Lógica Matemática

Fundamentos de Prolog e ECL^iPS^e

(Joinville, 17 de novembro de 2013)

Claudio Cesar de Sá
Rogério Eduardo da Silva
Departamento de Ciência da Computação
Bloco F - 2o. Piso - F-209 - Sala 13
Joinville - SC
email: `claudio@colmeia.udesc.br`

Sumário

Introdução

Histórico

A linguagem Prolog:

✓ As bases na lógica clássica

✓ Início da década de 70 sua primeira implementação

✓ Foi evoluindo, e continua até hoje ativo.

✓ Os fundamentos da LPO (Lógica de Primeira Ordem), seguindo para uma notação em cláusulas (forma clausal), e numa notação restrita em cláusulas de Horn.

Depois tem mais, ...

Proposta destes Slides

- A proposta é tornar este texto acessível a todas as pessoas sem base em lógica, e até mesmo aquelas que nunca tenham utilizado nenhuma outra linguagem de programação.
- Lembrar que: apenas fazendo/praticando o verdadeiro aprendizado ocorre. Escutar e ver, trazem apenas lembranças.
- Assim, ao final das partes, faças os exercícios propostos.
- Estes slides foram feitos há uns 10 anos atrás e abandonados e ainda estão com alguns erros. O abuso (exagero) das aspas é o principal deles.
- Agora retomado devido o ECL^iPS^e e a Programação em Lógica com Restrições – PLR (*CLP – Constraint Logic Programming*) e as pesquisas em torno da PLR.
- A segunda parte deste material, tem o foco na PLR.

- Alguns fontes estão disponíveis em: `http://www2.joinville.udesc.br/~coca/cursos/ic/pgms_em_prolog/`
- Assim, esta atualização ainda está incompleta. *Please, react me with suggestions.*

Requisitos

- Novamente: apenas fazendo/praticando o verdadeiro aprendizado ocorre. Escutar e ver, trazem apenas lembranças.
- Assim, tenha em mãos um dos dois programas instalados.
- Prolog sugerido: `http://www.swi-prolog.org/`, vá na aba de *download*.
- Quanto ao ECL^iPS^e: `http://www.eclipse-clp.org` ou `http://87.230.22.228/`, mais especificamente `http://www.eclipse-clp.org/Distribution/`, escolha uma versão e a plataforma (linux, sun, mac, ..., windows) que lhe convier.

Index of Distribution

```
Index of /Distribution/6.0_96
              22-Jul-2009 07:26

Icon  Name
[DIR] RPMS/
[   ] This is ECLiPSe 6.0#96
[DIR] common/                    ===> aqui tem os manuais ...
[DIR] i386_linux/
[DIR] i386_nt/
[DIR] sparc_sunos5/
[DIR] src/
```

Slides de: 17 de novembro de 2013

Instalando o ECL^iPS^e no Linux

Seguindo os passos abaixo numa console Linux:

```
Faça os downloads:
.......

mkdir clp_eclipse
mv eclipse_misc.tgz  clp_eclipse/
mv eclipse_basFaça os downloads:
.......

mkdir clp_eclipse
mv eclipse_misc.tgz  clp_eclipse/
mv eclipse_basic.tgz clp_eclipse/
mv eclipse_doc.tgz clp_eclipse/
mv tcltk.tgz clp_eclipse/
tar -xvf tcltk.tgz
```

```
tar -xvf eclipse_basic.tgz
tar -xvf eclipse_misc.tgz
 tar -xvf eclipse_doc.tgz
mv *.tgz /home/coca/Downloads/
coca@russel:~/Downloads/clp_eclipse$ cd ..
```

Agora como root:

```
su
Senha:
root@russel:/home/coca/Downloads#
mv clp_eclipse/ /usr/share/
cd /usr/share/
cd clp_eclipse/
```

Saiba apenas onde está a interface TCL, por exemplo:
/usr/share/clp_eclipse/tcltk/i386_linux/lib

```
./RUNME
```

se quiser java ... vj o caminnho de onde está o seu Java

Crie os links simbolicos em:

```
cd /usr/bin/
ln -s /usr/share/clp_eclipse/bin/i386_linux/tkeclipse tkeclipse
ln -s /usr/share/clp_eclipse/bin/i386_linux/eclipse eclipse
```

Teste

```
tkeclipse
```

```
ic.tgz clp_eclipse/
mv eclipse_doc.tgz clp_eclipse/
mv tcltk.tgz clp_eclipse/
tar -xvf tcltk.tgz
tar -xvf eclipse_basic.tgz
tar -xvf eclipse_misc.tgz
 tar -xvf eclipse_doc.tgz
mv *.tgz /home/coca/Downloads/
coca@russel:~/Downloads/clp_eclipse$ cd ..
```

Agora como root:

```
su
Senha:
root@russel:/home/coca/Downloads#
mv clp_eclipse/ /usr/share/
cd /usr/share/
cd clp_eclipse/

Saiba apenas onde está a interface TCL, por exemplo:
/usr/share/clp_eclipse/tcltk/i386_linux/lib

./RUNME
se quiser java ... vj o caminnho de onde está o seu Java

Crie os links simbolicos em:
cd /usr/bin/
ln -s /usr/share/clp_eclipse/bin/i386_linux/tkeclipse tkeclipse
ln -s /usr/share/clp_eclipse/bin/i386_linux/eclipse eclipse

Teste
tkeclipse
```

Implementações e Ambientes

- ✓ Há vários Prolog disponíveis na WEB, um dos mais utilizados é o SWI-Prolog http://www.swi-prolog.org/,
- ✓ Multiplataforma: Windows (95, 98 ou NT), Linux ("*Unix-like*"), Mac.
- ✓ Tem-se Prolog bem-pagos e gratuitos;
- ✓ Interfaces gráficas ou texto;
- ✓ E o ECL^iPS^e, objeto de investigação da segunda parte deste curso.

Características

✓ Um programa em Prolog rompe com o conceito de sequencialidade e fluxo de programas, a exemplo de outras linguagens tradicionais de programação.

Resolvendo um Problema

- ✓ Esse exemplo é instigante e contém *todos* conceitos iniciais do Prolog, acompanhe com atenção a discussão em sala de aula.
- ✓ A proposta é resolver de imediato um problema interessante, antes de entrar nos detalhes da linguagem.

Fui há uma festa e apresentado há três casais. Os maridos tinham profissões e esposas distintas. Após alguns goles *me confundi quem era casado com quem, e as profissões. Apenas lembro de alguns fatos, então me ajude descobrir quem são estes casais, com base nos seguintes dados*:

1. O médico é casado com a Maria;
2. O Paulo é advogado;
3. Patrícia não é casada com Paulo;
4. Carlos não é médico.

(Retirado da revista Coquetel: *Problemas de Lógica*)

Modelagem

Os demais nomes de nossos personagens são (sim, estavam faltando mesmo no enunciado):

1. `H = { carlos (c) , luiz (l) , paulo (p) }`
2. `M = { maria, lucia, patricia }`
3. `P = { advogado (a), medico (m), engenheiro (e) }`

Em resumo, um tupla-3 do tipo `(H,M,P)`

O problema se resume em encontrar: `((H1,M1,P1), (H2,M2,P2), (H3,M3,P3))`

Codificando a Descrição Acima

```
/*
Os demais nomes de nosso personagens sao:
H = { carlos (c) , luiz (l) , paulo (p) }
M = { maria, lucia, patricia }
P = { advogado (a), medico (m), engenheiro (e) }

```

```
Modelagem: (H,M,P) logo ((H1,M1,P1), (H2,M2,P2), (H3,M3,P3))
*/

/* aqui comeca o codigo */

prof(advogado).
prof(medico).
prof(engenheiro).
esposa(patricia).
esposa(lucia).
esposa(maria).

x :- deduz(X, Y, Z), write(X), nl, write(Y), nl, write(Z), nl.
deduz((carlos,M1,P1), (luis,M2,P2), (paulo,M3,advogado)) :-

 esposa(M1),
 esposa(M2),
 esposa(M3),
 prof(P1),
 prof(P2),
 M3 \== patricia,
 P1 \== medico,
 P1 \== advogado,
 P2 \== advogado,
 P1 \== P2,
 M1 \== M2,
 M1 \== M3,
 M2 \== M3.
```

```
/* A restricao a) nao foi atendida:
 %% O medico eh casado com a Maria
  logo o programa acima deve sofrer uma modificacao:
  basta reler as hipoteses e voce vai concluir  que o
  programa se reduz a:
  */

y :- deduz2(X, Y, Z), write(X), nl, write(Y), nl, write(Z), nl.

deduz2((carlos,M1,P1), (luis,maria,medico), (paulo,M3,advogado)) :-
  esposa(M1),
  esposa(M3),
  prof(P1),
  M3 \== patricia,
  P1 \== medico,
  P1 \== advogado.


/*?- ['casais.pl'].
% casais.pl compiled 0.00 sec, 0 bytes

Yes
?- x.
carlos, patricia, engenheiro
luis, lucia, medico
paulo, maria, advogado
```

```
Yes
?- y.
carlos, patricia, engenheiro
luis, maria, medico
paulo, lucia, advogado

Yes
?-
*/
```

Listing 1: 3 casais e profissões dos maridos

Ao longo deste texto, este exemplo será detalhado, bem como o estudante vai estar apto em resolvê-lo de outras maneiras.

Exercícios

1. Execute este programa no Tkeclipse explorando `Tracer`. Utilize a opção `Creep` para avançar passo-a-passo.
2. Assegure o entendimento do `Tracer` pois vai favorecer a sua auto-aprendizagem.
3. No SWI-Prolog, Execute este programa ativando o *trace* gráfico, **guitracer**; e analise porque alguns resultados foram duplicados. Ou seja:

```
?- guitracer.     /* ativa o trace gráfico */
....
```

4. Faça as seguintes experimentações na console do interpretador:
   (a) ?- prof(X).
   (b) ?- esposa(Y).
   (c) ?- prof(X) , X $\doteq$ medico.
5. Altere e/ou inclua algumas regras, afim de restringir a quantidade de respostas.

Avance ao exercício seguinte, caso tenhas entendido a Máquina Prolog, deduzir tais respostas. Caso tenhas dúvida, habilite o trace (?- trace, ...) no prefixo das questões acima.

Explicando o Fluxo da Máquina Prologuiana

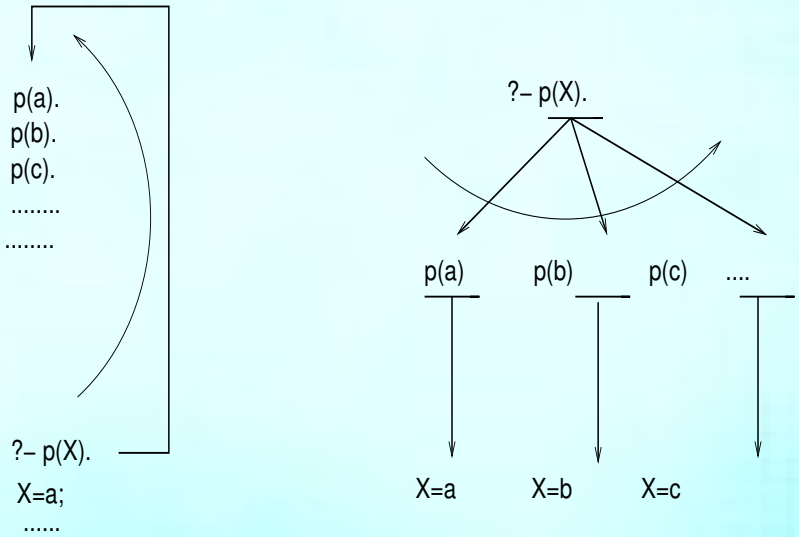
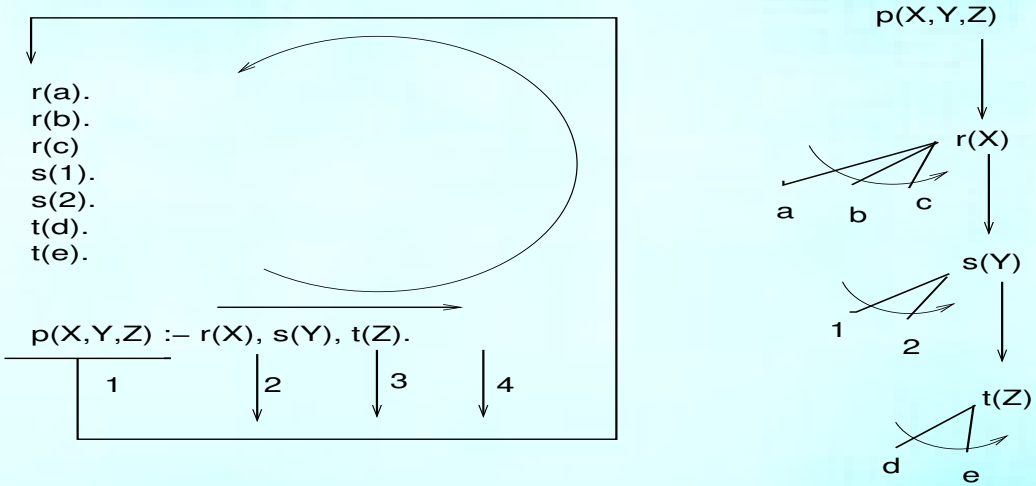


Figura 1: Fluxo básico de um predicado em Prolog

Um Predicado Composto




```
?– p(X,Y,Z).
X=a
Y=1
Z=d
```

Figura 2: Fluxo básico de um predicado composto em Prolog

Implementando estas Figuras

```
r(a).
r(b).
r(c).
s(1).
s(2).
t(d).
t(e).
p(X,Y,Z) :- r(X), s(Y), t(Z).
saida :- p(X, Y, Z),
         format('X:␣~w␣\t␣Y:␣~w␣\t␣Z:␣~w', [X, Y, Z]) .
/*
?- saida.
X: a     Y: 1     Z: d
true ;
X: a     Y: 1     Z: e
.......................
X: c     Y: 2     Z: d
true ;
X: c     Y: 2     Z: e
true.
*/
```

Listing 2: Semântica operacional do Prolog

Observações:

- Estes últimos slides devem ser acompanhados com a explicação em sala de aula.
- Nestes dois slides se encontram várias páginas de texto.
- Muita calma nesta hora com as figuras 9 e 10.
- Como exercício explique o que dizem estas figuras, bem como a numeração indicada nas flechas.
- Feito? Agora avance!

