9, 10] ?
yes
```

EXERCÍCIO 4: Construir um predicado em Prolog, denominado "ordenada", o qual relaciona duas listas (L1 e L2 – possivelmente vazias) de itens numéricos, onde L2 tem exatamente os mesmos itens de L1, porém de maneira ordenada. Seu comportamento é o expresso abaixo:

```
?- ordenada([ ], Z).
Z = [] ?
yes

?- ordenada([11 , 10, 9], Z).
Z = [9, 10, 11] ?
yes
```

```
?- ordenada([11 , 10, 9, 5, 7, 2, 4, 8, 9, 1111, 2, 3, 45, 7, 888, 989], Z).
Z = [2, 2, 3, 4, 5, 7, 7, 8, 9, 9, 10, 11, 45, 888, 989, 1111] ?
yes
```

*** ATENÇÃO ***

```
?- ordenada([ 5 ], [ 4 ]).
no

?- ordenada([ 5, 6, 7 ], [ 4, 5, 6 ]).
no
```

EXERCÍCIO 5: Suponha que existam 10 (dez) pessoas em uma cidade, e seus nomes são: "a", "b", "c", "d", "e", "f", "g", "h", "i", "j". Também é conhecido o fato de que os seguintes pares de pessoas se comunicam freqüentemente:

[[a f] [f e] [g i] [h b] [c h] [j d] [g j] [b h] [d i]]

Claramente, "a", "e" e "f" formam uma sub-cultura na qual seus membros se comunicam mutuamente, porém não se comunicam com nenhuma outra sub-cultura da cidade. Quantas sub-culturas existem no conjunto acima (ou em qualquer conjunto semelhante ao conjunto dado) ?

Para ajudar a responder à pergunta acima, construir um predicado em Prolog, denominado "subcultura", o qual relaciona duas listas (L1 e L2 – possivelmente vazias), as quais englobam sub-listas de apenas 1 (um) nível de profundidade. Cada sub-lista de L1 é composta de um par de itens (nomes de pessoas) ao passo que cada sub-lista de L2 representa um grupo com todas as pessoas de uma sub-cultura. Seu comportamento é o expresso abaixo:

```
?- subcultura([[a, b], [b, c], [d, e], [e, f]], S).
S = [[a, b, c], [d, e, f]] ?
yes

?- subcultura([[a, e], [b, f], [c, g], [d, g]], S).
S = [[a, e], [b, f], [c, g, d]] ?
yes

