
Aplicações de Prolog

Prolog Avançado

Mario Benevides
COPPE-UFRJ

Listas

Lista $L=[a,b,c,d,e,f]$

- onde a,b,\dots,f são elementos da lista.
- o elemento a é chamado de **cabeça** e a lista restante $[b,c,d,e,f]$ é chamada de **cauda**.

Notação: $[X|Y]$ é uma lista onde X é a cabeça e Y é a cauda.

- Tipo de dados tipicamente recursivo: uma lista é um elemento concatenado com uma lista.

Definição: O conjunto de termos Prolog denotando listas é definido recursivamente da seguinte forma:

1. $[]$ é um termo Prolog denotando a lista de comprimento zero chamada de lista vazia;
2. Se X é um termo Prolog e Y é uma lista de comprimento $n-1$, então $[X|Y]$ é um termo Prolog denotando uma lista de comprimento n . O termo Prolog X é chamado de cabeça e o Y de cauda da lista $[X|Y]$.

Exemplos:

Para ser rigoroso a lista $[a,b,c,d]$ seria $[a|[b|[c|[d|[]]]]]$. Esta notação é útil algumas vezes mas em geral podemos omitir as barras como:

$[a,b,c,d]$	cabeça= a	cauda= $[b,c,d]$
-------------	-------------	------------------

$[[a,b],c,[d,e,f],g]$	cabeça= $[a,b]$	cauda= $[c,[d,e,f],g]$
-----------------------	-----------------	------------------------

- Vamos definir agora algumas operações sobre lista:

Programa 1: Predicado lista(X) que verifica se X é uma lista;

```
lista([]) ←.  
lista([X|Y] )← lista(Y).
```

Pergunta: ← lista([a,b,c])

lista([a,b,c])
↓
lista([b,c])
↓
lista([c])
↓
lista([])

- Outra operação bastante útil é testar quando um dado elemento pertence a uma certa lista.

programa 2: predicado $\text{membro}(X,Y)$ = X o elemento X é membro da lista Y.

```
membro(X,[X|Z]) ← .
membro(X,[Y|Z]) ← membro(X,Z).
```

- A primeira cláusula testa se X é o cabeça da lista Y e a segunda testa se X está na cauda.

Convensão:	variáveis para elementos:	X,Y,Z,...
	variáveis para listas:	Xs,Ys,Zs,....

- Próxima operação é para testar se uma lista é sublista de outra.
- Exemplo: $L=[a,b,c,d]$ uma lista, $[b,c]$ é uma sublista de L, mas $[a,c]$ não, pois para ser sublista os elementos tem de ser consecutivos em L.
- Exercício: $\text{membro}(a,[[b,c],d,[a,f]])$ $\text{membro}([a,f],[[b,c],d,[a,f]])$

- Vamos primeiro definir dois tipos particulares de sublistas.

programa 3: predicado `prefixo(Xs,Ys)`

```
prefixo([],Ys) ← .  
prefixo([X|Xs],[X|Ys]) ← prefixo(Xs,Ys).
```

- Exercício: `prefixo([a,b],[a,b,c,d])` é verdade.

programa 4: predicado `sufixo(Xs,Ys)`

```
sufixo(Xs,Xs) ← .  
sufixo(Xs,[Y|Ys]) ← sufixo(Xs,Ys).
```

- Exercício: `sufixo([c,d].[a,b,c,d])` é verdade.

- Existem várias maneiras de se implementar a operação de sublista, a seguir apresentaremos 3:

programa 5: predicado sublista(Xs, Ys)

a. $sublista(Xs, Ys) \leftarrow prefixo(Ps, Ys) , sufixo(Xs, Ps).$

ou

b. $sublista(Xs, Ys) \leftarrow prefixo(Xs, Ss) , sufixo(Ss, Ys).$

ou

c. $sublista(Xs, Ys) \leftarrow prefixo(Xs, Ys)$

$sublista(Xs, [Y|Ys]) \leftarrow sublista(Xs, Ys) .$

- O predicado membro pode ser visto como um caso especial de sublista:

$membro(X, Xs) \leftarrow sublista([X], Xs).$

- Uma outra operação muito útil é concatenar duas listas resultando numa terceira. Por exemplo concatenar $[a,b]$ com $[c,d]$ resultando $[a,b,c,d]$.

programa 6: predicado `concat(Xs,Ys,Zs)`, concatena lista `Xs` com lista `Ys` resultando na lista `Zs`.

```
concat([],Ys,Ys) ←.  
concat([X|Xs],Ys,[X|Zs]) ← concat(Xs,Ys,Zs).
```

- Pergunta: `concat([a,b],[c,d],[a,b,c,d]) ←`

`concat([a,b],[c,d],[a,b,c,d])`
↓
`concat([b],[c,d],[b,c,d])`
↓
`concat([], [c,d], [c,d])`

- Exercício:
1. definir prefixo, sufixo e membro em função de `concat`.
 2. `adjacente(X,Y,Zs)` - se elemento `X` é adjacente a `Y` na lista `Zs`.
 3. `ultimo(X,Ys)` - se `X` é o último elemento da lista `Ys`.