Implementando em ECL^iPS^e

O exemplo dos três casais em ECL^iPS^e (ao interessado apenas em Prolog, salte este exemplo):

```
/* :-lib(ic_symbolic). */
:- lib(ic).

prof(1, medico).
prof(3, advogado).
prof(2, engenheiro).
m(1, patricia).
m(2, lucia).
m(3, maria).

x :- deduz((carlos,M1,P1), (luis,M2,P2), (paulo,M3,advogado)),
     m(M1,E1), m(M2,E2), m(M3,E3),
     prof(P1,P1_out) , prof(P2,P2_out),
     writeln(" A saida e dada por : "),
     writeln((carlos,E1,P1_out)),
     writeln((luis,E2,P2_out)),
     writeln((paulo,E3,advogado)).

deduz((carlos,M1,P1), (luis,M2,P2), (paulo,M3,advogado)) :-
```

```
    [M1,M2,M3] :: 1 .. 3,
    [P1,P2] :: 1.. 2,
     P2 #= 1,
     M2 #= 3,
     M3 #\= 1,
     P1 #\= 1,
     P1 #\= P2,
     alldifferent([M1,M2,M3]).
```

Listing 3: 3 casais e profissões dos maridos

Ambiente de Programação: ECL^iPS^e console

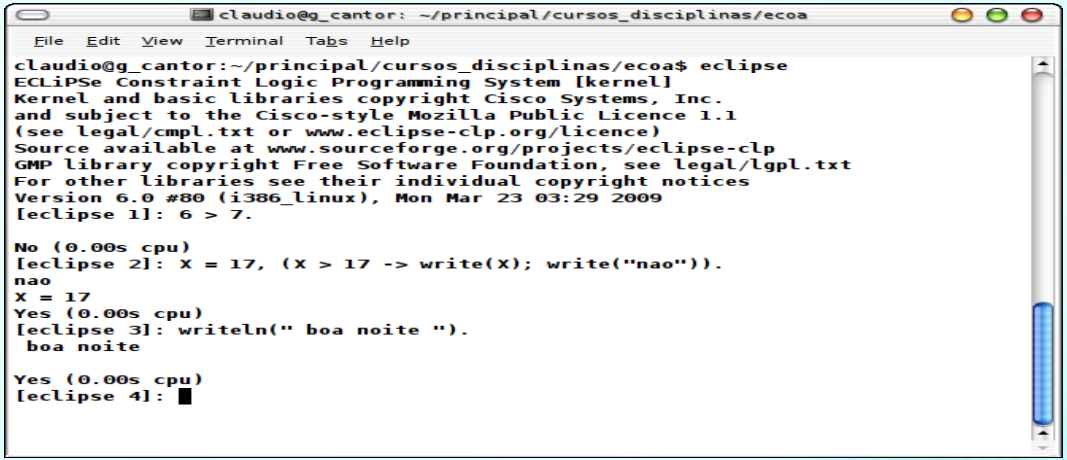


Figura 3: A console básica do ECL^iPS^e (análoga aos Prologs)

Ambiente de Programação: TKECL^iPS^e

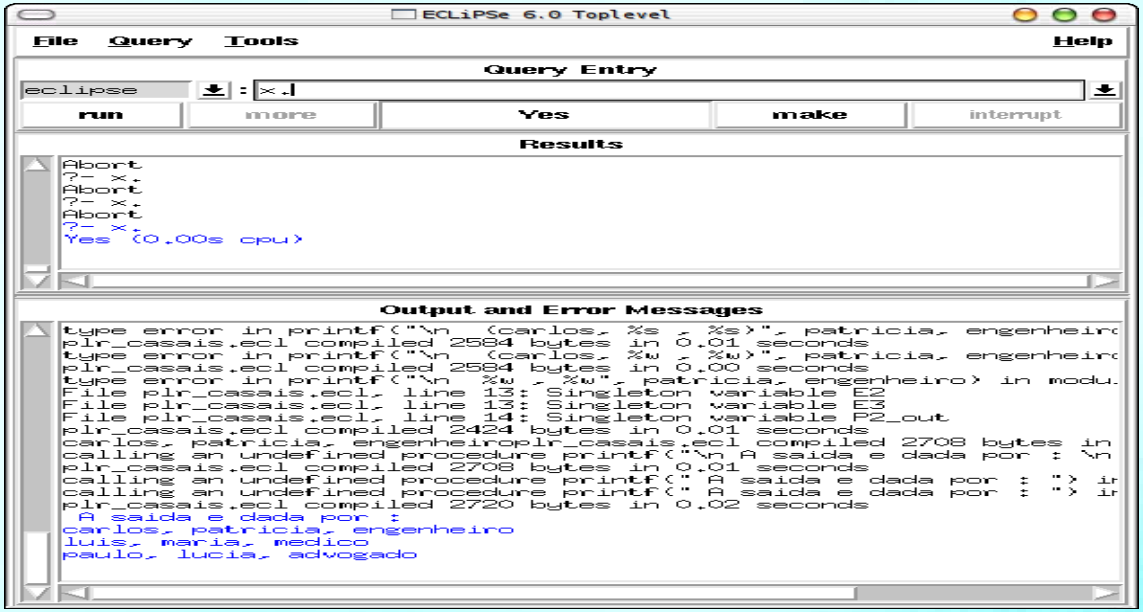


Figura 4: O ECL^iPS^e com a bilioteca TK

Um Outro Exemplo: Os Três Músicos

"Três músicos, João, António e Francisco, tocam harpa, violoncelo e piano. Contudo, não se sabe quem toca o quê. Sabe-se que o António não é o pianista. Mas o pianista ensaia sozinho à Terça. O João ensaia com o Violoncelista às Quintas. Quem toca o quê ?"

O texto acima, muma versão em Prolog é representado por:

```
1
2   instrumento(harpa).
3   instrumento(violoncelo).
4   instrumento(piano).
5   dia(3).
6   dia(5).
7
8   musicos( m(joao,X1,5), m(antonio,X2,D2), m(xico,X3,D3) ) :-
9
10    instrumento(X1),
11    instrumento(X2),
12    instrumento(X3),
```

```
13      (X2 \== piano),
14      (X1 \== X2 , X3 \== X2, X1 \== X3),
15      dia(D3),
16      dia(D2),
17      ((D2 == 3 , D3 == 5);(D2 == 5 , D3 == 3)),
18      (X1 \== violoncelo , X3 == piano , D3 == 3).
```

Fácil, certo? Veja a execução:

```
1   16 ?- consult('c:/publico/musicos.txt').
2   % c:/publico/musicos.txt compiled 0.00 sec, 64 bytes
3     Yes
4   17 ?- musicos(X,Y,Z).
5     X = m(joao, harpa, 5),
6   Y = m(antonio, violoncelo, 5),
7   Z = m(xico, piano, 3) ;
8     No
9   18 ?-
```

Este exemplo será melhorado ..... e como o anterior refeito com recursos da PLR.

Apresentação do Prolog

- Prolog é uma linguagem de programação implementada sobre um paradigma lógico (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, chamado de SLD. Detalhes: *Logic, Programming and Prolog*, de Ulf Nilsson e Jan Maluszyns, editado pela by John Wiley & Sons, 2000.

Características

- Manipula símbolos (objetos) por natureza, logo `7 + 13` tem várias conotações;
- Um átomo, literal ou um objeto é dado por: '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 Delphi, C, Visual Basic, e outras;
- Portabilidade entre plataformas para o Prolog padrão. Logo, Prolog é uma linguagem livre;
- Fácil aprendizado;
- Áreas de aplicações: problemas de IA, provadores de teoremas,

sistemas especialistas, pesquisa operacional, construção de compiladores, etc.

Máquina *Prologuiana*

✓ Uma arquitetura clássica para essa linguagem de programação, encontra-se na figura 15. Sua fundamentação teórica encontra-se no processo de raciocínio típico do *backward chaining*.

✓ A operacionalidade é análogo a esse esquema de raciocínio.

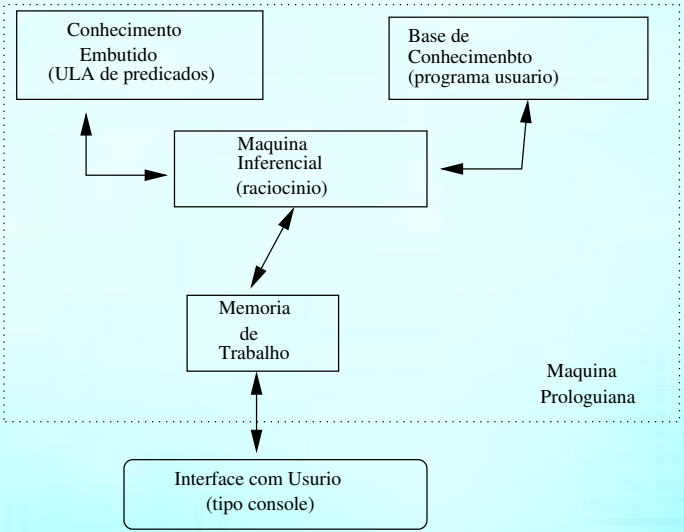


Figura 5: Arquitetura Típica de uma Máquina *Prologuiana*

Exemplos e Como Trabalhar

✓ Além do Prolog ou do Eclipse instalado, é necessário ter um editor de texto padrão ASCII. Aquele de sua preferência.

✓ Qualquer um desde que apresente o número das linhas no texto.

✓ Caso a escolha seja um Prolog com interface gráfica tipo XPCE (http://www.swi-prolog.org/), o qual funciona muito bem sob o Linux, tal editor é acionado pelo comando:

?- edit(file('nome do pgm.pl')).

O ambiente do SWI-Prolog é inicialmente interpretado, e apresenta um *prompt default* ao ser executado: **?-**

✓

Veja os exemplos iniciais:

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

```
No
?- 3 > 2.
Yes
```

Para carregar um programa usuário, previamente editado e salvo no padrão ASCII, para ser usado no SWI-Prolog, então:

```
?- consult('c:/temp/teste.pl'). /* ou */
?- ['c:/temp/teste.pl'].

% teste.pl compiled 0.00 sec, 560 bytes
Yes
?- homem( X ). /* X maiúsculo */
X = joao
Yes
?- homem( x ). /* x minúsculo */
No
```

➽ As letras maiúsculas são as **variáveis** da linguagem Prolog, isto é, aceitam qualquer objeto.

Quanto ao Eclipse

Como todo ambiente com algum requinte de interfaces, exige acertos iniciais para dar produtividade. O Eclipse tem no modo console, mas aconselho o modo com janelas, mais confortável para quem está começando. Então vamos lá:

- ✓ Veja a janela gráfica inicial. Há apenas `File`, `Query`, e `Tools`. Apenas as duas abas iniciais nos interessam de momento.
- ✓ Vá a `File` e acerte suas preferências como Editor, cores, fontes, etc.
- ✓ Em seguida acerte o seu diretório de trabalho. Algo como `meus_programas\`
- ✓ Ou selecione um arquivo novo para digitares, ou carregue um arquivo para execução na opção `Compile File`.
- ✓ Veja na janela de `Output`. Se nenhum erro, estamos indo bem.