?- subcultura([[a, f], [f, e], [g, i], [h, b], [c, h], [j, d], [g, j], [b, h], [d, i]], S).
S = [[b, c, h], [d, g, i, j], [a, e, f]] ?
yes
```

Você pode ver algum propósito especifico para um predicado como este? Caso negativo, considere uma lista de pares de cidades ligadas por uma linha telefônica ou de transporte com as quais se pretende fazer estudos de ampliação do sistema de interligação. Ou mesmo, imagine que a lista de pares seja de números com um fator comum.

Estruturas de Dados em PROLOG

Estruturas são objetos compostos, utilizados como "termos" para auxiliar a construção de predicados. Tais estruturas possuem dois tipos de componentes:

1. O "funtor," que é invariavelmente um átomo o qual identifica a estrutura;
2. Os "argumentos" do funtor, que são usados para dar nome às partes do objeto composto. Tais argumentos podem ser constantes, variáveis, ou mesmo outras estruturas.

Em geral, os argumentos são colocados entre parênteses e estão separados uns dos outros por virgulas. Isto gera uma semelhança com a "forma" de se escrever predicados.

Por exemplo, em:

livro("Programação Prolog" , "J. da Silva")

⇒ livro é o funtor da estrutura acima.

Ou ainda, no caso de termos estruturas de estruturas, como em:

livro(autor("J. A.", "da Silva"), "Programação Prolog", 287, "A37.64")

⇒ Tanto livro e autor são os funtores das 2 (duas) estruturas acima;

⇒ A sub-estrutura cujo funtor é autor foi usada para compor a estrutura cujo funtor é livro.

A idéia principal por traz da estruturação de dados se refere à possibilidade de se decompor um problema em sub-componentes, de acordo com a solução adotada. Isto pode evitar o colapso indesejável de objetos de tipos distintos.

Por exemplo:

* *Fatos:*

esta_emprestado(alberto, rich, ???, ???, ...).
esta_emprestado(alberto, raquete, ??? , ...).
esta_emprestado(alberto, aho, ???, ???, ...).
…

* *Pergunta:*

?- esta_emprestado(alberto, Livro, ...).

* *De Forma Estruturada:*

esta_emprestado(alberto, livro(autor("J. A.", "da Silva"),"Programação Prolog", 287, "A37.64")).

A partir daí podemos escrever termos mais complexos para predicados, como no exemplo abaixo:

esta_emprestado(alberto, livro(autor("J. A.", "da Silva"), "Programação Prolog", 287, "A37.64")).

Ou mesmo fazer perguntas mais complexas como:

