
**Inteligência Artificial**
**Revisão de Prolog**

**CCENS UFES – Departamento de Computação**
Prof. M. Sc. Jacson Rodrigues Correia da Silva

Observações:

- Utilizem o SWI-Prolog: <http://www.swi-prolog.org/download/stable>
- No Linux, para executar o programa, faça:
  - Abra o terminal na pasta que estão seus arquivos Prolog (*.pl);
  - Execute: swipl
    se não estiver instalado, faça a instalação com: sudo apt-get install swi-prolog
  - Carregue seu arquivo de regras e fatos. Exemplo com "*familia.pl*":
    ?- consult(*familia*).
    ou
    ?- ['familia'].
    - Você também pode carregá-lo diretamente do terminal: swipl -s *familia.pl*
  - Ao mudar algum fato ou regra, recarregue o arquivo:
    ?- make.
  - Saia do SWI-Prolog:
    ?- halt.
    ou
    ?- Pressionando CTRL+D

## Exercícios

1. No arquivo "*familia.pl*" Crie fatos sobre sua família, como por exemplo:
   - `pai( alfredo, suse ).`
   - `pai( alfredo, joana ).`
   - `pai( jose, alfredo ).`
   - `casado( alfredo, ingred ).`

   Note que:
   Todos as regras com o mesmo nome devem ficar juntas e a ordem importa, pois o prolog pesquisará a primeira regra, para depois a segunda e assim sucessivamente.
   Não pode haver espaço em branco entre o nome da regra e o parêntesis. Assim, "*pai (* "daria erro.
   Ao final de cada linha, deve-se terminar a declaração com o sinal de ponto.
2. Teste o arquivo, abrindo-o no prolog e fazendo perguntas. Exemplo:
   - `Quem é o pai de suse?`
     `?- pai( Pai, suse ).`
     Note que *Pai* com *P* maiúsculo é a variável e que *pai* com *p* minúsculo é a regra.
   - `Quem é filho de alfredo?`
     `?- pai(alfredo, Filho).`
3. No arquivo família, faça regras para mãe, irmão, irmã, etc. Exemplo:
   - `mae( Mae, Filho ) :- pai( Pai, Filho ), casado( Pai, Mae ).`

Agora vamos trabalhar com listas:

4. No prolog, digite as seguintes perguntas, entenda as respostas e descreva como foram seus casamentos (*matchings*):
   - `[1, 2, 3, 4] = [1, 2, 3, 4].`
   - `[1, 2, 3, 4] = [A|B].`
   - `[1, 2, 3, 4] = [A,B|C].`
   - `[1, B, 3, 4] = [1, 2, 3, D].`
   - `[A, B, 3, 4] = [1, 2, 3, 4].`
   - `[A, B | [3, 4]] = [1, 2, 3, 4].`

- `[A, B | C] = [1, 2, 3, 4].`
- `[A, B | [C, D]] = [1, 2, 3, 4].`
- `[A, B | [C, D]] = [1, 2 | E].`
- `[1 | [2,3]] = A.`

Agora, vamos trabalhar na inversão de uma lista:
Com as seguintes regras:

```
% base:
inverte([], R).
% passo:
inverte([A|B], R) :- inverte(B, [A|R]).
```

Ao perguntar:

```
?- inverte([1,2,3], R).
```

Temos **true**, porém não temos a variável R unificada.
Isso acontece, pois o prolog faz a unificação assim:

```
1 → inverte( [1 | [2,3]], R ) :- inverte( [2,3], [1 | R] ).
2 → inverte( [2 | [3]]  , R ) :- inverte( [3]  , [2, 1 | R] ).
3 → inverte( [3 | []]   , R ) :- inverte( []   , [3, 2, 1 | R] ).
4 → inverte( [], R ).
```

Como vocês viram, a variável nunca é preenchida.

Por isso, as regras corretas devem utilizar uma lista auxiliar:

```
% base:
inverte([], L, L).
% passo:
inverte([A|B], Aux, Resp) :- inverte(B, [A|Aux], Resp).
```

Sua execução é:

```
?- inverte([1,2,3], [], R).  %inciamos Aux com lista vazia: []
1 → inverte( [1 | [2,3]],    [], R ) :- inverte( [2,3], [1|[]]    , R] ).
2 → inverte( [2 | [3]]  ,   [1], R ) :- inverte( [3]  , [2|[1]]   , R ).
3 → inverte( [3 | []]   , [2,1], R ) :- inverte( []   , [3|[2, 1]], R ).
Nesse próximo casamento:
    inverte( [], [3,2,1], L ).
A variável R deve se casar com [3,2,1], pois a regra é: inverte([],L,L).
Então:
4 → inverte( [], [3,2,1], [3,2,1] ).
```

Assim, para que a variável *R* case com as regras que temos, ela deve assumir o valor `[3,2,1]`.