✓ Vá para linha de `Query` e teste seus predicados.

```
?- dia(X).  /* digistaste no Query */
X = 3
Yes (0.00s cpu, solution 1, maybe more) /* apertei o more */
X = 5
Yes (0.03s cpu, solution 2)
```

✓ Se compararmos com outras interfaces de desenvolvimento esta é minimalista.

Ambiente Gráfico de Programação: TKECL^iPS^e

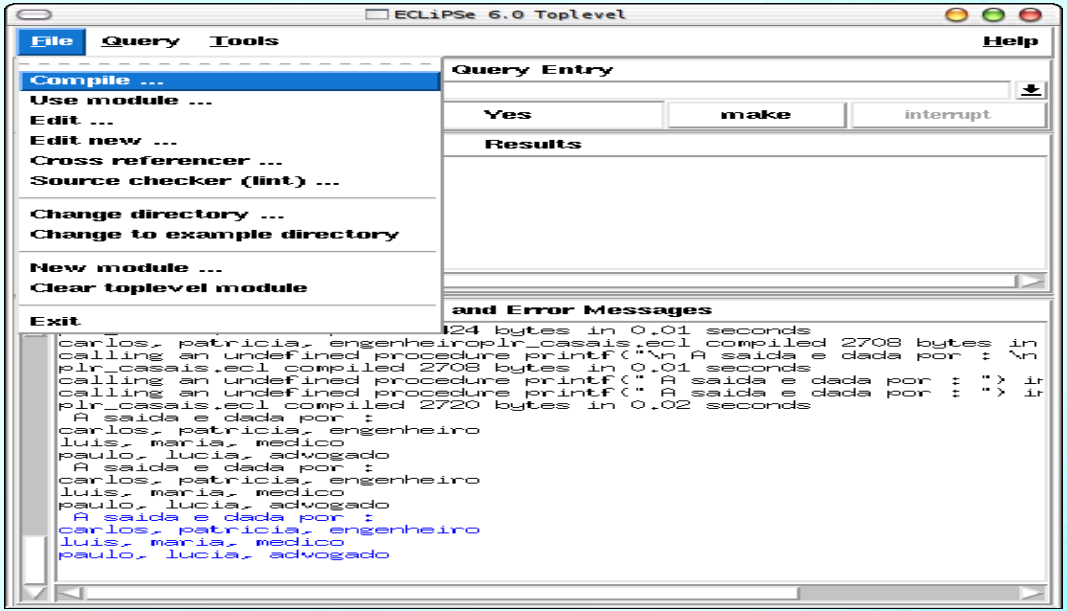


Figura 6: O ECL^iPS^e gráfico – a aba `File`

O *Tracer* para Auto-aprendizagem

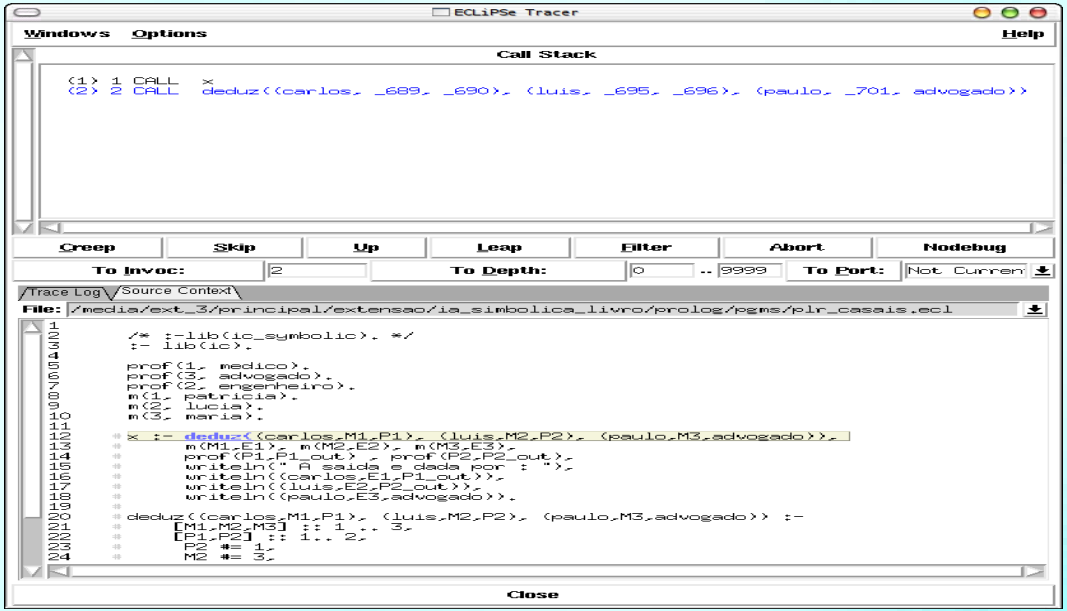


Figura 7: O depurador gráfico do ECL^iPS^e

Sintaxe do Prolog

✓ A sintaxe **Prologuiana** é construída a partir de formulações com os predicados lógicos

✓ o conceito de *predicado* reflete o *espírito* do Prolog.

✓ Inclua um código exemplo aqui....

As construções (*linhas de código em Prolog*) seguem uma sintaxe de três tipos, que podem ser identificada novamente no programa inicial.

1o. tipo: são as **questões** ou *goals* (*query*), isto é, uma pergunta à uma base de conhecimento
...“a idéia a de um teorema a ser demonstrado” .... Seja uma pergunta: *o número 3 é maior que o 2?*

```
?-  >(3,2).    /* ou  ?-  3 > 2.      Ou seja,  3  é maior qu
?-
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. o símbolo  :- é o implica ao contrário  <- */
/*  homem -->  mortal
 leia-se ... para demonstrar que algum X é
 mortal, preciso demonstrar e provar que A é
 um homem  */
Yes
```

Resumindo o Prolog

- A construção de um programa em Prolog, é feita de: *fatos, questões, regras e nada mais!*
- Logo, quanto a parte da léxica e sintática, é construída com poucos elementos.

Detalhando a Sintaxe

➽ Há alguns detalhes de como se montam esses predicados *prologu-ianos*, os mais trabalhosos são os das regras. Essa *Receita de Bolo* segue abaixo:

- Os predicados devem ser em letras minúsculas: *OBRIGATORI-AMENTE*!
- Os fatos, regras e questões terminam por `.` (ponto);
- Seja uma regra, exemplo:

  ```
  irmão(X,Y) :- pai(X,Z), pai(Y,Z), X \== 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 "`:-` " é 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*) dispensa a necessidade de ser instanciada, isto é, sem 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;

Outros Conceitos

➽ Eventualmente vais encontrar os termos abaixo em livros de lógica, de Prolog, do Eclipse, de IC, etc:

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, algumas discutidas neste livro. 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.

Os conceitos acima, fazem parte de uma **tentativa** de descrever o funcionamento aa **Máquina Prologuiana** da figura 19, via um fluxograma, e este é mostrado:

➽ Avalie e verifique a interpretação desse esquema da *pilha abstrata* de questões que o Prolog faz, representado na figura 19. Esta figura efetivamente atende as `queries` do Prolog?

Operações Básicas do SWI-Prolog

Exceto o Visual Prolog (http://www.visual-prolog.com/), 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. Logo, ao leitor sinta-se livre em experimentar outros ambientes do Prolog.

Quanto ao SWI-Prolog, 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 *edit* do DOS, ou outro editor ASCII **com numeração de linhas**;
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.

Alguns comandos para usar no ECL^iPS^e e SWI-Prolog no padrão terminal:

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

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

- Idem ao *eclipse*. Veja o passo-a-passo para instalação no Linux do Claudio.
- Para sair desses ambientes:

```
?- 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 MT:

```
?- 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 do Prolog:

```
?- 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
?-
```

- As imperfeições: *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. Vejamos o 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 */
```

- *Apavoramentos* com o modo *trace* na console:

```
===> <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 !

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, com o uso do predicado interno **fail**:

```
?- 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 "*back-tracking*".

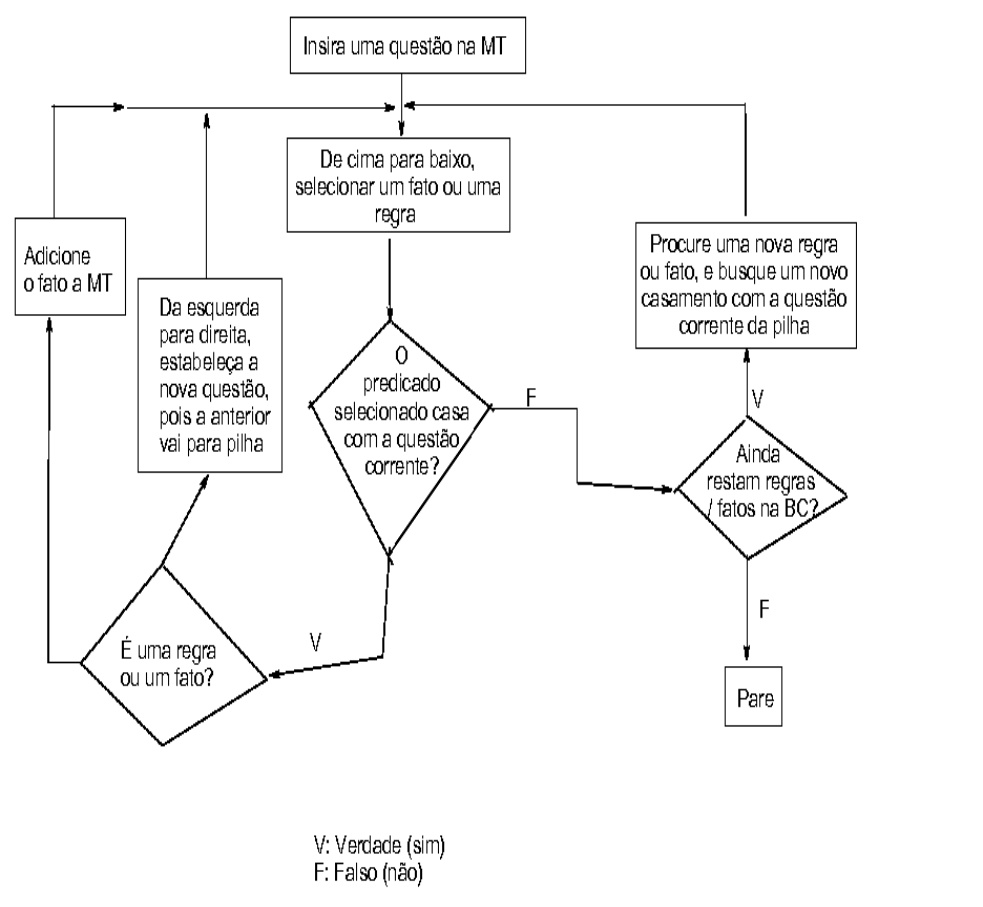


Figura 8: Fluxo *Aproximado* de Inferências em Prolog

Resumindo Via Exemplo

Antes da próxima seção, é necessário compreender o que faz o programa abaixo. Caso tenhas dificuldade, execute-o com auxílio do *tracer*.

```
x(7).
x(5).
x(3).

par(Xpar) :- x(N1), x(N2), N1 =\= N2,
             Xpar is (N1+N2),
             write(N1) , write(' .... '),
             write(N2) , write(' .... '),
             write(Xpar),
             nl,
             fail.

?- par(N).
```

Cuja árvore de busca é dada pela figura 20:

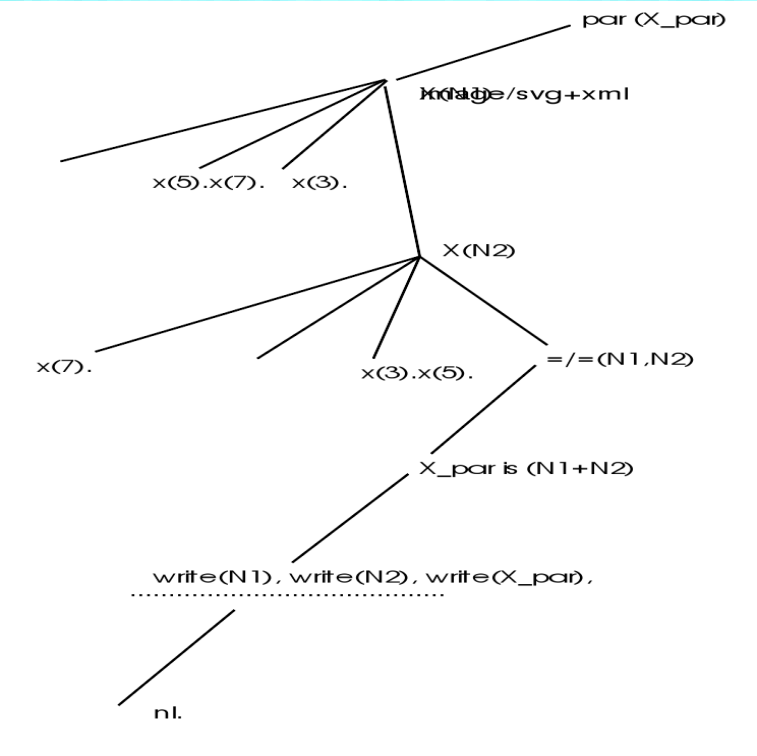


Figura 9: Árvore de busca para regra `par`

➽ Aconselho estudar cada ponto deste exemplo, pois o mesmo sumariza todo conhecimento até o momento sobre Prolog. Avance à

próxima seção, apenas se este exercício for dominado em detalhes.