* Quais os sobrenomes e títulos dos livros emprestados a alberto ?

?- esta_emprestado(alberto, livro(autor(_ , Sobrenome), Titulo, _ , _)).

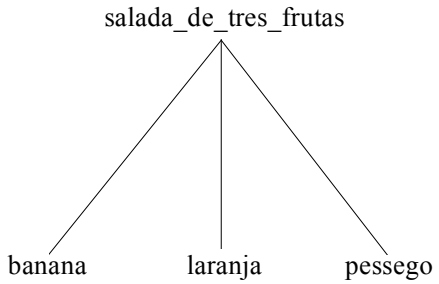
* Há algum livro do autor "Rich" emprestado a alberto ? (R: Sim/Não)

?- esta_emprestado(alberto, livro(autor(_ , "Rich"), _ , _ , _)).

Árvores com Estruturas de Dados de Prolog:

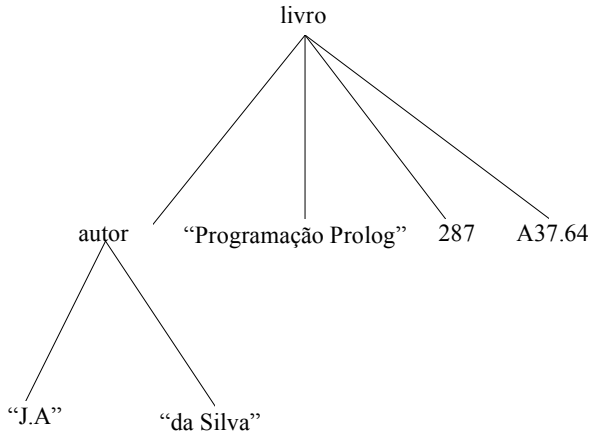
Árvores são uma forma clara de se "visualizar" uma estrutura complexa. Por exemplo, na estrutura abaixo, o funtor "salada_de_tres_frutas" é o nodo raiz e os argumentos são os ramos da árvore:

salada_de_tres_frutas(banana, laranja, pessego)

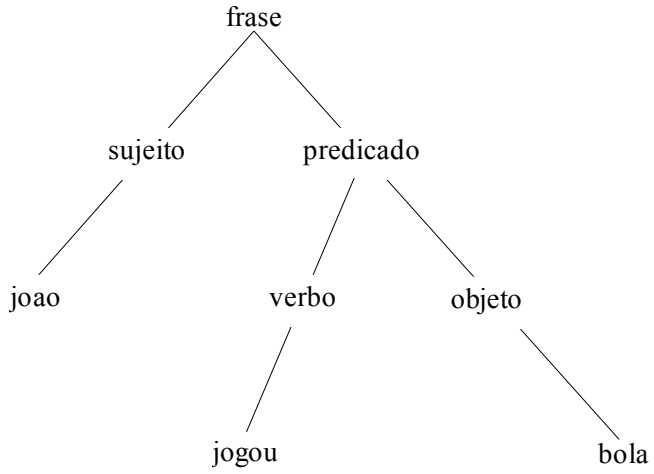


Ou mesmo da expressão já apresentada:

livro(autor("J. A.", "da Silva"), "Programação Prolog", 287, "A37.64")



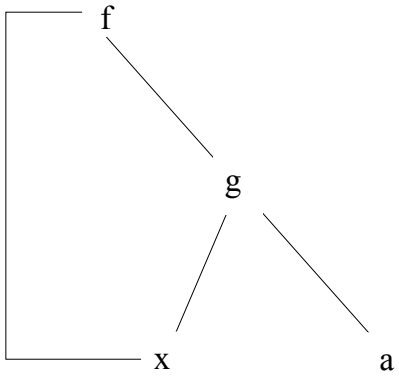
Estruturas podem ser usadas para casamento de padrão complexo. Como exemplo podemos citar a identificação de frases em linguagem natural. Seja a frase "joao jogou bola". Em forma de árvore, sua visualização seria:



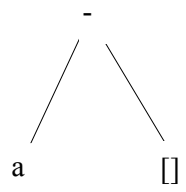
A frase acima, em estrutura Prolog é:

frase(sujeito(joao), predicado(verbo(jogou), objeto(bola)))

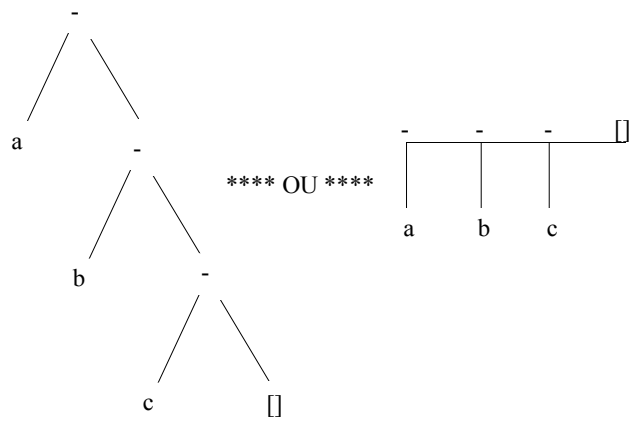
Estruturas menos restritivas do que árvores também podem ser representadas neste sentido. Por exemplo, a expressão f(X, g(X, a)) pode ser visualizada como o seguinte grafo:



Listas podem ser representadas como casos especiais de árvores. Por exemplo, a lista em forma de estrutura que consiste de um elemento (o elemento 'a') poderia ser escrita como (a, []) e visualizada como:



Ou mesmo a estrutura (a, .(b, .(c, []))) seria visualizada como:



Porem, listas dinâmicas em Prolog, por si só, são formas flexíveis de se representar estruturas muito heterogêneas. Implemente muitos programas em Prolog para entender melhor esta afirmativa.

Um exemplo

Construir uma estrutura de dados, em PROLOG, que seja capaz de armazenar informações de nome, endereço e idade em uma árvore binaria, ordenada, onde os nodos mais a esquerda estão com os nomes em ordem alfabética mais baixa.

/ INSERE */*

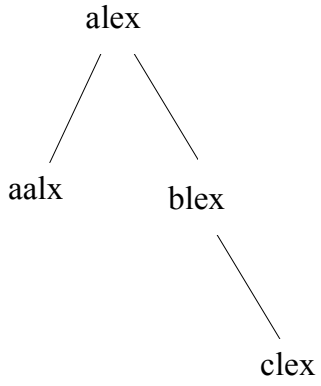
/ Insere um nodo em uma árvore VAZIA ou em um nodo FOLHA. */*
insere(Descritor, vazia, nodo(vazia, Descritor, vazia)).

