Tutorial de Prolog
"**Pro***gramming in* **Log***ic*"
(Programando em Lógica)

Claudio Cesar de Sá
(c_cesar@joinville.udesc.br)
UDESC/CCT/DCC - Campus Universitário
Bloco F - 3o. Andar - Sala F. Gauss
89.223-100 - Joinville - SC

Versão 2.0
Joinville, 21 de Maio de 2001

21 de Maio de 2001

Conteúdo

Lista de Figuras

1 Introdução

1.1 Comece aqui ...

Antes de iniciar a leitura deste tutorial, assegure que "*tenhas em mãos*" uma das últimas versões do SWI-Prolog http://swi.psy.uva.nl/projects/SWI-Prolog/download.html, para Windows (95, 98 ou NT) ou Linux ("*Unix-like*"). Neste site, bem como na Internet (ver alguns destes na seção 6), encontra-se disponível uma boa documentação sobre a linguagem Prolog.

Nestas notas, apenas os principais tópicos da linguagem são apresentados, outros detalhes "*o aluno terá que correr atrás*". Ou seja, nestas páginas o aluno tem um "*guia de sobrevivência*" com o Prolog, pois consta a parte mais complicada da linguagem. Os livros e manuais sobre o assunto são indispensáveis, e se encontram disponíveis na Web.

Os termos enfatizados ou em itálicos, são explicados em sala de aula, bem como, parte do material complementar da disciplina. Eventualmente, o aluno deve repassar todas estas notas, antes de insistir numa compreensão sequencial do texto. A propósito, o Prolog não apresenta o conceito de sequencialidade e fluxo linear de programas, a exemplo de outras linguagens tradicionais de programação.

Finalmente, este tutorial encontra-se "*relaxado*" no que concerne a terminologia de lógica e do Prolog. A proposta é tornar o texto acessível a todas as pessoas que não tenham base em lógica, e até mesmo nunca tenham utilizado nenhuma outra linguagem de programação.

1.2 Noções Iniciais

Definição: Prolog é uma linguagem de programação implementada sobre um paradigma lógico. A base de tal lógica é o Cálculo dos Predicados ou Lógica de 1^a. Ordem (LPO). Logo, a concepção da linguagem apresenta uma terminologia própria, mas fundamentada sobre a LPO. As restrições da completude e corretude da LPO são contornadas com um método de prova sistemático e completo.
Características:

- Manipula símbolos (objetos) por natureza, logo "*7 + 13*" pode apresentar várias conotações;
- Átomo, literal ou um objeto: 'a', 77, "aa", '7' , "777" ...
- Seu princípio inferencial é o "*backward chaining*" (*encadeamento regressivo*), ou seja, para encontrar algum símbolo como verdade, devemos demonstrar que suas premissas eram também verdadeiras;
- Apresenta um propósito geral como Linguagem de Programação, bem como tem interface com linguagens como C e outras;
- Portabilidade entre plataformas para o Prolog padrão. Logo, não é uma linguagem proprietária;
- Fácil aprendizado;
- Áreas de aplicações: problemas de IA, provadores de teoremas, sistemas especialistas, pesquisa operacional, construção de compiladores, etc.

1.3 Exemplos

Uma vez instalado o Prolog em questão, o ambiente interpretado do Prolog apresenta um "*prompt default*" ao ser executado: **?-**

Veja os exemplos iniciais:

```
?- 2 > 3.
No
?- 2 == 2.
Yes
?- mae(eva, abel).
```

```
No
?- 3 > 2.
Yes

?- consult('teste.pl').
% teste.pl compiled 0.00 sec, 560 bytes
Yes
?- homem( X ).
X = joao
Yes
?- homem( x ).
No
```

Adiantando, as letras maiúsculas são as variáveis da linguagem Prolog, isto é, aceitam qualquer objeto.

1.4 Sintaxe do Prolog

A sintaxe "**Prologuiana**" é simples. Esta é construída a partir de formulações com os predicados lógicos, logo, o conceito de "*predicado*" reflete o "*espírito*" do Prolog.

As construções (*linhas de código em Prolog*) seguem uma sintaxe de três tipos.

1o. tipo: são as questões ou *"goals"*, isto é, uma pergunta à uma base de conhecimento (que fica instanciado e referenciado neste ambiente inferencial do Prolog)
... *"a idéia a de um teorema a ser demonstrado"* ...

```
?-  >(3,2).    /* ou  ?-  3 > 2.      Ou seja,   3  é maior que 2  */
?-
Yes
```

2o. tipo: são os fatos ... algo sempre verdadeiro ou *verdades incondicionais*, encontrados na "*base de conhecimento*".

```
?- listing(homem).
homem(joao).
homem(jose).
homem(jedro).
Yes
```

3o. tipo: são as regras ... que "*aproximadamente*" são verdades ou teoremas condicionais, isto é: *necessitam de uma prova lógica*!

```
?- listing(mortal).
mortal(A)  :-   homem(A).
/*  obs.     :-   eh o implica ao contrário  <-   */
/*  homem ---> mortal
 leia-se ... para demonstrar que algum X é
```

```
 mortal, preciso demonstrar e provar que A eh
 um homen  */
Yes
```

Conclusão: *a construção de um programa em Prolog, é feita de: fatos, questões e regras... e nada mais!*

1.5 Detalhando a sintaxe

Há alguns detalhes de como se montam esses predicados "*prologuianos*". Essa "*Receita de Bolo*" segue abaixo:

- Os predicados devem ser em letras minúsculas.... *OBRIGATORIAMENTE*!
- Os fatos, regras e questões terminam por "." (ponto);
- Seja uma regra, exemplo: $\boxed{\text{irmão(X,Y) :- pai(X,Z), pai(Y,Z), X \textbackslash == Y.}}$. Os seguintes elementos são identificados:
  - "*irmão/2*", "*pai/2*" e "\ == */2*", são predicados binários (i. é. aridade igual a dois);
  - A vírgula (",") é o "*and*" lógico;
  - O "*or*" lógico é o ponto-e-vírgula (";") ;
  - O ":- " e o "*pescoço*" da regra;
  - O ponto no final "." é o "*pé*" da regra;
  - Logo a regra possui uma "*cabeça*", que é o predicado "*irmão/2*";
  - O corpo é composto pelos predicados: "*pai/2*" e "\ == */2*".
- Os nomes que começam por letras maiúsculas são variáveis no Prolog. As variáveis podem receber números, letras, frases, arquivos, regras, fatos, ... até mesmo outras variáveis, mesmo sendo desconhecidas !
- A variável anônima "_" (o "*underscore*") não necessita ser instanciada, i.é. não impõe restrições quanto ao seu escopo;
- Nomes que começam por letras minúsculas são símbolos no Prolog;
- O escopo é a validade da instância de uma variável na regra;
- As variáveis possuem o seu escopo apenas dentro de suas regras;

1.6 Outros conceitos

1. Aridade: Número de argumentos do predicado. Leia-se que "*capital(paris, França)*" é um predicado de aridade dois;

2. "*Matching*": Ao processar uma busca na *Memória de Trabalho* (MT), a máquina inferência do Prolog realiza verificações do início do programa para o final. Encontrando um predicado de mesmo nome, em seguida verifica a sua aridade. Se nome e aridade "*casarem*", o "*matching*" foi bem sucedido;

3. Instância: Quando uma variável de uma regra for substituída por um objeto, esse é dito ser uma instância temporária à regra;

4. Unificação: Quando uma regra for satisfeita por um conjunto de objetos, se encontra um instância bem sucedida para regra inteira. Neste caso, as variáveis foram unificados pelos objetos, resultando em uma regra verdade e demonstrável. Em resumo, é uma instância que foi bem sucedida;

5. "*Backtracking*": Esse conceito é particular ao Prolog, e o diferencia das demais linguagens convencionais de computador. Basicamente, *e não completamente*, o conceito de "*backtracking*" é o mesmo da Engenharia de Software. Ou seja, na falha de uma solução proposta, o programa deve retroceder e recuperar pontos e/ou estados anteriores já visitados, visando novas explorações a partir destes. Como exemplo ilustrativo, imaginemos uma estrutura em árvore qualquer, e em que um dos nós terminais (folhas) exista uma ou mais saídas. Qual a heurística de encontrar uma das saídas ou o nó desejado? Várias abordagens podem ser feitas, assunto esse que foge da discussão no momento. O Prolog usa uma busca em profundidade ("*depth-first*"), cujo retrocesso ocorre ao nó mais recente, cuja visita foi bem sucedida. Esses nós referem aos predicados que formam um corpo da árvore corrente de busca. Logo, várias árvores são construídas, cada vez que uma nova regra é disparada. Mas o controle dessas árvores e seu objetivo corrente, é implementado como uma estrutura de pilha, em que nós não solucionados são empilhados.

1.7 Operações básicas do SWI-Prolog

Exceto o Visual Prolog (www.pdc.dk ou www.visual.dk), todos os demais Prolog são basicamente idênticos. Atualmente deve existir ter uns dez (10) fabricantes de Prolog comerciais e "*um outro tanto*" oriundos da academia. Todos oferecem algo gratuito como chamariz ao seu sistema.

O SWI-Prolog não foge a regra, as suas características são: velocidade, portabilidade (interoperacionabilidade), padrão, robustez, facilidades de uso, interface com várias outras linguagens de programação, "*free-source*", etc.

Os passos básicos para se usar o SWI-Prolog em seu ambiente interpretado são:

1. Editar o programa. Use o "*bloco de notas*" do windows, ou o "*edit*" do DOS. Preferencialmente editores ASCII;
2. Carregar o programa usuário para o ambiente interpretado SWI-Prolog;
3. Executar os predicados;
4. Validar os resultados;
5. Voltar ao passo inicial, se for o caso.

- Para carregar o programa: *pl* + <Enter>

```
Welcome to SWI-Prolog (Version 3.2.3)
Copyright (c) 1993-1998 University of Amsterdam.  All rights reserved.
For help, use ?- help(Topic). or ?- apropos(Word).
Yes
?-
```

- Para sair do ambiente SWI-Prolog:

```
?- halt.
```

- Para carregar um programa na Memória de Trabalho (MT) ("**Working Memory-WM**", ou "*base dinâmica de conhecimento*"):

```
?- consult('d:/curso/udesc/autind/iia/ex1').
d:/curso/udesc/autind/iia/ex1 compiled, 0.00 sec, 52 bytes.
Yes
?-
ou
?-  load_files('nomes','opcoes').
/* para conferir....se está na ``{\em Working  Memory}'' */
?- ensure_loaded(rosa).
Yes
?-
```

- Para eliminar um predicado da WM:

```
?- abolish(cor/1).
Yes
?- listing(cor/1).
[WARNING: No predicates for `cor/1']
No
?-
```

- Para executar uma regra:

```
?- eh_maior_que_10.
 DIGITE UM NUMERO:: 30
|    .  < ......... faltou o ponto . >
 numero maior que 10
Yes
?-
```

- Para "*bisbilhotar*" o "*help*", bem como inferir novas características:

```
?- help(nome_do_predicado).
ou
?- apropos(padrão_desejado).
ou
?- explain(nome_do_predicado).
?- apropos(setenv).
setenv/2                      Set shell environment variable
unsetenv/1                    Delete shell environment variable
Yes
?-
```

Isto é, pretendendo fazer uma leitura do teclado, mas não conhece o predicado que faz tal função, procure os predicados relacionados com o "*apropos*", algo como: ?- apropos(read).