1. $\text{prefixo}(Xs, Ys) \leftarrow \text{concat}(Xs, Zs, Ys).$
 $\text{sufixo}(Xs, Ys) \leftarrow \text{concat}(Zs, Xs, Ys).$
 $\text{membro}(X, Ys) \leftarrow \text{concat}(Zs, [X|Xs], Ys).$
2. $\text{adjacente}(X, Y, Zs) \leftarrow \text{concat}(Ws, [X, Y|Ys], Zs).$
3. $\text{ultimo}(X, Xs) \leftarrow \text{concat}(Ys, [X], Xs).$

programa 7: predicado $\text{inverso}(Xs, Ys)$ - a lista Ys é o inverso da lista Xs .

$\text{inverso}([], []) \leftarrow.$
 $\text{inverso}([X|Xs], Zs) \leftarrow \text{inverso}(Xs, Ys), \text{concat}(Ys, [X], Zs).$

- outra maneira de definir o inverso é usando-se um predicado $\text{inverte}(Xs, Ys, Zs)$ - Zs é o resultado da concatenação de Xs inverso com Ys .

$\text{inverte}([], Ys, Ys) \leftarrow.$
 $\text{inverte}([X|Xs], As, Ys) \leftarrow \text{inverte}(Xs, [X|As], Ys).$
 $\text{inverso}(Xs, Ys) \leftarrow \text{inverte}(Xs, [], Ys).$

Exercício: fazer a árvore de derivação para provar `inverso([a,b,c,d],[d,c,b,a])` para a primeira definição.

`inverso([a,b,c,d],[d,c,b,a])`
↓
`inverte([a,b,c,d],[],[d,c,b,a])`
↓
`inverte([b,c,d],[a],[d,c,b,a])`
↓
`inverte([c,d],[b,a],[d,c,b,a])`
↓
`inverte([d],[c,b,a],[d,c,b,a])`
↓
`inverte([], [d,c,b,a], [d,c,b,a])`

programa 8: $\text{deleta}(L1, X, L2)$ - lista $L2$ é obtida da lista $L1$ deletando-se todas as ocorrências de X em $L1$.

$\text{deleta}([X|Xs], X, Ys) \leftarrow \text{deleta}(Xs, X, Ys).$
 $\text{deleta}([X|Xs], Z, [X|Ys]) \leftarrow X \neq Z, \text{deleta}(Xs, Z, Ys).$
 $\text{deleta}([], X, []) \leftarrow .$

Exercício: 1. $\text{deleta}([a,b,c,b], b, X)$?
 2. Como fazer para deletar apenas uma ocorrência?

programa 9: predicado $\text{seleciona}(X, Xs, Ys)$.

$\text{seleciona}(X, [X|Xs], Xs) \leftarrow .$
 $\text{seleciona}(X, [Y|Ys], [Y|Zs]) \leftarrow \text{seleciona}(X, Ys, Zs).$

- desenvolvimento de programas top-down. Veremos como exemplo programas para ordenação de lista: $\text{ordena}(Xs, Ys)$ - Ys é Xs ordenada.

programa 10: ordenação usando método da bolha.

$\text{ordena}(Xs, Ys) \leftarrow \text{permuta}(Xs, Ys) , \text{ordenado}(Ys).$

$\text{permuta}([], []).$

$\text{permuta}(Xs, [Z|Zs]) \leftarrow \text{seleciona}(Z, Xs, Ys) , \text{permuta}(Ys, Zs).$

$\text{ordenado}([X]).$

$\text{ordenado}([X, Y|Ys]) \leftarrow X \leq Y , \text{ordenado}([Y|Ys]).$

programa 11: ordenação usando método da inserção.

```
ordena([],[]).  
ordena([X|Xs],Ys) ← ordena(Xs,Zs) , insere(X,Zs,Ys).
```

```
insere(X,[],[X]).  
insere(X,[Y|Ys],[Y|Zs]) ← X>Y , insere(X,Ys,Zs).  
insere(X,[Y|Ys],[X,Y|Ys]) ← X≤Y.
```

programa 12: ordenação usando método quicksort.

```
quicksort([],[]).  
quicksort([X|Xs],Ys) ← particao(Xs,X,Infs,Sups) , quicksort(Infs,Is) ,  
                        quicksort(Sups,Ss) , concat(Is,[X|Ss],Ys).  
  
particao([],Y,[],[]).  
particao([X|Xs],Y,[X|Is],Ss) ← X≤Y , particao(Xs,Y,Is,Ss).  
particao([X|Xs],Y,Is,[X|Ss]) ← X>Y , particao(Xs,Y,Is,Ss).
```

Árvores Binárias

- Vamos representar usando uma função `arv(elemento, esquerda, direita)`.
- Árvore vazia é `nil`.