Exercícios de *Warmup*

1. Construa a árvore geneológica de sua família, tendo as relações (predicados): `filho` e `pai`;
2. A partir do programa anterior, construa as relações: irmão, avô, tio, bisavô, etc. Num próximo assunto, o conceito de ancestral generaliza estes conceitos de pai, avô, bisavô, etc;
3. Quais dos predicados abaixo casam, e se a unificação ocorrer, quais são os resutados? Escreva os seus significados, tomando por base os conceitos de termos ou átomos, *matching* (casamento), e unificação:

```
?- 2 > 3.
?- >(2, 3).
?- 2 == 2.
?- a(1, 2) = a(X, X).
?- a(X, 3) = a(4, Y).
?- a(a(3, X)) = a(Y).
?- 1+2 = 3.
```

```
?- X = 1+2.
?- a(X, Y) = a(1, X).
?- a(X, 2) = a(1, X).
?- 1+2 = 3
?- X + 2 = 3 * Y.
?- X+Y = 1+2.
?- 1+Y = X + 3.
?- pai(X, adao) = pai(abel, Y).
?- X+Y = 1+5, Z = X.
?- X+Y = 1+5, X=Y.
```

4. Seja o programa abaixo:

```
b(1).
b(2).
d(3).
d(4).
c(5).
c(6).
a(W) :- b(X), c(Y), W is (X+Y), fail.
a(W) :- c(X), d(Y), W is (X+Y).
```

Encontre os valores para `a(Y)` e explique como foi o esquema de

busca. Onde e porquê ocorreu *backtraking*?

Semântica Operacional

- Mais um resumo sobre a operacionalidade dos predicados.
- Estas figuras são difíceis de serem encontradas, mas revelam a essência do Prolog

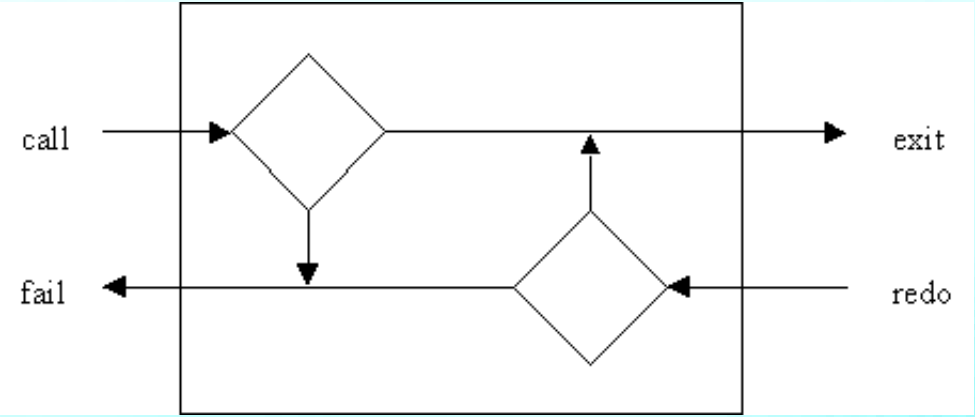


Figura 10: Um fluxo operacional de um predicado típico

➽ Retirado de `http://grack.com/downloads/school/enel553/report/prolog.html`

➽ Ainda da figura 22, temos:

call: é o caminho de entrada do predicado, o qual é chamado em sua primeira vez, para aquela instância.

exit: uma solução encontrada, sua porta de saída para um retorno com `Yes`.

redo: é a porta de entrada para o *backtracking* (caso ocorra uma falha neste predicado, este será exaustivamente inspecionado em todas as suas possibilidades de soluções).

fail: é a porta de saída no caso de não existirem mais nenhuma alternativa a ser pesquisada. Retorno com o `No`.

Predicados de E/S

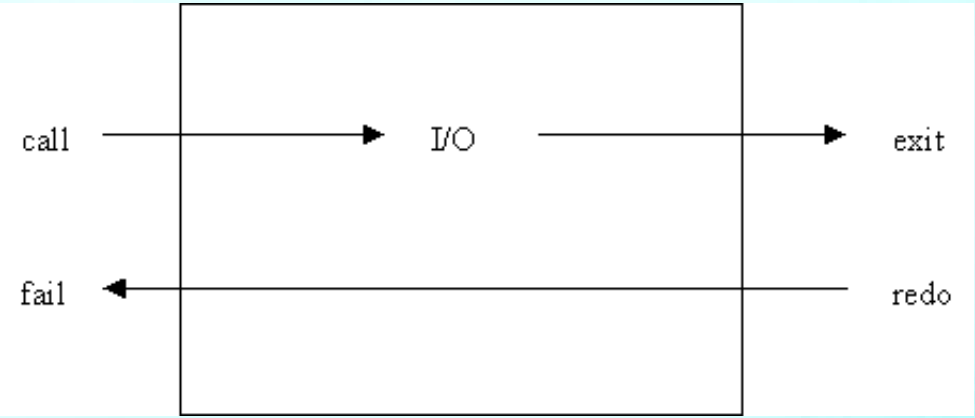


Figura 11: Um fluxo operacional de um predicado de E/S - sem `redo`

Exemplo: o predicado `write/1`

Um predicado que sempre falhe

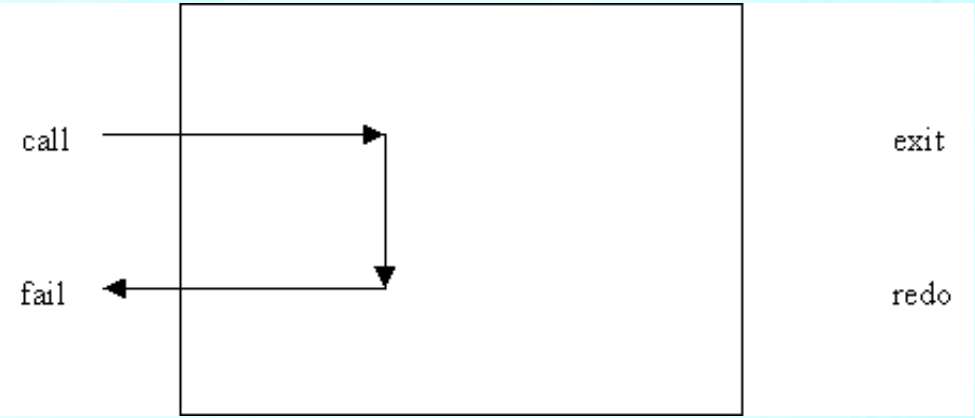


Figura 12: Um fluxo operacional de um predicado com retorno em `fail`

Exemplo: o predicado `fail/0`

Um fluxo completo

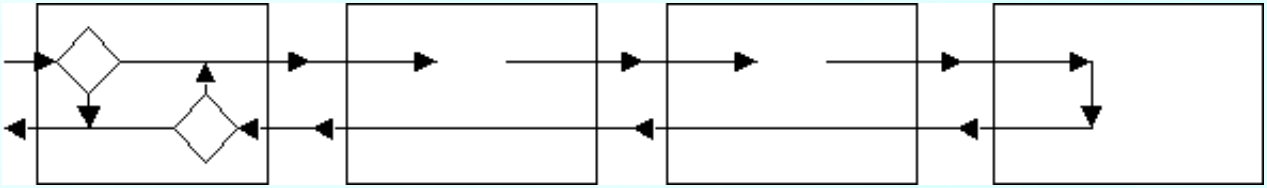


Figura 13: Um fluxo operacional completo

Faça uma reflexão e veja se está tudo claro ....

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. Os exemplos se auto-descrevem.
- ✓ 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.

Algumas ilustrações:

Iteração Clássica

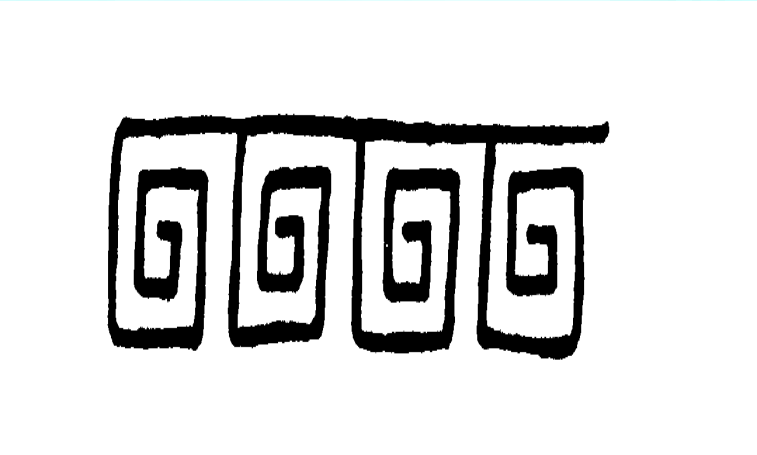


Figura 14: Uma iteração ilustrada

Recursão Ilustrada

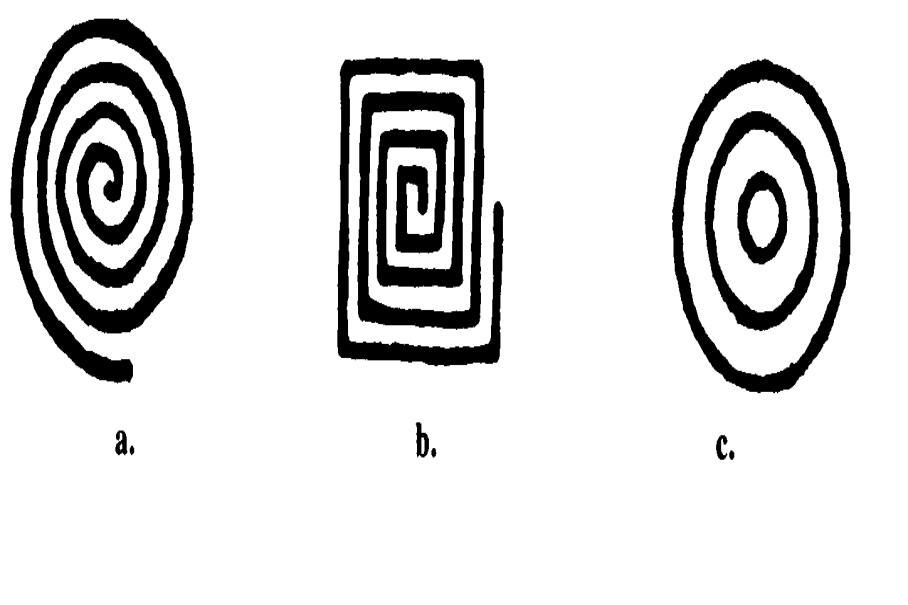


Figura 15: Recursão ilustrada

Recursão Ilustrada 2

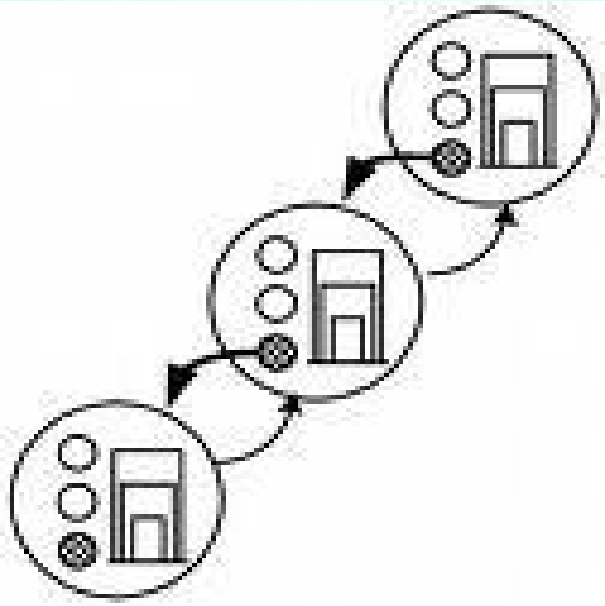


Figura 16: Recursão ilustrada 2

Quase Tudo São Flores

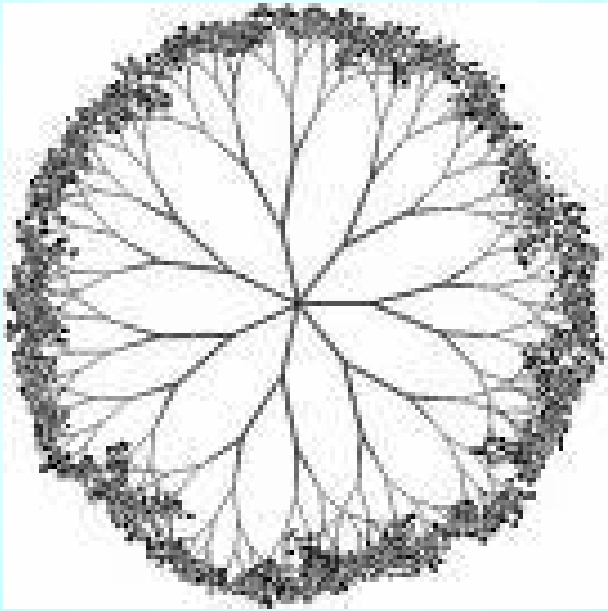


Figura 17: Recursão ilustrada – flores

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:

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

Como o procedimento é recursivo, é necessário encontrar a definição para a "**parada**" da recursividade. Como n não tem limite superior, é para qualquer n, inicia-se pelo que se 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).

Notas:

- 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 alguns casos, a regra recusirva, regra #2, vem antes da regra #1. Alguns desses exemplos são mostrados nestes slides.
- Quando estiveres confiante com o Prolog, estas condições de parada pode vir depois da regra geral. Mas para isto, outros mecanismos de **controle de fluxo de programa** precisam serem apresentados. de tais casos.

O exemplo acima é reescrito em Prolog, é dado por:

```
s(1,1) :- true.          /* REGRA #1 */
s(N, S) :-            /* REGRA #2 */
      N > 1,
      Aux is (N-1),
      format('\n N: ~w \t AUX: ~w \t PARCIAL: ~w \t S: ~w',
              [N, Aux, Parcial, S]),
      s(Aux, Parcial),
      format('\n ==> Apos o casamento da  REGRA #1:'),
      S is (N + Parcial),
      format('\n N: ~w \t AUX: ~w \t PARCIAL: ~w \t S: ~w',
              [N, Aux, Parcial, S]).
```

Listing 4: Soma dos números de 1 a N

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>... →	↖...<3>...
s(1,X)	#1 (aterrada)	1	-	-	↖1

A execução na console:

```
?- s(5,X).

 N: 5     AUX: 4          PARCIAL: _G254          S: _G181
 N: 4     AUX: 3          PARCIAL: _G269          S: _G254
 N: 3     AUX: 2          PARCIAL: _G284          S: _G269
 N: 2     AUX: 1          PARCIAL: _G299          S: _G284
 ==> Apos o casamento da  REGRA #1:
 N: 2     AUX: 1          PARCIAL: 1       S: 3
 ==> Apos o casamento da  REGRA #1:
 N: 3     AUX: 2          PARCIAL: 3       S: 6
 ==> Apos o casamento da  REGRA #1:
 N: 4     AUX: 3          PARCIAL: 6       S: 10
 ==> Apos o casamento da  REGRA #1:
 N: 5     AUX: 4          PARCIAL: 10      S: 15
X = 15
```

Observações:

- O predicado `format` funciona apenas no Prolog;
- Acompanhe as explicações em sala de aula;
- Nem todo conjunto de regras recursivas são passíveis de admitirem tal quadro.

Um Outro Clássico: 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 28, em termos da relação *pai*.

Escrevendo este conceito em Prolog, tem-se:

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

Este conhecimento em LPO é dado por:

- $\forall x \forall y((pai(x,y) \rightarrow ancestral(x,y))$
- $\forall x \forall y \forall z((pai(x,z) \wedge ancestral(z,y) \rightarrow ancestral(x,y))$

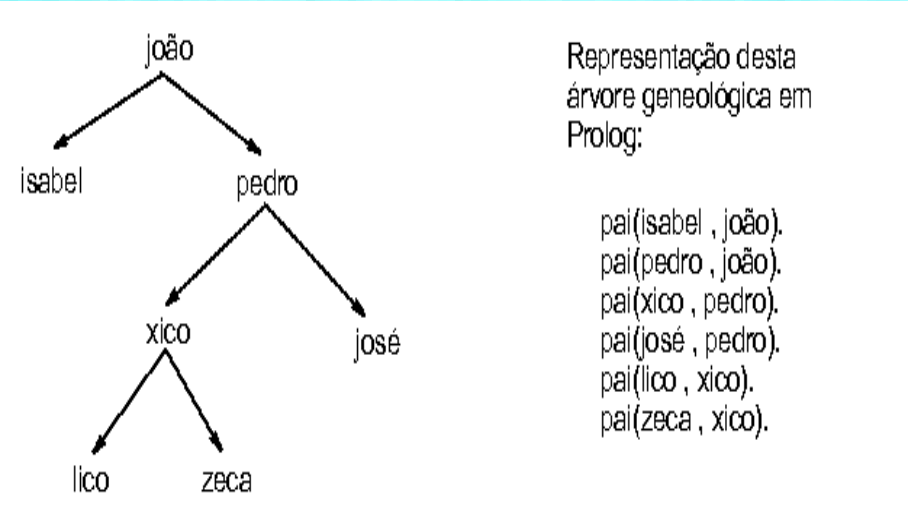


Figura 18: Uma árvore que representa os descendentes de alguém

Este código em Prolog, agora em inglês:

```
parent(john,paul).      /* paul is john's parent      */
parent(paul,tom).       /* tom is paul's parent       */
parent(tom,mary).       /* mary is tom's parent       */
ancestor(X,Y) :- parent(X,Y).
/* If Y is a parent of X, then Y is an ancestor of X */

ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```

```
/* if Y is an ancestor of Z and Z is a parent of X,
   then Y is an ancestor of X */
```

Seu Processamento

Calculando `?- ancestor(john,tom).`

```
CALL ancestor(john,tom).
        CALL parent(john,tom).
        FAIL parent(john,tom).
        CALL parent(john,Z).
            TRY Z=paul
            CALL ancestor(paul,tom).
                CALL parent(paul,tom).
                SUCCEEDS parent(paul,tom).
            SUCCEEDS ancestor(paul,tom).
        SUCCEEDS with Z=paul
    SUCCEEDS ancestor(john,tom).
```

Repita o exemplo com nomes conhecidos de sua família.

Outros Exemplos de Recursividade

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

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

```
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).
```

Clássico “Sócrates” com `fail` :

```
        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 último "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. */

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

Veja a seqüência dos fatos e da saída.

Cálculo do Fatorial : Reformulando sob uma visão matemática, mais especificamente, pela indução finita tem-se:

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

O que é um fato verdadeiro pois:

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

Como o procedimento é recursivo, deve-se encontrar a definição para "**parada**" da recursividade. Como n não tem limite superior, é para qualquer n, então inicia-se pelo que se 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 tem-se:

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

Complete a tabela abaixo, para descrição do fatorial:

Chamada Pendente	**Regra Casada**	**X**	**Aux**	**Parcial**	**Fat**
fatorial(5,X)	#	.....	.....	?..... →	.....
fatorial(4,X)	#	.....	.....	?..... →	↖.....
fatorial(3,X)	#	.....	.....	?..... →	↖.....
fatorial(2,X)	#	.....	.....	?..... →	↖.....
fatorial(1,X)	#	.....	.....	?..... →	↖.....
fatorial(0,X)	#	.....	.....	?..... →	↖.....

"***O caminho do português***" : A proposta do problema é encontrar o custo entre dois vértices quaisquer em um grafo orientado dado pela figura 29. O nome do problema define a proposta do exercício. Este grafo não apresenta ciclos, e nem é bidirecional entre os vértices.

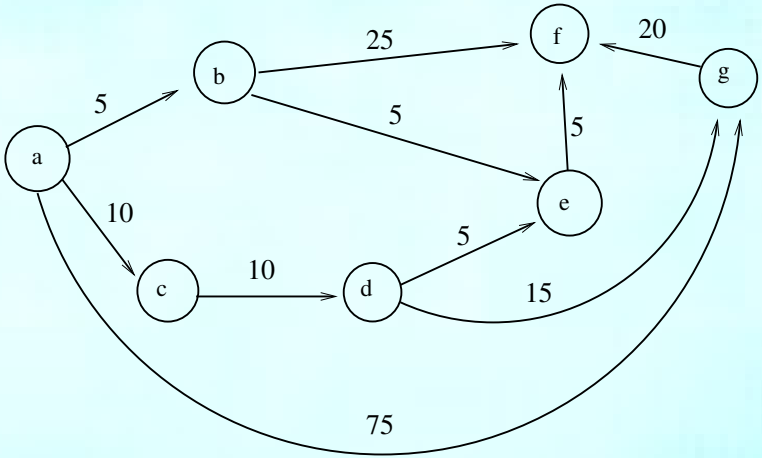


Figura 19: Um grafo que define custos entres as cidades - unidirecional

Para expressar a idéia da rota tem-se que:

- Uma rota entre X e Y é uma estrada que liga X e Y direta-

mente;

- Uma rota entre X e Y é uma estrada entre X e Z e e uma rota entre Z e Y.

```
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
?-
```

Triangulo de Astericos : ir para os exercícios ...

Functores

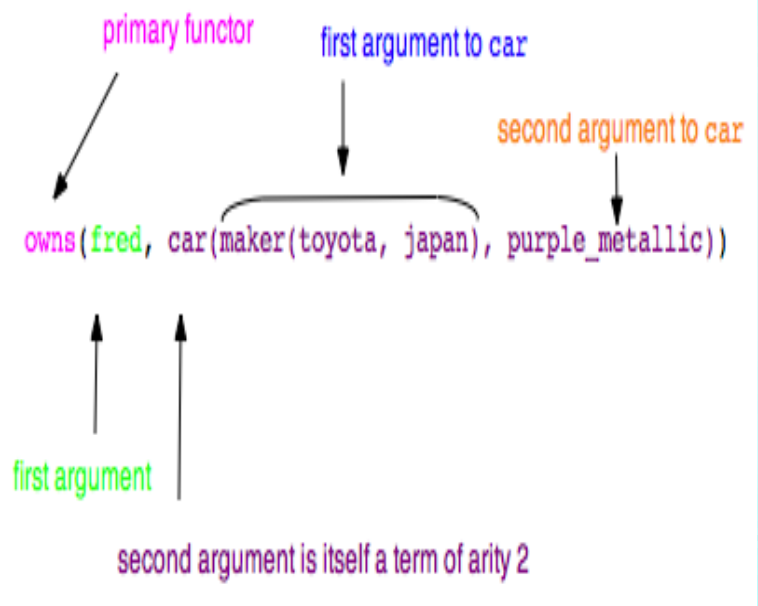


Figura 20: Um exemplo de functor – um termo composto

Definições dos Functores

Functores: são funções lógicas, logo há uma equivalência entre um domínio e imagem.

➽ Os functores permitem contornar o mapeamento típico sobre $\{V, F\}$ (*yes* ou *no*)

➽ Um domínio de objetos mapeados à objetos.