```
/* Caminha na árvore a ESQUERDA se o nome a ser inserido for alfabaticamente ANTERIOR ao do nodo corrente. */
insere(descritor(Nome, E, I), nodo(AE, descritor(N1, E1, I1), AD), nodo(AE1, descritor(N1, E1, I1), AD)) :-
        Nome < N1,
        insere(descritor(Nome, E, I), AE, AE1).

/* Caminha na árvore a  DIREITA se o nome a ser inserido for alfabaticamente POSTERIOR ao do nodo corrente. */
insere(descritor(Nome, E, I), nodo(AE, descritor(N1, E1, I1), AD), nodo(AE, descritor(N1, E1, I1), AD1)) :-
        Nome > N1,
        insere(descritor(Nome, E, I), AD, AD1).

/* Atualiza o nodo CORRENTE se o nome contido no descritor for igual. */
insere(descritor(Nome, E, I), nodo(AE, descritor(N1, _ , _ ), AD), nodo(AE, descritor(Nome, E, I), AD)) :-
        Nome = N1.
```

Cálculo de um exemplo:

```
?- insere(descritor(alex, e1, 10), vazia, A1), insere(descritor(blex, e2, 11), A1 , A2), insere(descritor(clex, e3, 12), A2 , A3), insere(descritor(aalx, e4, 13), A3 , A4), write(A4).

nodo(nodo(vazia, descritor(aalx, e4, 13), vazia), descritor(alex, e1, 10), nodo(vazia, descritor(blex, e2, 11), nodo(vazia, descritor(clex, e3, 12), vazia)))
```

/* VISUALIZAÇÃO GRÁFICA DO EXEMPLO */



/* ELIMINA */

```
/* Elimina um nodo FOLHA, ou nodo ÚNICO da árvore, cujo nome no descritor coincide com o nome procurado. */
elimina(descritor(Nome, _ , _ ), nodo(vazia, descritor(N1, _ , _ ), vazia), vazia) :-
        N1 = Nome.

/* Caminha na árvore a ESQUERDA se o nome a ser inserido for alfabaticamente ANTERIOR ao do nodo corrente. */
elimina(descritor(Nome, E, I), nodo(AE, descritor(N1, E1, I1), AD), nodo(AE1, descritor(N1, E1, I1), AD)):-
        Nome < N1,
        elimina(descritor(Nome, E, I), AE, AE1).

/* Caminha na árvore a  DIREITA se o nome a ser inserido for alfabaticamente POSTERIOR ao do nodo corrente. */
```

```
elimina(descritor(Nome, E, I), nodo(AE, descritor(N1, E1, I1), AD), nodo(AE, descritor(N1, E1, I1), AD1)):-
        Nome > N1,
        elimina(descritor(Nome, E, I), AD, AD1).

/* Não elimina nenhum nodo da árvore caso não seja encontrado o nodo com o descritor cujo nome coincide, ou se a árvore estiver vazia. */
elimina(descritor( _ , _ , _ ), vazia, vazia).


                                /*  TROCA  RAMOS  */


troca_ramos(descritor(Nome, _ , _ ), nodo(AE, descritor(N1, _ , _ ), AD),
                                nodo(AD, descritor(N1, _ , _ ), AE)) :-
        N1 = Nome.
…
```

O Uso do CUT(!) e do FAIL

O uso do "CUT" (!)

Corte de soluções múltiplas – permite informar ao interpretador Prolog quais ponteiros anteriores do processamento devem ser desconsiderados para efeito de controle de "BACKTRACKING" ou "retroação":

```
…
pai(joao, jose).
pai(joao, paulo).
pai(alberto, joao).
pai(fabricio, alberto).
pai(jonatas, fabricio).
…

?- pai(X, _ ).
X = joao ? ;
X = joao ? ;
X = alberto ? ;
X = fabricio ? ;
X = jonatas ? ;
no

filho(X, Y) :-
        pai(Y, X).

?- filho(F, P).
F = jose  P = joao ? ;
F = paulo  P = joao ? ;
F = joao  P = alberto ? ;
F = alberto  P = fabricio ? ;
F = fabricio  P = jonatas ? ;
no


filho(X, Y) :-
        pai(Y, X), !.


?- filho(F, P).
F = jose  P = joao ? ;
no
```

```
pai(X) :-
     pai(X, _ ).

?- pai(X).
X = joao ? ;
X = joao ? ;
X = alberto ? ;
X = fabricio ? ;
X = jonatas ? ;
no

pai(X) :-
          pai(X, _ ),
          !.

