# Introdução a Prolog

**Listas e Cut**

**(Cap. 5 e 6 do livro de prolog)**

## Listas

Estruturas simples, sequência de qualquer numero de itens.

Em Prolog, deve-se considerar:

- lista vazia []
- lista não-vazia [ cabeça | corpo ]

[brasil, uruguai, argentina, paraguai]

[brasil, [uruguai, [argentina, [paraguai,[] ] ] ] ]

**Unificação**:

[X | Y] ou [X | [Y | Z] ] unificam com [a, b, c, d]
pois X = a, Y = b e Z = [c, d]

[X, Y, Z] não unifica com [a, b, c, d]

# Listas – operações

- **Construção de listas**

  Elementos básicos: uma cabeça e um corpo.

  Tal relação pode ser escrita em um único fato:

  ```
  cons(X, Y, [X | Y]).          % construção
  ```

# Listas – operações

- **Ocorrência de elementos**

  Um elemento X é membro de L se:

  (1) X é a cabeça de L, ou

  (2) X é membro do corpo de L.

Em prolog:

```
membro(X, [X | _ ]).              % ocorrência de elemento
membro(X, [ _ | C]) :- membro(X, C).
```

A relação membro/2 é não-determinística por natureza (várias ocorrências de X, cada uma será uma resposta).

# Listas – operações

- **Concatenação de listas conc(L1, L2, LR)**

  Considera-se:

  (1) Se L1 é uma lista vazia, então a LR será a própria lista L2.

  (2) Se L1 for uma lista não-vazia, então ela possui uma cabeça e um corpo e é denotada por [X1|R1]. A concatenação de [X|R1] com a L2, produzirá uma lista resultante com a cabeça X1 da L1 e um corpo R3, que será a concatenação do corpo R1 com toda a lista L2.

Em prolog:

```
conc([], L, L).                                  % concatenação
conc([X1 | R1], L2, [X1 | R3]) :- conc(R1, L2, R3).
```

# Listas – operações

**Testar** conc(L1, L2, [a, b, c]) **para ver todas as possibilidades para a formação das listas, via backtracking !!!!!**

**Lembrando da reversibilidade dos argumentos, pode-se também achar sucessores e predecessores de um item, ou "apagar" elementos:**

conc( _, [ X, g, Y| _ ], [a, b, c, d, e, f, g, h]).

**=> Resp:** X=f e Y=h

conc( Trab, [sex | _ ], [seg, ter, qua, qui, sex, sab, dom]).

**=> Resp:** Trab = [seg, ter, qua, qui]

# Listas – operações

- Sublista

  Considera-se que S é uma sublista de L se:

  (1) L pode ser decomposta em duas listas, L1 e L2, e

  (2) L2 pode ser decomposta em S e L3.

  Em prolog:

```
sublista(S, L) :-  conc(L1, L2, L), conc(S, L3, L2).
```

# Listas – operações

**Remoção de elemento**

Para a remoção de um elemento X de uma lista L considera-se:

(1) Se X é a cabeça da lista L, então L1 será o seu corpo;

(2) Se X está no corpo de L, então L1 é obtida removendo X desse corpo.

Em prolog

```
remover(X, [X | C], C).                % remoção de elemento
remover(X, [Y | C], [Y | D]) :- remover(X, C, D).
```

- A relação remover/3 é não-determinística por natureza. Se há diversas ocorrências de X em L, ela é capaz de retirar cada uma delas através do mecanismo de *backtracking* do Prolog. Porém, em cada execução do predicado remove/3 somente uma das ocorrências de X, deixando as demais intocáveis.

# Listas – operações

**Outros usos do remover/3**

- Essa relação pode ser ainda **usada no sentido inverso para inserir um novo item em qualquer lugar da lista**. Por exemplo, pode-se formular a questão: "Qual é a lista L, da qual retirando-se 'a' , obtém-se a lista [b, c, d]?"

**?-remover(a, L, [b, c, d]).**

⇒**Resp: L=[a, b, c, d]; L=[b, a, c, d]; L=[b, c, a, d]; L=[b, c, d, a]; não**

# Listas – operações

**Inversão de lista**

- Uma das estratégias é "inversão ingênua" que se baseia numa abordagem muito direta, embora seu **tempo de execução** seja **proporcional ao quadrado do tamanho** da lista:
  (1) Tomar o primeiro elemento da lista;
  (2) Inverter o restante;
  (3) Concatenar a lista formada pelo primeiro elemento ao inverso do restante.
- Em prolog:

```
inverter([], []).                    % inversão de lista
inverter([X | Y], Z) :-  inverter(Y, Y1), conc(Y1, [X], Z).
```

Esse predicado, juntamente com conc/3, costuma ser usado como um teste benchmark para sistemas Prolog. Quando o número de inferências lógicas (objetivos Prolog) é dividido pelo número de segundos gastos, o valor obtido mede a velocidade do sistema Prolog em LIPS (*logic inferences per second*).