Exemplo: seja a descrição de uma classe de objetos do tipo carro (ver figura 31):

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

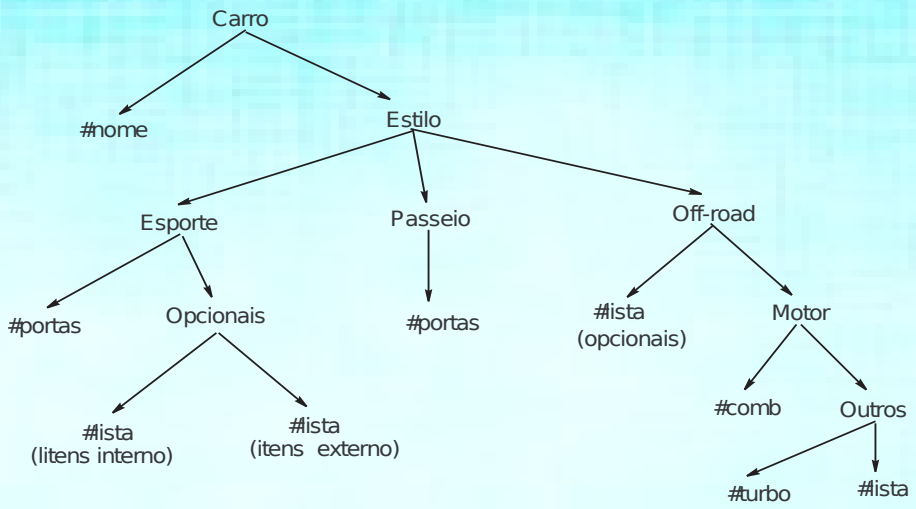


Figura 21: Representando o conceito hierárquico com um functor

Motivação

A principal observação do exemplo acima é a hierarquia "*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 tem-se:

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.

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.)**.

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, deduz-se que functor ``\emph{casa}''
  (``\emph{matching}'')  com variáveis (letras maiúsculas).

/* 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 !

Listas

- Pré-requisito: conceitos de recursividade e functor dominados!
- Seguem conceitos das LPs convencionais
- Essencialmente vamos computar sob uma árvore binária
- Ilustrando esta computação

Resumindo o Fluxo do Cálculo Recursivo

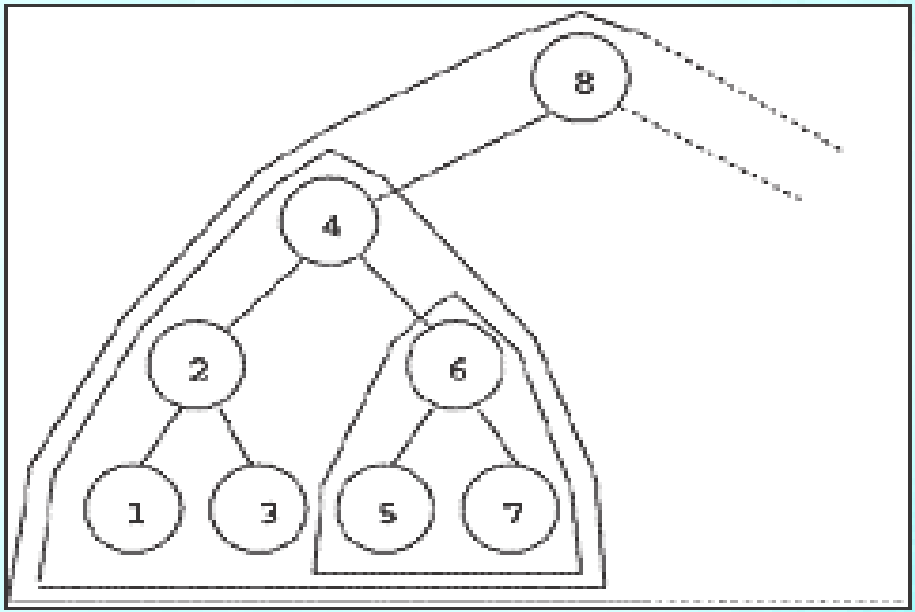


Figura 22: Cálculo Recursivo 1

O Fluxo Operacional das Listas é Análogo

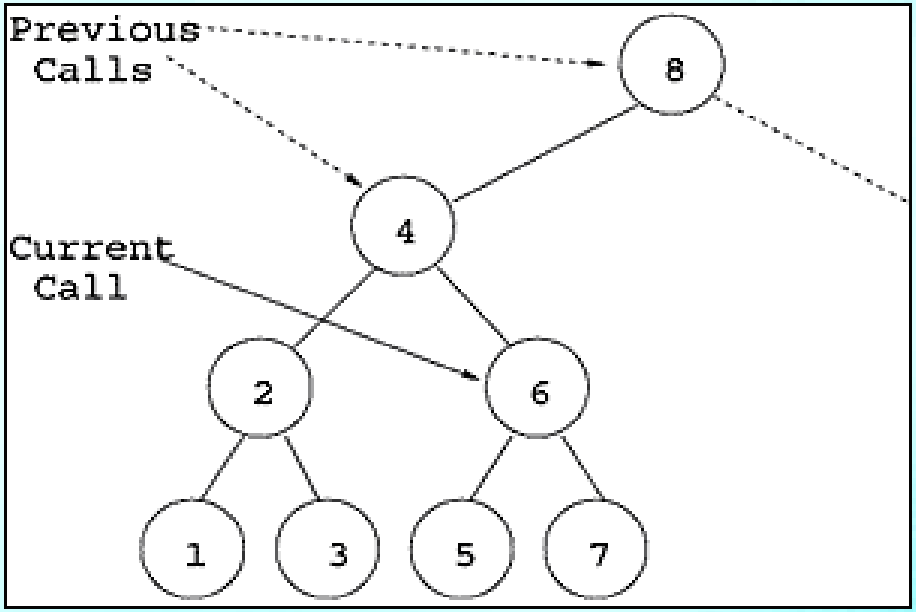


Figura 23: Cálculo Recursivo 2

Uma Lista Genérica

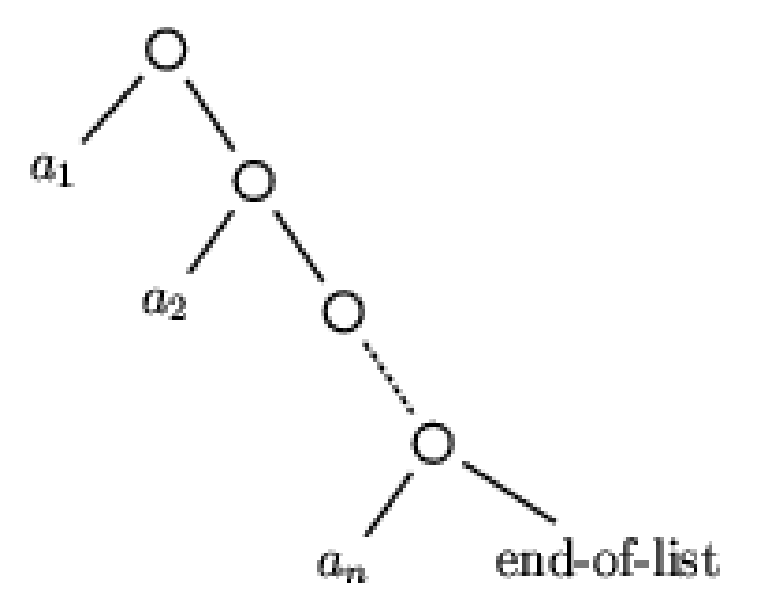


Figura 24: Uma lista em Prolog

A Sintaxe das Listas

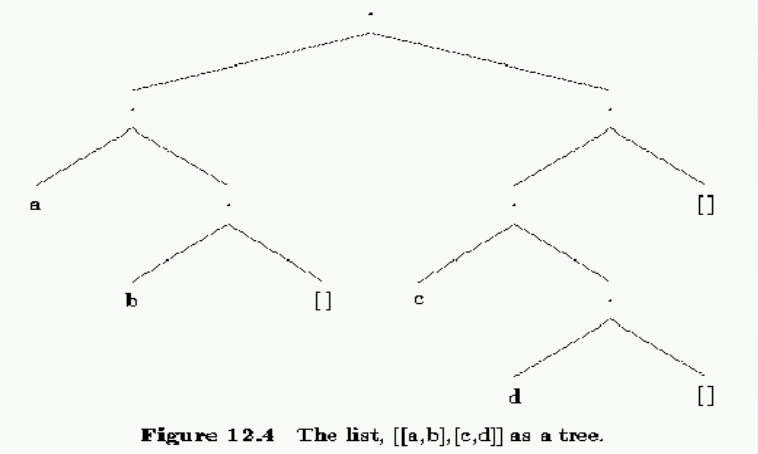


Figura 25: Lista1 ilustrada

- Após as definições e exemplos volte a esta figura

Definições sobre Listas

Definições iniciais: e recursivas ar

- 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] (veja esta nota de roda-pé), pois apresenta uma hierarquia interna.

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

[1] **Logo, a lista apresenta uma hierarquia natural, internamente.**

Exemplos: [a, b, c, d], logo um predicado cujo argumento seja algumas letras, tem-se 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 38, também é uma lista.

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.

Listas: Uma Auto-Definição

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, tem-se 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. */

?- 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.

Exemplos de Listas

As regras sobre listas são diversas e elegantes. Apenas exercitando é que se cria a destreza necessária para resolver qualquer desafio em Prolog. Alguns clássicos são mostrados nos exemplos que se seguem. Há alguns que são combinados com outros criando alguns bem complexos.

Comprimento de uma lista: O comprimento de uma lista é o comprimento de sua sub-lista, mais um, sendo que o comprimento de uma lista vazia é zero. Em Prolog isto é dado por:

```
#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 →	3+1=4
compto([b,c,d],N)	#2	b	[c,d]	2 →	↖ 2+1
compto([c,d],N)	#2	c	[d]	1 →	↖ 1+1
compto([d],N)	#2	d	[]	0 →	↖ 0+1
compto([],N)	#1	–	–	–	↖ 0

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
```

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]*

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

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

```
#1  divide([], [], []).  % N par de objetos na lista
#2  divide([X], [], [X]). % N impar da 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]*	*[b, 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]*	–

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 ).
```

Como uso simplificado tem-se:

Ao final de qualquer exemplo que use listas, temos que imprimir....
assim, há muitos exemplos

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).
```

O interessante é observar a versatilidade dos predicados. Explorando este tem-se:

```
?- member(3, [4,5,3]).

Yes
?- member(X, [4,5,3]).
X = 4 ;
X = 5 ;
X = 3 ;
No
?- member(3, X).
X = [3|_G231]
Yes
?- member(3, X).
```

```
X = [3|_G231] ;
X = [_G230, 3|_G234] ;
X = [_G230, _G233, 3|_G237]
.........................
```

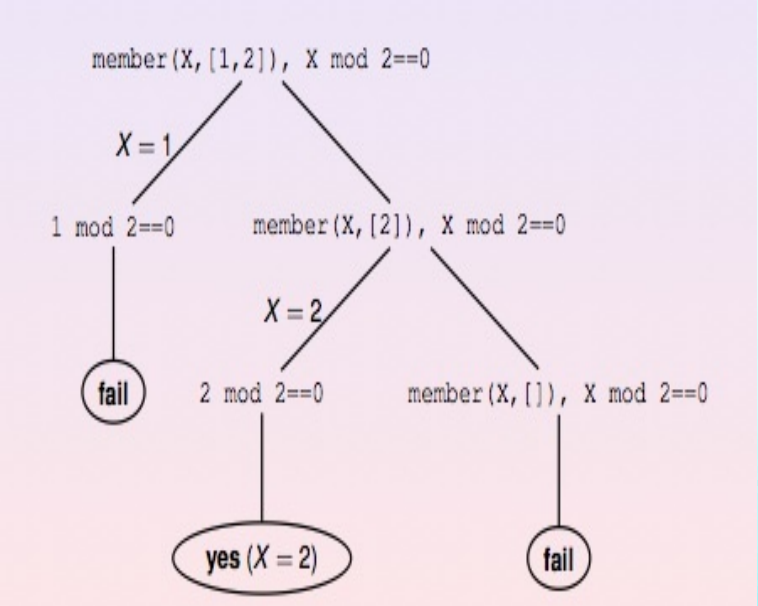


Figura 26: Um exemplo com member

Reflita sobre a variedade de usos deste predicado

Adiciona um objeto em uma lista: Neste exemplo, um objeto é adicionado a lista sem repetição caso este já esteja contido na lista:

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

O maior valor de uma lista: Retorna o maior valor numérico 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).
```

➠ Uma perigosa e difícil recursão dupla no predicado acima. Veja o exemplo do `menor` a seguir, e refaça o `max`.

O menor valor de uma lista: Retorna o menor valor numérico de uma lista.

```
menor(_,[]) :- write(' sua lista estah vazia ').
menor(A,[A]).
menor(A,[A,B]):- A =< B.
menor(B,[A,B]):- B =< A.
menor(X, [A , B | C] ) :- A < B, menor(X, [A|C]).
menor(X, [A , B | C] ) :- B =< A, menor(X, [B|C]).
```

- Neste momento, o estudante 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 são 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.