- As imperfeições do SWI-Prolog, ... *não pode existir espaço entre o nome do predicado e o início dos argumentos, no caso o "(...)"*. Este erro é bastante comum entre os iniciantes do Prolog. Vejao exemplo abaixo no predicado "¿":

```
?-   >    (9, 5).
[WARNING: Syntax error: Operator expected
> (9, 5
** here **
) . ]
?-  >(9, 5).
Yes
?-
```

- Para entrar no módulo de depuração ou debugger:

```
?- trace, predicado.
ou
?- debug, predicado.
ou
?- spy(predicado_em_particular).
para desativar:
?- nospy(predicado_em_particular).
e
?- notrace.
e
?- nodebug. /* ou Crtl-C para sair do modo debug */
```

1.8 Outros Detalhes

- Correções automáticas:

```
?- patos(X,Y).
Correct to: 'pratos(X, Y)'? yes
X = alho
Y = peixe ;        /*   ; ^ */
/*  ;  para forçar uma nova resposta manualmente */
X = cebola
Y = peixe ;
X = tomate
Y = peixe ;
No
?-
```

- Forçando todas as respostas:

```
?- prato(X, Y),  write(X), nl, fail.
alho
cebola
tomate
No
?-
```

Logo o "*fail*" é um predicado que provoca uma falha na regra, ou seja, torna-a sempre falsa; esse "*fail*" força o mecanismo "*backtracking*".

- "*Apavoramentos*" com o modo "*trace*":

```
===> <abort.> sai do debug/trace ou da pendencia...
ou <Crtl C> +  <a> de abort
ou
?- abort.
Execution Aborted
Typing an 'h' gets us:
----
a:                 abort      b:                 break
c:                 continue   e:                 exit
g:                 goals      t:                 trace
h (?):             help
Action (h for help) ?
----
`What does this mean?`, that's my first question.
'e'    exits the program immediately
'a'    aborts whatever you've type so far
'c'    continues where you left off
'h'    gets us the 'Action' help menu, again
So what does 'g', 't' and 'b' do?
I don't know yet, but I suspect:
'g'    lists possible goals ===> os pendentes
't'    traces the program, a debugging tool
'b'    breaks the current script (whatever your working on)
       and give a new shell.
```

- Não esqueça que:

```
?- trace, t.
e
?- spy(t), t.
```

são equivalentes !

2 Recursividade

A recursividade em Prolog é a sua própria definição em lógica. Apesar da ineficiência de soluções recursivas, esta é uma das elegâncias do Prolog, ver exemplos. A propósito, caso tenhas compreendido o exemplo da seção anterior, pule esta seção.

A exemplo das demais linguagens de programação, uma função recursiva é aquela que busca em si próprio a solução para uma nova instância, até que esta encontre uma instância conhecida e retorne um valor desejado. A elegância do conceito da recursividade é a principal virtude do SWI-Prolog.

Exemplo: Calcular a soma dos primeiros n-valores inteiros:

$$S(n) = 1 + 2 + 3 + 4 + ... + (n-1) + n$$

Este problema pode ser reformulado sob uma visão matemática, mais especificamente, pela indução finita como:

$$S(n) = \begin{cases} 1 & \text{para } n = 1 \\ S(n-1) + n & \text{para } n \geqslant 2 \end{cases}$$

O que é um fato verdadeiro pois:

$$S(n) = \underbrace{1 + 2 + 3 + ... + (n-1)}_{S(n-1)} \quad +n$$

Como o procedimento é recursivo, temos encontrar a definição para "**parada**" da recursividade. Como n não tem limite superior, é para qualquer n, então vamos começar pelo que conhece:

#1. A soma de 1 é 1, logo: soma(1,1).
#2. Para soma dos n-ésimos termos, é necessário a soma do $(n-1)$-ésimos termos, logo: soma(N,S) ... = ... Nant = (N-1), soma(Nant, S_Nant) e S = (N + S_Nant).

A regra #1, "*soma(1,1).*", é conhecida como condição ou *regra de parada*, ou ainda, *regra de aterramento*. Pelo fato de que o Prolog inicia seu procedimento sequencial "*de cima para baixo*", a condição #1 deve vir antes de #2. Em casos raríssimos, a regra recusirva, regra #2, vem antes da regra #1, contudo, há exemplos de tais casos. O exemplo acima reescrito em Prolog, é dado por:

```
s(1,1) :- true.                            /* regra #1 */
s(N, S) :- N > 1,                        /* regra #2 */
    Aux is (N-1),
    write('*'),
   s(Aux, Parcial),
   S is (N + Parcial).
```

O procedimento recursivo é típico da movimentação da pilha intrínseca do Prolog. O quadro abaixo ilustra o exemplo de um "*goal*" do tipo:

?-s(5,X).

Chamada Pendente	**Regra Casada**	**N**	**N1**	**Parcial**	**S**
s(5,X)	#2	5	4	?...<10>... →	...<15>...
s(4,X)	#2	4	3	?...<6>... →	↖...<10>...
s(3,X)	#2	3	2	?...<3>... →	↖...<6>...
s(2,X)	#2	2	1	?...<1>... →	↖...<1>...
s(1,X)	#1 (aterrada)	1	-	-	↖1

Duas observações:

1. A rigor, internamente, a recursividade não procede exatamente como descrito no quadro acima. Mas é algo muito "*próximo*" a isto;
2. Nem todo conjunto de regras recursivas são passíveis de admitirem tal quadro.