Outra característica útil para ser utilizada em regras recursivas é o corte (poda, cut).
Suponha que tenhamos as seguintes premissas:

```
Se X é menor que 3, então Y = 0.
Se X é maior ou igual a 3 e é menor que 6, então Y = 2.
Se X é menor ou igual a 10, então Y = 4.
```

Temos:

```
X < 3 → Y = 0
X ≥ 3 ∧ X < 6 → Y = 2
X ≤ 10 → Y = 4
```

Que podem ser escritas da seguinte forma em Prolog:

```
regra(X, 0) :- X < 3.
regra(X, 2) :- X >= 3, X < 6.
regra(X, 4) :- X =< 10.
```

Ao testar com uma pergunta como a de baixo, temos:

```
?- regra(2, Y).
Y = 0 ;
Y = 4.
```

Ao invés de fornecer somente Y=0, temos também Y = 4.
Isso acontece porque o Prolog executa o *backtracking* para testar todas as possibilidades.

Para evitar isso, podemos fazer o corte:

```
regra(X, 0) :- X < 3, !.
regra(X, 2) :- X >= 3, X < 6, !.
regra(X, 4) :- X =< 10, !.
```

5. Teste essas regras e veja se o resultado agora está correto.

Baseando-se nesses conceitos de lista, crie as regras necessárias para:

6. Verificar se o elemento Elem existe na lista. Ex:

```
membro(Lista, Elem).
?- membro([3,1,7,2,0], 7).
true.
```

7. Encontrar o elemento da posição N de uma lista. Ex:

```
itemDaLista(Lista, Elem, N).
?- itemDaLista([3,1,7,2,0], Elem, 3).
Elem = 7.
```

8. Contar quantos elementos Elem existem na lista. Ex:

```
contarElemDaLista(Lista, Elem, Qnt).
?- contarElemDaLista([3,2,3,1,3,8,3], 3, Qnt).
Qnt = 4.
```

9. Encontrar números consecutivos em uma lista. Ex:

```
consecutivosNaLista(Lista, 3).
?- consecutivosNaLista([1,2,3,3,4], 3).
true.
```

10. Contar o tamanho da lista. Ex:

```
tamLista(Lista, Tam).
?- tamLista([3,5,8,2,4,9], Tam).
Tam = 6.
```

11. Remover um item da lista:

```
removerItemDaLista(Lista, Item, NovaLista).
?- removerItemDaLista([3,5,8,2,4,9], 4, NovaLista).
NovaLista = [3,5,8,2,9].
```

12. Fornecer o último elemento de uma lista.

```
ultimoElemDaLista(Lista, Elem).
?- ultimoElemDaLista([3,2,3,1,3,8], Elem).
Elem = 8.
```

13. Encontrar o maior elemento da lista.

```
maiorElemDaLista(Lista, Elem).
?- maiorElemDaLista([3,2,3,1,3,8], Elem).
Elem = 8.
```

14. Encontrar o menor elemento da lista.

```
menorElemDaLista(Lista, Elem).
?- menorElemDaLista([3,2,3,1,3,8], Elem).
Elem = 1.
```

15. Somar todos os número da lista.

```
somarLista(Lista, Soma).
?- somarLista([3,2,3,1,3,8], Soma).
Soma = 20.
```

16. Encontrar o índice de um número (da esquerda para a direita). Ex:

```
indiceDe(Num, Lista, Idx).
?- indiceDe(7, [4,7,2,3,5,8], Idx).
Idx = 2.
```

17. Encontrar o índice de um número (da direita para a esquerda). Ex:

```
indiceDe(Num, Lista, Idx).
?- indiceDe(7, [4,7,2,3,5,8], Idx).
Idx = 5.
```

18. Encontrar qual é o índice que tem o menor número da lista.

```
indiceMenorDaLista(Lista, Idx).
?- indiceMenorDaLista([4,7,2,1,5,8], Idx).
Idx = 4.
```

19. Encontrar qual é o índice que tem o maior número da lista.

```
indiceMaiorDaLista(Lista, Idx).
?- indiceMaiorDaLista([4,7,2,1,5,8], Idx).
Idx = 6.
```

20. Inserir um item em uma lista já ordenada. Ex:

```
inserirNaListaOrdenada(Item, Lista, ListaOrd).
?- InserirNaListaOrdenada(4, [3,6,7,8,10], ListaOrd).
ListaOrd = [3,4,6,7,8,10].
```

21. Ordenar uma lista. Ex:

```
ordenarLista(Lista, ListaOrd).
?- ordenarLista([2,1,3,4,2,5,6], ListaOrd).
ListaOrd = [1,2,2,3,4,5,6].
```

22. Somar um inteiro a todos os itens de uma lista. Ex:

```
somarIntNaLista(Lista, Numero, ListaSomada).
?- somarIntNaLista([1,2,3,4,5], 2, ListaSomada).
ListaSomada = [3,4,5,6,7].
```

23. Cria uma lista com todos os filhos apresentados no exercício 1. Ex.:

```
listaDeFilhos(Pai, Lista).
?- listaDeFilhos(alfredo, L).
L = [suse, joana].
```