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 ).
```

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).
```

Verifica se uma lista está contida em outra lista: usa uma técnica simples de ir comparando sequencialmente. Caso ocorra um erro, a substring procurada é restaurada por meio de uma cópia, presente no terceiro argumento.

```
subs(A,B) :- sub(A,B,A).
/* A lista A está contida em B ? */
sub([],_,_).
sub([A|B] , [C|D] , Lcopia) :- A == C,
                               sub( B, D, Lcopia).
sub([A|_] , [C|D] , Lcopia) :- A \== C,
                               sub(Lcopia,D, Lcopia).
```

Como exercício, faça este predicado utilizando dois `append`.

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.
```

Há outros casos com o aterramento depois da regra geral.

Removendo um item da lista: Exlcui todas ocorrências de um termo na lista. Junto com o `união` (*append*) este predicado tem várias utilidades. Observe os exemplos:

```
del_X_all(X, [],[]).
del_X_all(X, [X|L], L1) :- del_X_all(X,L,L1).
del_X_all(X, [Y|L1], [Y|L2]) :- del_X_all(X,L1,L2).


?- del_X_all(3, [3,4,5,3,3,7,3],X).

X = [4, 5, 7]

Yes
?- del_X_all(8, [3,4,5,3,3,7,3],X).

X = [3, 4, 5, 3, 3, 7, 3]

Yes
?- del_X_all(3, [3],X).

X = []

Yes
?- del_X_all(3, [],X).

```

```
X = []

Yes
?- del_X_all(X, [3,4],Y).

X = 3
Y = [4] ;

X = 4
Y = [3] ;

X = _G189
Y = [3, 4] ;

No
?-
?- del_X_all(X, [3,4],[3]).

X = 4

Yes
```

Observe que neste último exemplo o predicado `del_X_all` deduziu o valor do termo `X` excluído no predicado. Ou seja, este é um dos muitos predicados que apresentam uma multi-funcionalidade.

Permutação: Alguns predicados são difíceis em qualquer linguagem de programação. Um destes é a permutação a qual é útil vários problemas. O predicado `exlclui_1a` excluia a primeira ocorrência de um termo na lista, enquanto o `del_X_all`, visto anteriormente, exclui todas ocorrências.

```
1
2  /* Permutacao de elementos */
3  permutar([], []). /* Condicao de parada*/
4  permutar([X|L], Lpermutada):-
5          permutar(L, L1),
6          exclui_1a(X, Lpermutada, L1).
7
8  /* Exclui  X apenas em sua primeira ocorrencia */
9  exclui_1a(X, [X|L], L).
10 exclui_1a(X, [Y|L], [Y|L1]):- exclui_1a(X, L, L1).
11
12 /* executando */
13 init :- permutar([5,7,9],X), nl , write(X), fail.
14
15 ?- init.
16
17 [5, 7, 9]
18 [7, 5, 9]
19 [7, 9, 5]
```

```
[5, 9, 7]
[9, 5, 7]
[9, 7, 5]

No
?-
```

Partição:

Problemas de Buscas em IA

- Faça muitos exercícios sobre listas e functores
- Combinando os functores as listas, qual é a sua utilidade?
- Resgate o exemplo da Cruz ... xx aulas atrás
- Os problemas de buscas em IA, basicamente se utilizam do núcleo descrito abaixo, ou uma variação sobre o mesmo. Acompanhe a discussão com o professor, e veja o link `http://www.csupomona.edu/~jrfisher/www/prolog_tutorial/`. O núcleo abaixo retirei deste sítio.
- Retrata exatamente/precisamente os problemas de buscas em geral.

Núcleo Mágico

➽ Reflita sobre este código, há muito conhecimento embutido.

```
solve(P) :-
        start(Start),
        search(Start,[Start],Q),
        reverse(Q,P).

search(S,P,P) :- goal(S), !.                /* done                    */
search(S,Visited,P) :-
       next_state(S,Nxt),                   /* generate next state     */
       safe_state(Nxt),                     /* check safety            */
       no_loop(Nxt,Visited),                /* check for loop          */
       search(Nxt,[Nxt|Visited],P).         /* continue searching...   */

no_loop(Nxt,Visited) :-
        \+member(Nxt,Visited).
```

Continuando com o *Núcleo Mágico*

```
next_state(S,Nxt) :-  < fill in here >.
safe_state(Nxt) :- < fill in here >.
no_loop(Nxt,Visited) :- < fill in here >.
                        /* if different from default clause */
start(...).
goal(...).
```

- Logo, voce tem um código *quase que padrão* para resolver qualquer problema de buscas!
- Basicamente tudo que fiz que problemas em IA envolve esta estrutura canônina de código *prologuiano*

Resumindo esta Idéia em uma Figura

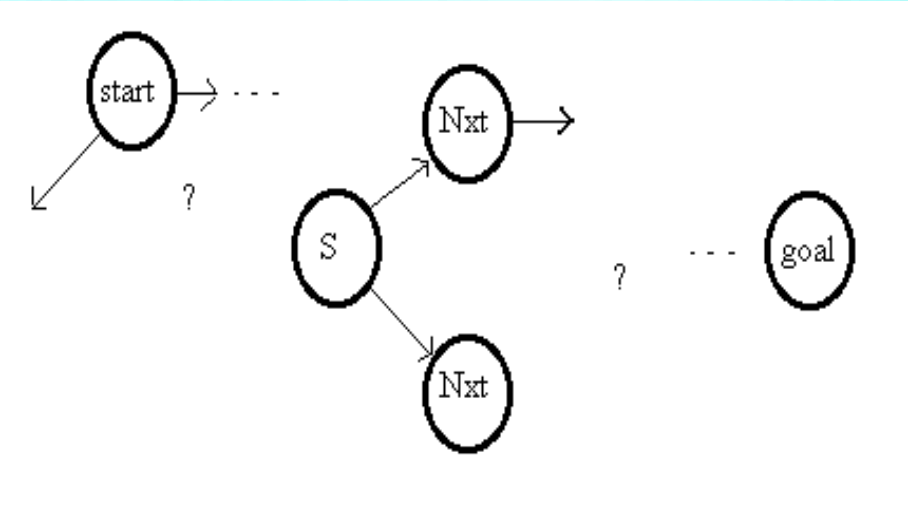


Figura 27: Nós iniciais e finais do problema

➽ Volte aos códigos e reflita mais uma vez!

Reusando o Conhecimento de Listas e Functores

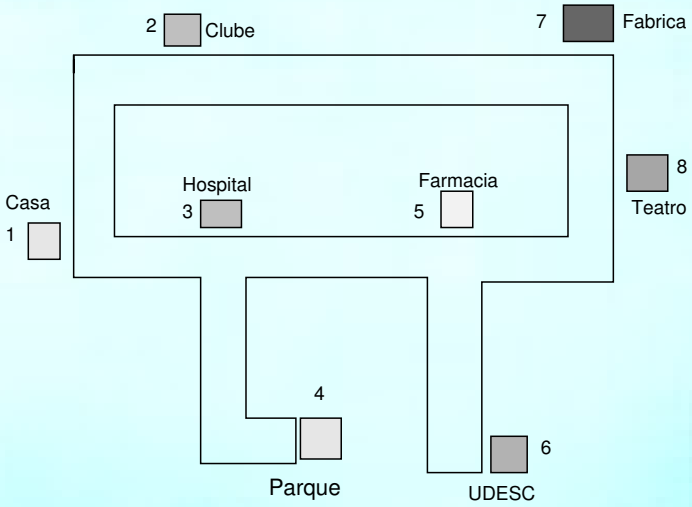


Figura 28: Um mapa – grafo clássico

Resolvendo com Busca em Profundidade

➽ Acompanhe as explicações dos códigos em sala de aula!

```
/*
BUSCA em PROFUNDIDADE - MAPA da CIDADE
*/

inicio(Sol_invertida) :-
    no_origem(No_inicial),
    busca(No_inicial, [No_inicial], Solucao),
    reverse(Solucao, Sol_invertida),
    write(Sol_invertida),
    no_destino(Y), nl,
    write('Onde o no de origem do mapa era: '), write(No_inicial), nl,
    write('E o no de destino do mapa era: '), write(Y).

inicio('Nao hah mais solucoes'):- !.
        /* para terminar com yes */
/*********************************************/

/* Nós iniciais e finais do problema */
no_origem(casa).
no_destino(teatro).
```

```
/*****************************************
 * Rrepresentação de um MAPA QUALQUER     *
 *****************************************/
/* MODELAGEM do Problema ===> parte mais dificil */
a(casa, hospital).
a(casa, clube).
a(hospital, parque).
a(hospital, farmacia ).
a(farmacia, udesc).
a(teatro, farmacia ).
a(fabrica, clube).
a(fabrica, teatro).

/* para não ter que reescrever que:
  a(fabrica, teatro) \== a(teatro,fabrica).
*/

conexao(X,Y) :- a(X,Y).
conexao(X,Y) :- a(Y,X).

/*********************************************************/

%% cond. de parada.... o nó corrente é o nó desejado = nó final
busca( No_final, [ No_final | Caminho] , [No_final|Caminho]):-
            no_destino(No_final).
%% termina a busca...
```

```
busca(Noh,  [Noh|Caminho], Solucao):-
      conexao(Noh , Novo_nodo), /* proximo no = estado */
      not( member(Novo_nodo,[Noh|Caminho])),
      /* verifica se não foi visitado */
      busca( Novo_nodo, [Novo_nodo , Noh | Caminho ] , Solucao).

/*****************************************************/
/*
?- inicio(X).
[casa, hospital, farmacia, teatro]
Onde o no de origem do mapa era: casa
E o no de destino do mapa era: teatro

X = [casa, hospital, farmacia, teatro] ;
[casa, clube, fabrica, teatro]
Onde o no de origem do mapa era: casa
E o no de destino do mapa era: teatro

X = [casa, clube, fabrica, teatro] ;

X = 'Nao hah mais solucoes' ;

No
?-
*/
```

Listing 5: Grafo da Cidade

Resolvendo com Busca em Largura

➽ Acompanhe as explicações dos códigos em sala de aula!

```
/*
BUSCA EM LARGURA - PROBLEMA DO MAPA
*/
/*****************************************
 * Rrepresentação de um MAPA/GRAFO QUALQUER      *
 *****************************************/
/* MODELAGEM do Problema ===> parte mais dificil */
a(casa, hospital).
a(casa, clube).
a(hospital, parque).
a(hospital, farmacia ).
a(farmacia, udesc).
a(teatro, farmacia ).
a(fabrica, clube).
a(fabrica, teatro).

/*
                    casa
              /              \
          hospital           clube
```

```
21              /     \                    \
22     parque    farmacia            fabrica
23                /   \                    \
24          udesc   teatro               teatro
25
26 */
27
28 /* para não ter que reescrever que:
29   a(fabrica, teatro) \== a(teatro,fabrica).
30 */
31
32 conexao(X,Y) :- a(X,Y).
33 conexao(X,Y) :- a(Y,X).
34
35 /* Nós iniciais e finais do problema */
36 no_origem(casa).
37 no_destino(parque).
38 /*****************************************************************/
39 inicio(Sol_invertida) :-
40     statistics(cputime,T1),
41     no_origem(No_inicial),
42     busca_L([[No_inicial]], Solucao), /* lista de lista */
43     /* Nó inicial: entra na lista inicial */
44     reverse(Solucao, Sol_invertida),
45     length(Sol_invertida, Total),
46     no_destino(FIM), nl,
47     format('\n CAMINHO PERCORRIDO: ~w ', [Sol_invertida]),
48     format('\n NÓ de origem do mapa era: : ~w ', [No_inicial]),
```

```
49      format('\n NÓ de destino do mapa era: ~w', [FIM]),
50      format('\n Total de estados: ~d', Total),
51      Aux is (Total -1),
52      format('\n Total de movimentos: ~w', Aux),
53      statistics(cputime ,T2),
54      Temp is T2 - T1,
55      format('\n T1: ~f \t T2: ~f  msec', [T1, T2]),
56      format('\n Tempo total: ~10f  msec', Temp).
57
58
59  inicio('Nao hah mais solucoes'):- !.
60          /* para terminar com yes */
61  /*********************************************************/
62  todas_solucoes:- findall(X, inicio(X), L),
63                   write('\n RESUMO DAS SOLUCOES :'),
64                   w_L(L).
65  /*********************************************************/
66  %% condição de parada.... o nó corrente é o desejado
67  busca_L(  [ [No_corrente|Caminho] |_] , [No_corrente | Caminho ] ):-
68            no_destino(No_final),
69            member(No_final, [No_corrente|Caminho]).
70            %% termina a busca...
71
72  /* parte recursiva geral  => DETALHE: LISTA DE LISTA */
73  busca_L([Nodo|Caminho],Solucao):-
74          expandir(Nodo, Expansao_do_NODO ),
75          /* na primeira vez Caminho = [] */
76          append(Caminho, Expansao_do_NODO, Novos_caminhos),
```

```
77          /* observar que o append está ao contrário....  fazendo
78          que no próximo passo Nodo, seja um proximo deste nível */
79          busca_L(Novos_caminhos , Solucao).
80
81 expandir([Nodo|Caminho],Novos_caminhos) :-
82          findall([Novo_nodo, Nodo|Caminho],
83          (conexao(Nodo, Novo_nodo),
84           not(member(Novo_nodo,[Nodo|Caminho]))),
85           Novos_caminhos), !. /* ultima linha do findall */
86
87 expandir(_,[]):- !.
88
89 /*****************************************************/
90 w_L([]).
91 w_L([A|Cauda]):- format('\n ==> ~w ', [A]),
92                  w_L(Cauda).
93 /*****************************************************/
94 /*
95 SAIDA:
96 ?- inicio(X).
97
98
99  CAMINHO PERCORRIDO: [casa, hospital, parque]
100 Nó de origem do mapa era: : casa
101 Nó de destino do mapa era: parque
102  Total de estados: 3
103  Total de movimentos: 2
104  T1: 0.090000     T2: 0.090000   msec
```

```
 Tempo total: 0.0000000000  msec
X = [casa, hospital, parque] .