2.1 Exemplos sobre Recursividade

1. **Um simples sistema de menu**: além do uso da recursividade, este exemplo mostra o que fazer quando aparentemente não há predicados disponíveis ao que se deseja; ou seja: "$X <= 3$" não existe, mas como opção equivalente temos:

   - "$X =< 3$" ;
   - "$\backslash + (X >= 3)$".

```
menu(0).
menu(_) :-
        repeat,
        write('......'), nl,
        ......................
        write('......'), nl,
        write(' DIGITE A OPCAO: '),
        read(X),
        X >= 0,
        \+(X >= 3), /* isto é: X =< 3 */
        /*   X \== 0    é equivalente a:  \+(X == 3 ) */
        acao(X),
        menu(X).
%%%%%%%%%%%%%%%%%%%%%%%%%%%
        acao(0).     /* continua acoes etc */
        acao(1).
        acao(2).
        acao(3).
```

2. **Novamente o "Sócrates"**:

```
        human(socrates).          % facts about who is human
        human(aristotle).
        human(plato).
        god(zeus).                      % and who is a god
        god(apollo).
        mortal(X) :-human(X).   % a logical assertion that X is mortal if X is

Fazendo as perguntas:
        ?-mortal(plato).                % is Plato mortal?
        yes
        ?-mortal(apollo).               % is apollo mortal?
```

```
no
?-mortal(X).                    % for which X is X mortal?
X = socrates ->;
X = aristotle ->;
X = plato ->;
no                               %human
```

Evitando este ultimo "no", usando a recursividade:

```
mortal_report :-
        write('Mostre todos mortais conhecidos:'), nl, nl,
        mortal(X),
        write(X), nl,
        fail.
mortal_report.                /* ou::    mortal_report :- true. */
```

Entao:

```
?- mortal_report.
Mostre todos mortais conhecidos:'
socrates
aristotle
plato
yes
```

3. **Cálculo do fatorial**: Reformulando sob uma visão matemática, mais especificamente, pela indução finita temos:

$$Fat(n) = \begin{cases} 1 & \text{para } n = 0 \\ Fat(n-1) * n & \text{para } n \geqslant 1 \end{cases}$$

O que é um fato verdadeiro pois:

$$Fat(n) = \underbrace{1 * 2 * 3 * \ldots.. * (n-1)}_{Fat(n-1)} \quad * \, n$$

Como o procedimento é recursivo, temos de encontrar a definição para "**parada**" da recursividade. Como n não tem limite superior, é para qualquer n, então vamos começar pelo que conhece:

#1. O fatorial de 0 é 1, logo: fatorial(0,1). **#2**. O fatorial do n-ésimo termo, é necessário o fatorial do $(n-1)$-ésimo termo, logo: fatorial(N, Fat) :: Nant = (N-1), fatorial(Nant, Fat_Nant) e Fat = (N * Fat_Nant).

em termos de Prolog temos:

```
fatorial( 0, 1 ).
fatorial( X, Fat ) :-
                         X  >  0,
                        Aux  is  (X - 1),
                        fatorial( Aux , Parcial ),
                        Fat  is  ( X   *  Parcial ).

Como eh apresentado por outro professor:
factorial(0, 1).
factorial(N, M):- N1 is N - 1,
                            factorial (N1, M1),
                            M is N*M1.

We can now read the Prolog code:
factorial(0, 1).   /* It is a fact that the factorial of 0 is 1*/
factorial(N, M) :- N1 is N - 1,   /* If temporary variable N1 is assigned to N-1*/
factorial(N1, M1),   /* and if the factorial of N1 is M1*/
M is N * M1.    /* and if M is assigned to N*M1*/
                        /* then the factorial of N is M    */
```

Complete a tabela abaixo, para descrição do fatorial:

Chamada Pendente	**Regra Casada**	**X**	**Aux**	**Parcial**	**Fat**
fatorial(5,X)	#	......	......	?...... $\rightarrow$	......
fatorial(4,X)	#	......	......	?...... $\rightarrow$	$\nwarrow$......
fatorial(3,X)	#	......	......	?...... $\rightarrow$	$\nwarrow$......
fatorial(2,X)	#	......	......	?...... $\rightarrow$	$\nwarrow$......
fatorial(1,X)	#	......	......	?...... $\rightarrow$	$\nwarrow$......
fatorial(0,X)	#	......	......	?...... $\rightarrow$	$\nwarrow$......

4. **Os ancestrais**: O ancestral de X, é no mínimo o pai de X. Logo, um avô X também é seu ancestral. O pai deste avô também é antepassado de X, e assim sucessivamente. Generalizando este conceito de ancestralidade, veja figura 4, em termos da relação *pai*’.

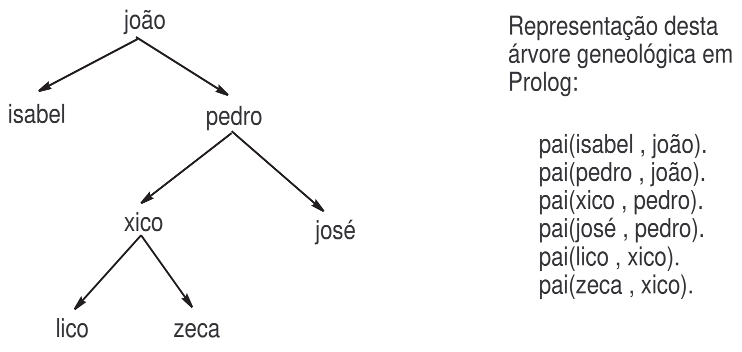

Figura 1: Uma árvore que representa os descendentes de alguém

Escrevendo este conceito em Prolog, temos:

```
ancestral(X,Y) :- pai(X,Y).
ancestral(X,Y) :- pai(X,Z), ancestral(Z,Y).
```

5. "**O caminho do português**": A proposta do problema é encontrar o custo entre dois vértices quaisquer em um grafo orientado dado pela figura 5. O nome do problema define a proposta do exercício. Este grafo não apresenta ciclos, e nem é bidirecional entre os vértices.