?- pai(X).
X = joao ? ;
no



soma_dos_int_de_1_a(1, 1).

soma_dos_int_de_1_a(N, Soma) :-
          N1 is N - 1,
          soma_dos_int_de_1_a(N1, Tmp_soma),
          Soma is Tmp_soma + N.


?- soma_dos_int_de_1_a(3, X).
** (1) Call : soma_dos_int_de_1_a(3, _1)?
** (2) Call : soma_dos_int_de_1_a(2, _2)?
** (3) Call : soma_dos_int_de_1_a(1, _3)?
** (3) Exit : soma_dos_int_de_1_a(1, 1)?
** (2) Exit : soma_dos_int_de_1_a(2, 3)?
** (1) Exit : soma_dos_int_de_1_a(3, 6)?
X = 6 ? ;
** (1) Redo : soma_dos_int_de_1_a(3, 6)?
** (2) Redo : soma_dos_int_de_1_a(2, 3)?
** (3) Redo : soma_dos_int_de_1_a(1, 1)?
** (4) Call : soma_dos_int_de_1_a(0, _4)?
** (5) Call : soma_dos_int_de_1_a(-1, _5)?
** (6) Call : soma_dos_int_de_1_a(-2, _6)?
** (7) Call : soma_dos_int_de_1_a(-3, _7)?
** (8) Call : soma_dos_int_de_1_a(-4, _8)?
** (9) Call : soma_dos_int_de_1_a(-5, _9)?
** (10) Call : soma_dos_int_de_1_a(-6, _10)?
… … …
… … …

;;; PROLOG ERROR - MEMORY LIMIT (pop_prolog_lim) EXCEEDED
;;; DOING    :  soma_dos_int_de_1_a/2
;;; [execution aborted]



soma_dos_int_de_1_a(1, 1) :-
          !.
soma_dos_int_de_1_a(N, Soma) :-
          N1 is N - 1,
          soma_dos_int_de_1_a(N1, Tmp_soma),
          Soma is Tmp_soma + N.
```

```
?- soma_dos_int_de_1_a(3, X).
** (1) Call : soma_dos_int_de_1_a(3, _1)?
** (2) Call : soma_dos_int_de_1_a(2, _2)?
** (3) Call : soma_dos_int_de_1_a(1, _3)?
** (3) Exit : soma_dos_int_de_1_a(1, 1)?
** (2) Exit : soma_dos_int_de_1_a(2, 3)?
** (1) Exit : soma_dos_int_de_1_a(3, 6)?
X = 6 ? ;
** (1) Redo : soma_dos_int_de_1_a(3, 6)?
** (2) Redo : soma_dos_int_de_1_a(2, 3)?
** (3) Redo : soma_dos_int_de_1_a(1, 1)?
** (3) Fail : soma_dos_int_de_1_a(1, _3)?
** (2) Fail : soma_dos_int_de_1_a(2, _2)?
** (1) Fail : soma_dos_int_de_1_a(3, _1)?
no
```

O uso do "FAIL"

O predicado "FAIL" é aquele que, quando incluído no corpo de uma cláusula qualquer, o valor-verdade da cláusula será "falso" (não necessariamente o do predicado). Isto provoca retroação (backtraching) e, consequentemente, combinações automáticas de valores instanciáveis nas variáveis da cláusula.

```
nomes_de_mulheres:-
        mulher(X),
        write(X),
        nl,
        fail.

?- nomes_de_mulheres.
claudia
silvia
clara
angela
maria
marcia
no
```

A combinação "CUT-FAIL":

Quando o FAIL é encontrado depois de um CUT (!), o comportamento esperado do "backtracking" é sempre alterado devido à presença do CUT. Na verdade, a combinação CUT-FAIL se torna útil em diversas condições práticas por razões variadas. Exemplo:

```
fatorial(X, _) :- X < 0, !, fail.

fatorial(0,1) :- !.

fatorial(X, FX)  :-  Y is X-1,   fatorial(Y, FY),  FX is X * FY.
```