- `arv(a,arv(b,nil,nil),arv(c,nil,nil))`.

programa 1: predicado `arvbin(X)` - testa se `X` é uma árvore binária.

`arvbin(nil).`

`arvbin(arv(X,E,D)) ← arvbin(E) , arvbin(D).`

programa 2: `busca(elemento,arvore)`

busca(X,arv(X,E,D)).
busca(X,arv(Y,E,D)) \leftarrow busca(X,E).
busca(X,arv(Y,E,D)) \leftarrow busca(X,D).

programa 3: pre(A,Xs), in(A,Xs) e pos(A,Xs) percorre a árvore A em ordem pré-ordem in-ordem e em pos-ordem respectivamente.

pre(arv(X,E,D),Xs) \leftarrow pre(E,Es) , pre(D,Ds) , concat([X|Es],Ds,Xs).
pre(nil,[]).

in(arv(X,E,D),Xs) \leftarrow in(E,Es) , in(D,Ds) , concat(Es,[X|Rs],Xs).
in(nil,[]).

pos(arv(X,E,D),Xs) \leftarrow pos(E,Es) , pos(D,Ds) , concat(Ds,[X],Ds1) ,
concat(Es,Ds1,Xs).
pos(nil,[]).

Cut & Fail

- Corte (cut) é denotado por !.
- A idéia é podar a árvore de prova para evitar ramos desnecessários e tornar o programa mais eficiente.
- É uma instrução imperativa(procedural) que vai contra a filosofia declarativa da programação em lógica.
- Obriga o programador a saber como o programa será executado para poder entendê-lo. Não é uma boa tecnica de programção.
- Porém, pode ser muito útil e tornar o programa bem mais eficiente.
- Para ser usado deve-se ter certeza de que seu uso não está levando a consequências inesperadas. Por exemplo, podando ramos que possuem soluções.

programa: merge(Xs,Ys,Zs) - faz o merge das listas ordenadas Xs e Ys produzindo a lista ordena Zs.

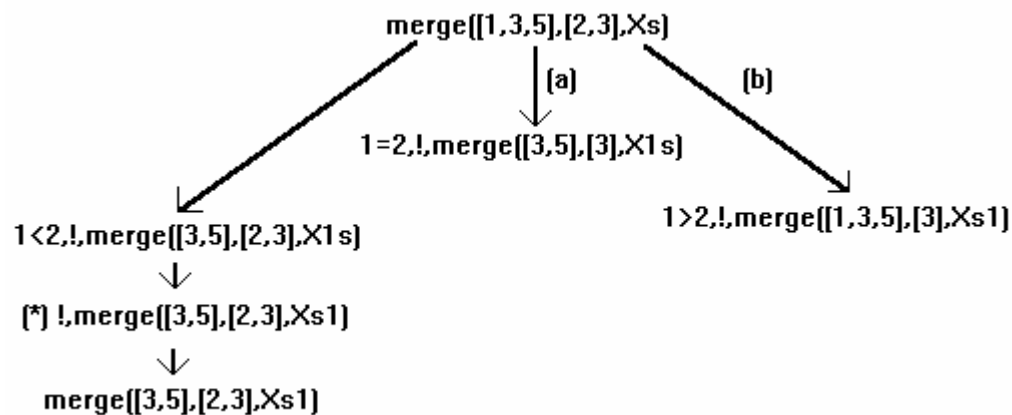
```
merge([X|Xs],[Y|Ys],[X|Zs]) ← X < Y , merge(Xs,[Y|Ys],Zs).  
merge([X|Xs],[Y|Ys],[X,Y|Zs]) ← X = Y , merge(Xs,Ys,Zs).  
merge([X|Xs],[Y|Ys],[Y|Zs]) ← X > Y , merge([X|Xs],Ys,Zs).  
merge(Xs,[],Xs).  
merge([],Ys,Ys).
```

- Reescrevendo o programa usando corte:

programa: merge(Xs,Ys,Zs) - faz o merge das listas ordenadas Xs e Ys produzindo a lista ordena Zs.

```
merge([X|Xs],[Y|Ys],[X|Zs]) ← X < Y , ! , merge(Xs,[Y|Ys],Zs).  
merge([X|Xs],[Y|Ys],[X,Y|Zs]) ← X = Y , ! , merge(Xs,Ys,Zs).  
merge([X|Xs],[Y|Ys],[Y|Zs]) ← X > Y , ! , merge([X|Xs],Ys,Zs).  
merge(Xs,[],Xs) ← !.  
merge([],Ys,Ys) ← !.
```

- Exemplo: merge([1,3,5],[2,5],Xs)?