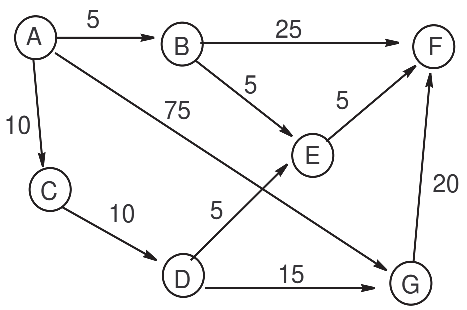

Figura 2: Um grafo que define custos entres as cidades

```
ligado(a,b,5).      ligado(a,c,10).     ligado(a,g,75).
ligado(c,d,10).    ligado(d,g,15).     ligado(d,e,5).
ligado(g,f,20).    ligado(e,f,5).       ligado(b,f,25).
ligado(b,e,5).    ligado(b,f,25).

rota(X,Y,C) :- ligado(X,Y,C).
rota(X,Y,C) :- ligado(X,Z,C1),
                      rota(Z,Y, C2),
                      C is (C1 + C2).

?- rota(a,g,X).
X = 75 ;
X = 35 ;
No
?-
```

3 Functores

3.1 Definições

Functores: são funções lógicas, logo a equivalência entre um domínio e imagem é identificada. Os functores permitem contornar o mapeamento típico sobre $\{V, F\}$ ("*yes*" ou "*no*", conforme o aluno já tenha percebido); extrapolando com isso a um domínio de objetos mapeados à objetos.

Exemplo: seja a descrição de uma classe de objetos do tipo carro:

```
carro(Nome, estilo(
        esporte(Portas, opcionais(L1,L2)),
        passeio(Portas),
        off_road(L3, motor(Comb, outros(Turbo, L4)))
                            )    ).
```

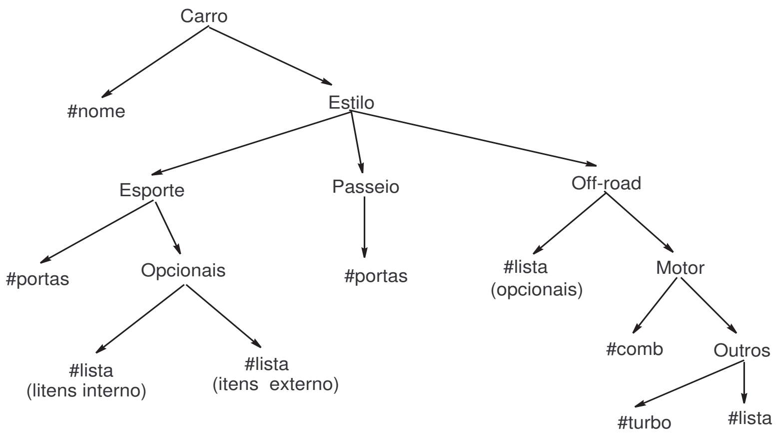

Figura 3: Representando o conceito de hierarquia com um functor

3.2 Motivação

A principal observação do exemplo acima é a hierarquica "*natural*" e estabelecida, para representar um conhecimento. Tal representação é conhecida por "*frames*", que incorpora a maioria dos conceitos da programação orientada-à-objetos. Em resumo temos:

1. Construção de fatos genéricos (próximo ao conceito de classes, instâncias, etc.);
2. Quantidade variável de argumentos, leva há novos predicados. Há com isso *um certo ganho* de generalidade, pois um mesmo predicado pode ser utilizado para descrever vários tipos de objetos, buscas sobre padrões são facilitadas;
3. Legibilidade dos dados, e consequentemente o entedimento do problema;
4. Em resumo, os functores podem serem vistos como uma estruturação os dados sob formas dos "*REGISTROS*" convencionais.

3.3 Exemplo

Seja o programa abaixo

```
artista("Salvador Dali", dados(pintor(surrealista), regiao( "Catalunha" ), 86, espanhol)).
artista("Pablo Picasso", dados(pintor(cubista), 89, espanhol)).
artista("Jorge Amado", dados(escritor(contemporaneo), 86, brasileiro)).
```

Para verificar seu aprendizado, verifique e entenda o porquê das perguntas e repostas abaixo:

1. Dentro do predicado "*artista*" há outros predicados?
   Resp: Errado... não são outros predicados ... e sim **funções lógicas**!
2. Quantos argumentos tem o predicado artista?
   Resp: 02 argumentos, um campo é o "'*nome*"... e o outro é a função "'*dados*" !
3. Quantos argumentos tem a função "*dados*"?
   Resp: 04 e 03 argumentos para o primeiro e segundo fato respectivamente. Logo, apesar de terem o mesmo nome, elas são diferentes!
4. Quantos argumentos tem a função "pintor"?
   Resp: 01 argumento!
5. A função "*pintor*" é equivalente a "*escritor*"?
   Resp: Não, um objeto tem características da função "*pintor*" e outro da "*escritor*";
6. Quanto ao valor de retorno dessas funções lógicas?
   Resp: Não existe... **ela por si só já é mapeada num domínio qualquer (números, letras, objetos, etc.....)**.

3.4 Discussão sobre Problemas de IA

Reescrevendo sob a forma regras o problema dos vasos (falta a listagem completa, que é apresentada em sala de aula):

```
......................................................
vaso(4,Y) :- vaso(X,Y) , X < 4.
......................................................
conjunto de regras de movimentacao de enchimento
e esvaziato. dos vasos...
........................................................
vaso(W,Z) :- vaso(X,Y), X+Y >= 4, Y > 0, Z is (Y-(4-X)),
     W is 4. /* enche o vaso A com sobra em B */
.........................................................
vaso(2,_) :- vaso(_,2).
...........................................................
```

Perguntas e Problemas:

1. Todas as regras tem que serem explicitadas?
   Resp: SIM.... são possibilidades de *movimentos* e/ou *ações* plausíveis de serem acionadas! Ex: despejar, encher, esvaziar... etc.