# Listas – operações

**Inversão de lista**

A inversão de listas pode, entretanto ser obtida de modo mais eficiente por meio de um predicado auxiliar, iterativo, aux/3, tornando o **tempo de execução** apenas **linearmente proporcional ao tamanho da lista** a inverter:

```
inverter2(X, Y) :- aux( [], X, Y).
aux(L, [], L).
aux(L, [X | Y], Z) :- aux( [X | L], Y, Z).
        % coloca a cabeça da 2ª lista como cabeça da 1ª lista
```

# Listas – operações

Permutação

- O predicado permutar/2 deve novamente basear-se na consideração de 2 casos, dependendo da lista a ser permutada:
  (1) Se a 1ª lista é vazia, então a 2ª também é;
  (2) Se a 1ª lista é não-vazia, então possui a forma [X|L] e uma permutação de tal lista pode ser construída primeiro permutando L para obter L1 e depois inserindo X em qualquer posição de L1.

Em prolog:

```
permutar([], []).
permutar([X | L], P) :- permutar(L, L1), inserir(X, L1, P).
```

# Listas – operações

Outras funções:

```
tamanho([], 0).                    % número de elementos da lista
tamanho([ _ | R], N) :- tamanho(R, N1), N is N1+1.

enesimo(1, X, [X | _ ]).           %selecionar elemento N
enesimo(N, X, [ _ | Y]) :- enesimo(M, X, Y), N is M+1.

seleciona([], _, []).              % reunir determinados itens em lista
seleciona([M | N], L, [X | Y]) :- enesimo(M, X, L), seleciona(N, L, Y).
```

# Listas – operações

Outras funções:

```
soma([], 0).                                   % lista vazia
soma([X | Y], S) :- soma(Y, R), S is R+X.

produto([], 0).                    % lista vazia
produto([X], X).                   % lista unitaria
produto(L, P) :- prod(L, P).       % lista com 2+ elementos

prod([], 1).                        % predicado auxiliar
prod([X | Y], P) :- prod(Y, Q), P is Q*X.

intersec([X | Y], L, [X |Z]) :- membro(X, L), intersec(Y, L, Z).
intersec([_ |Y], L, Z) :- intersec(Y, L, Z).
intersec(_, _, []).
```

## Controle

O programador pode controlar a execução de seu programa através da reordenação das cláusulas ou de objetivos no interior destas.

Existe outro instrumento para o controle de programas: o "**cut**“ (!), que se destina a prevenir a execução do *backtracking* quando este não for desejado.

## Controle

No Prolog, a evolução da busca por soluções assume a forma de uma árvore ("árvore de pesquisa“) que é percorrida sistematicamente de cima para baixo (top-down) e da esquerda para direita, segundo o método denominado "depth-first search" (pesquisa em profundidade).

O papel desempenhado pelo **cut (!)**, é de extrema importância para semântica operacional dos programas Prolog, permitindo declarar ramificações da árvore de pesquisa que não devem ser retomadas no *backtracking*.

# Controle

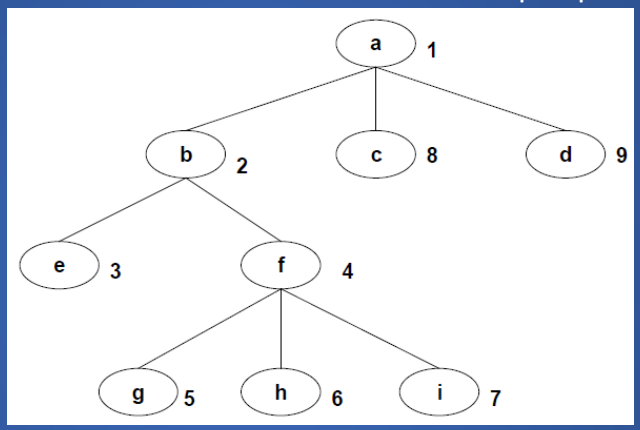
Algumas das principais aplicações do **cut** são:

- Unificação de padrões, de forma que quando um padrão é encontrado os outros padrões possíveis são descartados
- Na implementação da negação como regra de falha
- Para eliminar da árvore de pesquisa soluções alternativas quando uma só é suficiente
- Para encerrar a pesquisa quando a continuação iria conduzir a uma pesquisa infinita, etc.

# Controle

Arvore de pesquisa

```
a :- b.
a :- c.
a :- d.
b :- e.
b :- f.
f :- g.
f :- h.
f :- i.
d.
```




O programa obtém sucesso quando acha d e o caminho percorrido é: **a, b, e, (b), f, g, (f), h, (f), i, (f), (b), (a), c, (a), d**

onde o caminho em *backtracking* é representado entre parênteses.