- O efeito do corte será que apenas o ramo (*) será percorrido e após o sucesso os ramos (a) e (b) serão podados, i.e., serão abandonados.
- A idéia é colocar o corte após o teste se $X < Y$, $X = Y$ e $X > Y$, pois apenas uma dessas condições será satisfeitas e as outras possibilidades não levarão a soluções. Logo, estas últimas podem ser abandonadas.

programa 2: $\text{minimo}(X,Y,\text{Min})$ - Min é o mínimo entre X e Y.

$\text{minimo}(X,Y,X) \leftarrow X \leq Y, !.$

$\text{minimo}(X,Y,Y) \leftarrow X > Y, !.$

- Efeito do corte: $C = A \leftarrow B_1, B_2, \dots, B_k, !, B_{k+2}, \dots, B_n$. Pergunta G.
 - se G unifica com A e B_1, B_2, \dots, B_k tem sucesso;
 - o programa fica comprometido com C, i.e., todas as outras cláusulas que tem A na cabeça que podem unificar com G são ignoradas;
 - se B_i falha, para $i > k$, então só fazemos backtracking até o corte !.
 - outras opções de computação de B_i , para $i \leq k$, são podadas da árvore de prova;
 - se o backtracking atinge o corte ! em C, então o corte falha, e a busca continua da última escolha feita antes de G escolher C.

Fail

- **fail** é um predicado especial do sistema, i.e., do Prolog cujo resultado da execução sempre falha (sempre da falso).
- *fail* combinado com o *cut* podem dar resultados bem interessantes.
- a negação *not* e o predicado \neq podem ser definidos usando-se *cut-fail*.

programa 3: not P

```
not X  $\leftarrow$  X , ! , fail.  
not X.
```

programa 4: $X \neq Y$

```
 $X \neq X \leftarrow$  ! , fail.  
 $X \neq Y$ .
```

Processamento de Ling. Natural

- Primeira aplicação de Prolog foi na construção de analisadores sintáticos.

- Uma gramática $G = \langle \text{Alfabeto}, \text{Regras}, S \rangle$, onde S é o símbolo inicial:

$$S \rightarrow aSb \quad S \rightarrow c$$

- gera todas as palavras com o mesmo número de a's e de b's a esquerda e a direita respectivamente do c.
- Convenção: símbolos terminais em letra minúscula e não-terminais em maiúscula.
- Vamos apresentar a seguir um pequeno fragmento da gramática do inglês.

- **Gramática:**

Sentença \rightarrow Frase_Nominal , Frase_Verbal.
Frase_Nominal \rightarrow Artigo , Frase_Nominal2.
Frase_Nominal \rightarrow Nome.
Frase_Nominal2 \rightarrow Adjetvo , Frase_Nominal2.
Frase_Nominal2 \rightarrow Nome.
Frase_Verbal \rightarrow Verbo.
Frase_Verbal \rightarrow Verbo , Frase_Nominal.
Artigo \rightarrow {a, the}
Verbo \rightarrow {contains, is , are, ...}
Nome \rightarrow {box, surprise, ...}
Adjetivo \rightarrow {white, nice, good, ...}

- Uma primeira maneira de implementar seria:

sentenca(S) \leftarrow concat(NP,VP,S) , frase_nominal(NP) , frase_verbal(VP).

Onde S,NP,VP são listas de palavras.

- Outra maneira de implementar seria usando listas de diferença.
- Seja listas $As=[1,2,3,4,5]$ e $Bs=[4,5]$ a lista de diferença entre As e Bs denotada por $As\backslash Bs=[1,2,3]$.
- $As\backslash Bs$ é chamada lista diferença e As é a cabeça e Bs a cauda.
- Listas diferença são uma outra maneira de se representar listas em Prolog.

programa: analisador sintático para uma pequena gramática do inglês.

```
sentenca(S\S0) ← frase_nominal(S\S1), frase_verbal(S1\S0).  
frase_nominal(S\S0) ← artigo(S\S1), frase_nominal2(S1\S0).  
frase_nominal(S) ← frase_nominal2(S).  
frase_nominal2(S\S0) ← adjetivo(S\S1), frase_nominal2(S1\S0).  
frase_nominal2(S) ← nome(S).  
frase_verbal(S) ← verbo(S).  
frase_verbal(S\S0) ← verbo(S\S1), frase_nominal(S1\S0).  
artigo([the|S]\S).      artigo([a|S]\S).  
adjetivo([white|S]\S).  adjetivo([nice|S]\S).  
nome([box|S]\S).        nome([surprise|S]\S).  
verbo([contains|S]\S).  verbo([are|S]\S).
```

Exercício 1: sentença(the white box contains a surprise)?

Exercício 2: como modificar a gramática a cima para o português.