2. Se tudo for explícito ... logo tenho a solução do do problema?
   R: Não ... pois se tivéssemos os caminhos, ou pelo menos um, o problema seria "*algoritimizável*", portanto, livre da IA. Mas mesmo encontrando uma solução, isto ainda não garante que esta seja uma solução robusta. Pois, com uma modificação da condição inicial, ou de alguma alteração do problema, todo algorítmo "*cairia por água abaixo*". Um algorítmo funciona para uma condição estática e esperada, algo que difere de programas de IA, que são genéricos para uma classe de situações de um problema.

3. Afinal, qual a proposta da IA em termos de ferramentas para resolver problemas?
   Resp: A idéia é que estas ferramentas (ver livro texto), no caso uma linguagem de programação do tipo Prolog, use um mescanismo "*inteligente*" de tentar soluções. Ou seja, diferentemente as Linguagens tradicionais, onde problema tinha que ser descrito em termos de sua solução, no Prolog a proposta é apenas "*declarar o problema*"! Logo, o Prolog é uma linguagem *declarativa*, e não *procedural* como as demais. Sendo assim, através de fatos e regras descreve-se o problema, e é deixado ao seu sistema de dedução interno buscar às soluções possíveis!

4. A representação acima é apropriada?
   Resp: Em termos de RC (Representação do Conhecimento) a resposta é "**sim**"! Em termos de Prolog, não, pois de imediato temos os seguintes problemas:

   (a) As regras são tipicamente recursivas. Nesse momento, tal aspecto é INDESEJÁVEL! A recursividade é elegante (e muito na maioria dos casos), contudo, computacionalmente, a nível de uso/gasto de memória, é ineficiente.

   (b) Decorrente do item anterior, podem surgir laços inifinitos ("*loops*");

   (c) Não há controlabilidade de quais regras foram usadas ou não ! Pois, para o Prolog todas as regras são factíveis de serem disparadas. Óbvio que é um resultado esperado e desejável.

   (d) Então precisamos de estruturas mais poderosas que functores?
   Sim, vamos usar um tipo especial de functor chamado de "*lista*". As listas em Prolog servem como um poderoso mecanismo de armazenamento temporário. No caso, o interesse é sobre o armazenamento de regras já disparadas e estados já visitados !

3.5 Usando Functores em Problemas de IA

Retomando o ponto acima, vamos contornar em parte o problema. Seja:

```
vaso(W,Z)  :- vaso(X,Y), X+Y >= 4, Y > 0, Z is (Y-(4-X)),
               W is 4. /* enche o vaso A com sobra em B */

como reescreve-la usando o conceito de functores:

 movimento(X,Y  --->  W,Z) :- X+Y >= 4, Y > 0, Z is (Y-(4-X)), W is 4.
 movimento(vaso(X,Y), vaso(W,Z)) :- X+Y >= 4, Y > 0, Z is (Y-(4-X)), W is 4.

     Estado anterior

                     Novo estado...ou corrente !
```

3.6 Concluindo Functores

Alguns de regras e questões que podem ser encontrados a partir do exemplo acima

```
 /*  qual o nome e os dados de cada artista ? */
?- artista(X,Y), write(X), nl, write(Y),nl, fail.

 Do exemplo, deduzimos que functor ``{\em casa}'' (``{\em
matching}''  com variáveis (letras maiusculas)....

/* quem são os cubistas ?*/
?- artista(X, dados(pintor(Y) ,_ ,_ )), Y == cubista, nl,
     write(X), write('=====>'), write(Y), nl, fail.

/* quem tem mais de 80 anos ? */
?- artista(X,dados(_ , Y _)), Y > 80, nl, write(X),  write(Y),nl, fail.

/* no caso acima, nem todos resultados foram encontrados, porquê ? */
```

Resumindo: os functores organizam dados e visualizam regras recursivas de uma maneira mais simples e controlável. Outro detalhe é que como objetos, functores respeitam todas as regras de "*casamento*" vistas até o momento. Sem exceção !

4 Listas

4.1 Definições

Definições iniciais:
- Uma lista é uma estrutura de dados que representa uma coleção de objetos homogêneos;
- Uma lista é uma seqüência de objetos;
- Uma lista é um tipo particular de functor[1], pois apresenta uma hierarquia interna.

Notação: O símbolo "[" é usado para descrever o início de uma lista, e "]" para o final da mesma;

Exemplos: [a, b, c, d], logo um predicado cujo argumento seja algumas letras, temos uma lista do tipo:

```
letras( [ a, b, c, d ] ).
          ^
          |
          |
  cabeça da lista
```

Os elementos de uma lista são lidos da esquerda para direita, logo a letra "a" é o primeiro elemento ou "*cabeça*" da lista. Quanto ao resto ou "*cauda*" da lista, é uma "*sub-lista*" dada por: [b, c, d]. Esta sub-lista, de acordo com uma outra definição complementar apresentada na seção 4.2, também é uma lista.

[1] Logo, a lista vai apresentar uma hierarquia natural, internamente.

Operador —: "*Como vamos distinguir de onde se encontra a cabeça da cauda da lista?*" Como o conceito de listas introduziram novos símbolos, isto é, os seus delimitadores [...], há um novo operador que separa ou define quem é a cabeça da cauda da lista. Este operador é o "*pipe*", simbolizado por:—, que distingue a parte da esquerda da direita da lista. Isto é necessário para se realizar os casamentos de padrões com as variáveis.