?- todas_solucoes.


 CAMINHO PERCORRIDO: [casa, hospital, parque]
 Nó de origem do mapa era: : casa
 Nó de destino do mapa era: parque
 Total de estados: 3
 Total de movimentos: 2
 T1: 0.090000     T2: 0.090000  msec
 Tempo total: 0.0000000000  msec

 CAMINHO PERCORRIDO: [casa, clube, fabrica, teatro, farmacia, hospital, parque]
 Nó de origem do mapa era: : casa
 Nó de destino do mapa era: parque
 Total de estados: 7
 Total de movimentos: 6
 T1: 0.090000     T2: 0.090000  msec
 Tempo total: 0.0000000000  msec
 RESUMO DAS SOLUCOES :
 ==> [casa, hospital, parque]
 ==> [casa, clube, fabrica, teatro, farmacia, hospital, parque]
 ==> Nao hah mais solucoes
true.

?-
```

```
133 */
```

Listing 6: Grafo da Cidade

Dicas de Programação

- Tenha um editor sensível a sintaxe do Prolog. Isto ajuda muito aos iniciantes.
- Ao carregar o programa no interpretador, certifique-se que não existam erros. Senão o código com erro não é carregado completamente.
- Evite ficar pensando obstinadamente sobre um predicado que está dando problema. Faça uma abordagem nova ou *vá andando*. Respire, saia de frente do computador, oxalá!
- Cuidado ao dar nomes de variáveis. Use nomes significativos e curtos.
- Cuidar nos predicados proibidos de *backtraking*. Exemplo é o `is`. Veja o que fazer para contornar, por exemplo:

```
cor(X) :- X is random(5),    /* sem backtraking */
          X > 0, !.
cor(X) :- cor(X).
```

- A cada predicado construído, teste! Trabalhe com o conceito de prototipação.

Predicados *Mão-na-roda*

✍ Na console digite `?- help(nome_do_predicado)` para detalhes, nos seguintes predicados *mão-na-roda*.

- `findall`, `setof` e `bagof`
- `format`
- `var` e `nonvar`
- `atom` e `string`
- `atomic`
- `ground`
- `compound` e `functor`
- `integer` e `float`
- `callable` e `call`

- `statistics` para estatísticas do sistema, tempo de cpu, etc.

```
.....................
statistics(cputime,T1),
/* seus calculos */ .........
statistics(cputime ,T2),
Temp is T2 - T1,
format('\n T1: ~f \t T2: ~f  msec', [T1, T2]),
format('\n Tempo total: ~10f  msec', Temp).
```

- `trace` e `notrace`
- `spy` e `nospy` faz uma depuração em um predicado em particular.

Os detalhes de uso deles voce descobre via `?- help` e `apropos`, manual e exemplos via *Google*.

Predicados *Baixaria*

- `if - then - else` , até tu?

```
?- (23 > 45 -> write(23) ; write(45)).
45
true.
```

- `for`
- `while`

Gerando Programas Executáveis (ou quase com o SWI-Prolog)

➽ O Prolog gera executáveis com velocidades compatíveis a linguagem C++, Delphi, e outras. Contudo, o SWI-Prolog pelo fato de ser um "*freeware*" (talvez), gera executáveis que traz consigo o seu interpretador. Ou seja, um executável no SWI-Prolog funciona como um Java, na base de um "*bytecode*".

➽ Para desenvolver aplicações comerciais ou de alto-desempenho, é interessante comprar um Prolog de algum dos muitos fabricantes.

➽ O ECL^iPS^e é mantido pela CISCO, explorado comercialmente por esta, mas é gratuito e livre.

➽ A seguir é mostrado como gerar um "*executável*" com o SWI-Prolog. Considere um programa exemplo, como este:

```
x(1).
x(5).
x(3).
par(PAR) :- x(N1),
            x(N2),
            N1 =\= N2,
            is(PAR , (N1+N2)),
            write(PAR), write(' .... '),
            write(N1), write(' .... '),
            write(N2), nl, fail.
/*, fail. */
par( 0 ) :- true.

inicio :- nl, par(_),  halt.
%% este halt é para sair do ambiente de interpreteção
%% ao final
```

Para o ambiente Linux, construa um dos dois *scripts* que se seguem:

```
#!/bin/sh
base=~/pesquisa/livro/pgms
PL=pl
exec $PL -f none -t halt -g "load_files(['$base/impares'],[silent(true)])" \
        -t inicio -- $*
```

ou

```
#!/bin/sh
pl --goal=inicio -t halt  --stand_alone=true -o saida.exe -c impares.pl
```

Detalhando:

➽ A opção `-goal=inicio` indica o ponto de início que o interpretador começar a processar. Neste exemplo, o predicado chama-se `inicio`.

➽ A opção `-t halt` indica um encerramento final do interpretador, inclusive, no processamento do executável.

Execute um destes *scripts* na linha de comando do Linux, conforme a ilustração abaixo:

```
[claudio@goedel pgms]$ sh comp1.script

6 .... 1 .... 5
4 .... 1 .... 3
6 .... 5 .... 1
8 .... 5 .... 3
4 .... 3 .... 1
8 .... 3 .... 5
```

ou

```
[claudio@goedel pgms]$ sh comp2.script
% impares.pl compiled 0.00 sec, 1,524 bytes
[claudio@goedel pgms]$ ./saida.exe

6 .... 1 .... 5
4 .... 1 .... 3
6 .... 5 .... 1
8 .... 5 .... 3
4 .... 3 .... 1
8 .... 3 .... 5
[claudio@goedel pgms]$
```

Neste último caso um executável foi gerado chamado "*saida.exe*".

➽ Estes dois scripts são equivalentes ao seguinte procedimento dentro do ambiente interpretado. Logo, este vai funcionar para outros SO's:

```
?- consult('impares.pl').
% impares.pl compiled 0.00 sec, 1,556 bytes

Yes
?- qsave_program('nova_saida.exe',
     [goal=inicio, stand_alone=true, toplevel=halt]).
Yes
```

Aqui, o predicado "*qsave_program*" gerou um executável chamado de "*nova_saida.exe*". Leia com atenção o *help* deste predicado.

A interface com linguagens como C, Java e outros é relativamente fácil. Contudo, neste momento será omitido.

Ainda sobre Gerar Executáveis

➽ Mais recentemente, o compilador do swi-prolog, o qual emula uma máquina virtual, é chamado de *swipl*. Assim, um segundo exemplo completo de compilação é dado por:

```
/*
$ swipl --goal=main -t halt -o saida2 -c fatorial.pl
% fatorial.pl compiled 0.00 sec, 1,164 bytes
$ ./saida2
6
120
5040
$ swipl --stand_alone=true  --goal=main -t halt -o saida1 -c fatorial.pl
% fatorial.pl compiled 0.00 sec, 1,164 bytes
$ ./saida1
6
120
5040
$ ls -al saida*
-rwxrwxrwx 1 root root 907617 Set  1 00:27 saida1
-rwxrwxrwx 1 root root 122822 Set  1 00:26 saida2
*/

%%% swipl -stand_alone=true  -g main -t halt -o saida.exe -c fatorial.pl
%% nesta ordem...
```

```
21
22 main :- fat(3,N), write(N), nl,
23         fat(5,X), write(X), nl,
24         fat(7,Y), write(Y), nl.
25
26 fat( 0, 1 ).
27 fat( X, Fat ) :-   X > 0,
28                    Aux is (X - 1),
29                    fat( Aux , Parcial ),
30                    Fat is ( X * Parcial ).
31
32 /*
33 Exemplificando:
34 $ swipl --stand_alone=true -g main -t halt -o saida.exe -c fatorial.pl
35 % fatorial.pl compiled 0.00 sec, 1,164 bytes
36 $ ./saida.exe
37 6
38 120
39 5040
40 $
41 */
```

Listing 7: Gerando um executável

Operações Especiais

- Alterando a memória de trabalho em tempo de execução.
- Alterações de dados e linhas de código de programa.
- Ou seja, isto permite que um programa que se *auto-modifique*

Observe o exemplo que se segue:

```
?- dynamic(algo_novo/1).
Yes
?-
?- assert(algo_novo(  alo___mundo  )).
Yes
?-
?- algo_novo(X).
X = alo___mundo ;
Yes
?-
```

Basicamente, o que se precisa fazer para remover ou adicionar um fato ou regra à MT, bastam dois passos:

1. Definir que este predicado é dinâmico na MT. Use o predicado: **dynamic(nome_do_novo/aridade)**.
2. Pronto para usar, com o **assert** ou **retract**.

Veja outros exemplos:

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:

```
?- assertz(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
```

3. Finalmente, reusando um fato já adicionado:

```
?- assert('uma_regra(X) :- algo_novo(X).').
Yes
/* usando a regra recém incluída */
?- uma_regra(Y).
Y = alo___mundo ;
Yes
?-
```

➽ Enfim, avalie o impacto do que representa **incluir** ou **excluir** uma regra durante a execução de um programa. Ou seja, potencialmente se constrói um programa que se "*auto-modifica*" durante a sua execução!

Programando com "*Elegância*"

Há um estilo de como programar **bem** em Prolog? Claro, a resposta é um **sim**. Contudo, esta elegância ou estilo de programação não é trivialmente expresso sob a forma de regras. Algumas dicas (experiências diversas) são:

- Entenda **profundamente** o problema que queres escrever em Prolog. Um problema mal entendido, dificilmente será bem implementado (se é que for);
- Escreva da mesma maneira que o problema é montado mentalmente. Assim como se fala, se escreve em Prolog. "**Declare o problema, sem se preocupar como ele é calculado passo-a-passo** ", esta é uma das máximas do Prolog;
- Evite o uso de operadores tradicionais como: $>$, $<=$, is ...etc, isto normalmente revela uma proximidade com a programação procedural. O foco do Prolog é "**Casamento de Padrões**".

Pense que dois objetos podem ser equivalentes ou diferentes, apenas isto. Logo, eles casam ou não;

- Quando uma implementação começa ficar complicada, é porque alguma premissa assumida previamente, está errada. Volte atrás, e refaça tudo sob um outro enfoque. Refazer um novo programa por inteiro, normalmente é mais simples do que "*forçar*" um problema mal estruturado, convergir em uma solução aceitável;
- Consulte os "*grandes mestres*" em Prolog. Aprender Prolog com programadores experientes é uma boa dica. Alguns conhecidos no Brasil: Pedro Porfírio (porfirio@unifor.br), Edilson Ferneda (ferneda@pos.ucb.br). Finalmente, a lista de discussão do SWI-Prolog também é fácil encontrar muitos peritos em Prolog.

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://dobrev.com/download.html
- http://www.swi-prolog.org/
- http://www.amzi.com (tem vários artigos e tutoriais que ilustram o uso e aprendizado do Prolog, respectivamente. Um site fortemente recomendado, pois há um farto material gratuito, incluindo um ambiente de programação.)
- http://www.arity.com tem um outro Prolog free
- http://www.ida.liu.se/~ulfni/lpp/
- Strawberry Prolog: http://www.dobrev.com/

- http://www.cse.unsw.edu.au/~billw/cs9414/notes/prolog/intro.html
- http://www.mars-attacks.org/~boklm/prolog/

Alguns Bons Livros

- 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;
- Há um título de um livro de IA, que é "*aproximadamente*" é: "*Solving Complexs Problems with Artificial Intelligence*", cuja linguagem utilizada nas soluções é o Prolog.

Sugestões

- Sugestões de exemplos são bem-vindos
- Envie para: `claudio@joinville.udesc.br`