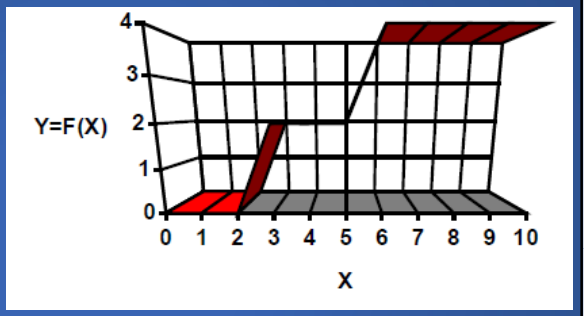
# Controle – *backtracking* desnecessário

**Problema:**

Se x<3 então y=0

Se 3<=x<6 então y=2

Se 6<=x então y=4



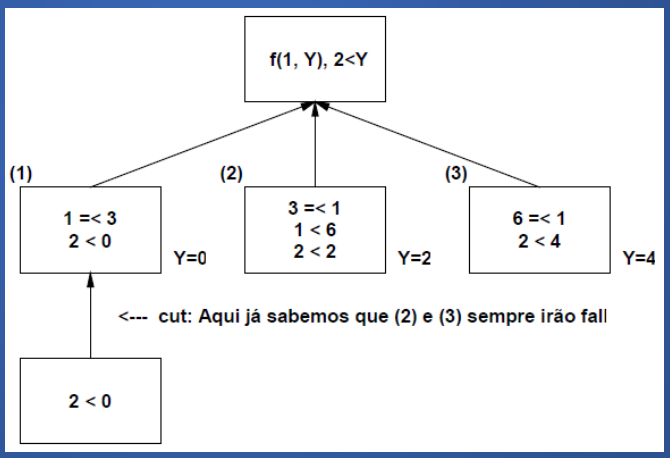

Em prolog:

```
f(X, 0) :- X < 3.
f(X, 2) :- 3 =< X, X < 6.
f(X, 4) :- 6 =< X.
```

# Controle - cut

**O que ocorre em:**

**?- f(1, Y), 2 < Y.**



# Controle - cut

**SOLUÇÃO COM EXCLUSÃO MÚTUA:** somente uma das possibilidades será executada.

```
f(X, 0) :- X < 3, !.
f(X, 2) :- 3 =< X, X < 6, !.
f(X, 4) :- 6 =< X.
```

# Controle - cut

**Alterando a Interpretação Declarativa**

**?- f(7,Y). => Y = 4**

Todas as 3 regras foram tentadas antes da resposta ter sido obtida, produzindo a seguinte sequência de objetivos:

(1) Tenta a regra (1): 7<3 falha. Aciona o *backtracking* e tenta a regra (2). O cut não foi atingido.

(2) Tenta a regra (2): 3=<7 é bem-sucedido, mas 7<6 falha. Aciona o *backtracking* e tenta a regra (3). O cut não foi atingido.

(3) Tenta a regra (3): 6=<7 é bem-sucedido.

Logo, pode-se ter uma formulação mais econômica do progr.:

```
f(X, 0) :- X < 3, !.
f(X, 2) :- X < 6, !.
f(X, 4).
```

## Controle – aplicações do cut

**MÁXIMO ENTRE 2 NÚMEROS**

```
max(X, Y, X) :- X >= Y.
max(X, Y, Y) :- X < Y.
```

```
max(X, Y, X) :- X >= Y, !.
max(X, Y, Y).
```

**SOLUÇÃO ÚNICA PARA OCORRÊNCIA (1ª)**

```
membro(X, [ X | _ ]) :- !.
membro(X, [ _ | Y ]) :- membro(X, Y).
```

**ADIÇÃO DE ELEMENTOS SEM DUPLICAÇÃO**
**(adicionar X a L somente se X não estiver em L)**

```
adicionar(X, L, L) :- membro(X, L), !.
adicionar(X, L, [ X | L ]).
```

## Controle – aplicações do cut

**IF-THEN-ELSE** (com meta-variáveis, embora não seja considerado estilo declarativo puro)

```
ifThenElse( X, Y, _):- X, !, Y.  % se X é true, corta, e executa Y
ifThenElse( _, _, Z):- Z.        % senao executa Z
```

Exemplos de chamada

```
ifThenElse( X, Y is Z+1, Y is 0)

ifThenElse( membro(X,L) , write('Yes'), write('No') )
```

# Controle – negação por falha (fail)

Pode-se dizer que alguma coisa NÃO É VERDADEIRA em Prolog usando o predicado **fail**:

```
gosta(maria, X) :-
    cobra(X), !, fail.
gosta(maria, X) :-
    animal(X).
```

```
gosta(maria, X) :-
    cobra(X), !, fail;
    animal(X).
```

```
diferente(X, Y) :- X==Y, !, fail; true.
```

```
nao(P) :- P, !, fail; true.
```

```
gosta(maria, X) :-
    animal(X), nao(cobra(X)).

diferente(X, Y) :- nao(X==Y).
```