Exemplos de "*casamentos*": os exemplos abaixo definem como ocorrem os casamentos entre variáveis e listas. Portanto, preste atenção em cada exemplo, bem como teste e faça suas próprias conclusões de como as listas operam.

```
[ a, b, c, d ] == X
[ X | b, c, d ]  ==  [ a, b, c, d ]
[ a | b, c, d ]  ==  [ a, b, c, d ]
[ a , b | c, d ]  ==  [ a, b, c, d ]
[ a , b , c | d ]  ==  [ a, b, c, d ]
[ a , b , c , d | [] ]  ==  [ a, b, c, d ]
[] == X
[ [ a | b , c , d] ] == [ [ a , b , c , d ] ]
[  a | b , c , [ d ] ] == [  a , b , c , [ d ] ]
[  _ | b , c , [d ] ] == [  a , b , c , [ d ] ]
[  a | Y ] == [  a , b , c ,  d ]
[  a | _ ] == [  a , b , c ,  d ]
[  a , b | c , d ] == [  X , Y | Z ]
```

Contra-exemplos de "*casamentos*": Explique porque nos exemplos abaixo, **não** ocorre o casamento de padrões:

```
[ a , b | [c, d] ]  \==  [ a, b, c, d ]
[ [ a , b , c , d] ]  \==  [ a, b, c, d ]
[  a , b , [ c ] , d, e ]  \==  [ a, b, c, d, e ]
[ [ [ a ] | b , c , d] ] \== [ [ a , b , c , d] ]
```

Enfim, os tipos de casamentos de objetos de uma lista, segue o mesmo padrão dos "*matching*" considerados até o momento em Prolog.

Aplicação: Devido ao fato de listas modelarem qualquer estrutura de dados, invariavelmente, seu uso é extensivo em problemas de IA, pois envolvem buscas sobre estas estruturas.

4.2 Exemplos

Retomando ao conceito de listas, se "*auto-define*" o conceito de lista com os seguintes axiomas:

1. "*Uma lista vazia é uma lista*";
2. "*Uma sub-lista é uma lista*".

As definições acima são recorrentes, isto é, uma depende da outra. Em outras palavras, temos uma definição recursiva uma vez mais. Sendo assim, reescrenvendo em Prolog tal definição é dada por:

```
#1          é_uma_lista( [ ] ).                              /* 1a. premissa */
#2          é_uma_lista( [X | T ] ) :- é_uma_lista( T ).    /* 2a. premissa*/

?- é_lista( [a,b,c] ).
yes
```

Um "*mapa de memória aproximado*" é dado por:

	Regra	X	T
é_uma_lista([a,b,c])	#2	a	[b,c]
é_uma_lista([b,c])	#2	b	[c]
é_uma_lista([c])	#2	c	[]
é_uma_lista([])	#1	–	–

Basicamente, quase todas operações com listas possuem regras análogas a definição acima. O exemplo anterior serve apenas para identificar que o objeto: [a,b,c,d], é uma lista. Contudo, regras sobre listas são inúmeras e bastante elegantes, algumas são mostradas nos exemplos que se seguem:

1. **Comprimento de uma lista**:

   ```
   #1        compto([ ], 0).
   #2        compto([X | T], N):- compto(T, N1), N is N1+1.

   ? - compto([a, b, c, d], X).
   X = 4
   ```

   Um "*mapa de memória aproximado*" é dado por:

	Regra	X	T	N1	N is N+1
compto([a,b,c,d],N)	#2	a	[b,c,d]	3 $\rightarrow$	3+1=4
compto([b,c,d],N)	#2	b	[c,d]	2 $\rightarrow$	$\nwarrow$ 2+1
compto([c,d],N)	#2	c	[d]	1 $\rightarrow$	$\nwarrow$ 1+1
compto([d],N)	#2	d	[]	0 $\rightarrow$	$\nwarrow$ 0+1
compto([],N)	#1	–	–	–	$\nwarrow$ 0

2. **Concatenar ou união de duas listas**: Em inglês este predicado[2] é conhecido como "*append*", e em alguns Prolog's pode estar embutido como predicado nativo:

   ```
   #1 uniao([],X,X).
   #2 uniao([X|L1],L2,[X|L3]) :- uniao( L1, L2, L3).

   O ''goal'':
   ?- uniao([a,c,e],[b,d], W).
   W=[a,c,e,b,d]
   yes
   ```

[2] A palavra predicado, neste contexto, reflete o conjunto de regras que definem as operações dos mesmos sobre listas

Um "*mapa de memória aproximado*" é dado por:

	Regra	X	L1	L2	L3	L≡[X | L3]
uniao([a,c,e],[b,d],L)	#2	a	[c,e]	[b,d]	*[c,e,b,d]*	*[a,c,e,b,d]*
uniao([c,e],[b,d],L)	#2	c	[e]	[b,d]	*[e,b,d]*	↖ *[c,e,b,d]*
uniao([e],[b,d],L)	#2	e	[]	[b,d]	*[b,d]*	↖ *[e,b,d]*
uniao([],[b,d],L)	#1	–	–	[b,d]	–	↖ *[b,d]*

3. **Dividir uma lista em duas outras listas**: Lista inicial é "em [X,Y | L]", em uma lista

```
#1 divide([], [], []).          % num par de objetos na lista
#2 divide([X], [], [X]).          % num impar de objetos na lista => lista L2
#3 divide([X,Y | L3] , [X | L1], [Y | L2] ):-
                                            divide( L3, L1, L2).
```

Obs: Estes dois últimos predicados apresentam uma particularidade interessante. Permitem que os predicados encontrem a lista original. Exemplo:

```
?- divide([a,b,c,d,e],L1,L2).
L1=[a,c]      L2=[b,d,e]
?- divide(L , [ a , b ], [ c , d ]).
L=[a, c, b, d]
```

Um "*mapa de memória aproximado*" é dado por:

	Regra	X	Y	[X | L1]	[Y | L2]	L3
divide([a,b,c,d,e],L1,L2)	#3	a	b	*[a,c]*	*[c,d,e]*	[c,d,e]
divide([c,d,e],L1,L2)	#3	c	d	*[c]*	*[d,e]*	[e]
divide([e],L1,L2)	#2	e	–	*[]*	*[e]*	–

4. **Imprimir uma lista**: observe o uso do predicado "*put*" ao invés do "*write*". Esta troca se deve a razão que o Prolog trata as listas no código original ASCII, ou seja "fred" = [102,101, 114, 100].

```
escreve_lista( [ ] ).
escreve_lista( [ Head | Tail ] ) :-
                                          write( ' : ' ),
                                          put( Head ),
                                          escreve_lista( Tail ).
```

5. **Verifica se um dado objeto pertence há uma lista**: novamente, em alguns Prolog's, este predicado pode estar embutido, confira.

```
member( H, [ H |  _ ] ).
member( H, [ _  | T ] ) :- member(H, T).
```

6. **Adiciona um objeto em uma lista, caso esse não esteja contido**:

```
add_to_set(X, [ ], [X]).
add_to_set(X, Y, Y) :- member(X, Y).
add_to_set(X, Y, [X | Y]).
```

7. **O maior valor de uma lista**: retornar o maior valor de uma lista.

```
1.    max( [] ,0) :- !.
2.    max( [M] , M ) :- !.
3.    max( [M , K], M ) :- M >= K , !.
4.    max( [M|R] ,N ) :- max( R , K ) , max( [K , M] , N).
```

Neste momento, o aluno deve ter percebido que a ordem com que as regras se encontram dispostas na Memória de Trabalho (MT) deve ser considerada. Pois o mecanismo de "*backtracking*" do Prolog, força uma ordem única de como estas regras serão avaliadas.

Sendo assim, uma disciplina de programação em Prolog se faz necessária algumas vezes! **No caso das listas, as condições de paradas devem ocorrer antes da regra geral recursiva**. A exceção é a leitura de uma lista, este predicado é descrito mais adiante. A regra geral é aquela que contém uma recursividade, no exemplo anterior, é a regra número 4. As demais regras, 1 a 3, são regras destinadas à parada da recursão. Logo, obrigatórias.

Estas regras de parada, também são chamadas de regras ou cláusulas "*aterradas*" ("*grounding*", pois delimitam o final da recursão. Como exercício, determine nos exemplos anteriores quem são as regras gerais e as aterradas.

8. **Inverter uma lista**: este método é ingênuo (primário) na inversão de uma lista, no sentido que faz $n(n+1)/2$ chamadas recursivas a uma lista de comprimento n.

```
naive_reverse( [ ] , [ ] ).
naive_reverse( [ H  | T ], Reversed ) :-
        naive_reverse( T , R ),
        append( R , [ H ], Reversed ).
```

9. **Inversão sofisticada de uma lista**: usa como "*truque*" um acumulador, compare com o anterior.

```
xinvertex(A, Z) :- reverse(A, [ ], Z).
reverse( [ ] , Z, Z).
reverse( [A | X ], Acumulador, Z ) :-
                           reverse( X, [ A | Acumulador ],  Z ).
```

10. **Leitura de uma lista via teclado**: observe que a cláusula aterrada "**quase sempre**" se encontra antes da cláusula geral. Contudo, a leitura de uma lista é uma das raras exceções em que o aterramento vem depois da regra geral recursiva.

```
le_lista( Lista ) :-
                     write('Digite <Enter> ou <Escape> para terminar: '),
                     write('  ===> '),
                     le_aux( Lista ).
le_aux( [Char | Resto ] ) :-
                                            write('  '),
                                            get0(Char),
                                            testa(Char),
                                            put(Char),
                                            put(7),  /* beep */
                                            le_aux( Resto ).
/* Condição da Parada */
      le_aux( [ ] ) :- !.
testa(13) :- !, fail.        /* Return */
testa(10) :- !, fail.        /* New line ===> UNIX */
testa(27) :- !, fail.        /* Escape */
testa( _ )  :- true.
```

5 Operações especiais

1. Removendo uma regra ou fato da Memória de Trabalho (MT):

```
 ?- retract(uniao([A|B], C, [A|D]) :- uniao(B, C, D)).
```

2. Adicionando **uma regra** ou **fato** da MT:

```
?- asertz(uniao(B, [A|C], [A|D]) :- uniao(B, C, D)).
Correct to: 'assertz( (uniao(B, [A|C], [A|D]):-uniao(B, C, D)))'?
yes
B = _G519
A = _G513
C = _G514
D = _G517
Yes
```

Avalie o impacto do que representa incluir ou excluir uma regra durante a execução de um programa!

6 Sites interessantes

- http://www.cbl.leeds.ac.uk/ tamsin/Prologtutorial/
- http://www.sju.edu/ jhodgson/ugai/
- http://www.cee.hw.ac.uk/ alison/ai3notes/
- http://www.sics.se/ps/sicstus.html
- http://dobrev.com/download.html

- http://swi.psy.uva.nl/projects/xpce/SWI-Prolog.html
- http://swi.psy.uva.nl/projects/xpce/download.html
- http://www.amzi.com (tem vários artigos e tutoriais que ilustram o uso e aprendizado do Prolog, respectivamente)
- http://www.arity.com

Finalizando, alguns bons livros sobre Prolog:

- Michael A. Convigton, Donald Nute, André Vellino; *Prolog - Programming in Depth*, Prentice-Hall, 1997;
- Ivan Bratko; *Prolog, Programming for Artificial Intelligence*, 2nd edition (or later if there is one), Addison-Wesley;
- W.F. Clocksin and C.S. Mellish; *Programming in Prolog*, 3rd edition, Springer-Verlag;
- Leon Sterling and Ehud Shapiro; *The Art of Prolog*, MIT Press;
- Richard A. O'Keefe; *The Craft of Prolog*, MIT Press 1990;
- A biblioteca da UDESC recebeu talvez o "*melhor´´* livro de Prolog, não tenho o nome (enviem por mail o nome), e quem emprestar, comunique aos colegas e disponibilize o mesmo.