



1/176



Lógica Matemática

Fundamentos de Prolog e *ECLiPSe*

(Joinville, 12 de março de 2014)

Claudio Cesar de Siqueira

Rogério Eduardo da Silva

Departamento de Ciência da Computação

Bloco F - 2o. Piso - F-209 - Sala 13

Joinville - SC

email: claudio@colmeia.udesc.br



Back

Close



Sumário

1	Introdução	7
2	Proposta destes Slides	8
3	Requisitos	10
4	<i>Index of Distribution</i>	11
5	Instalando o <i>ECLiPSe</i> no Linux	12
6	Implementações e Ambientes	17
7	Características	18
8	Resolvendo um Problema	19



Back

Close

9	Explicando o Fluxo da Máquina Prologuiana	26
10	Um Predicado Composto	27
11	Implementando em <i>ECLiPSe</i>	30
12	Ambiente de Programação: <i>ECLiPSe</i> console	32
13	Ambiente de Programação: <i>TKECLiPSe</i>	33
14	Um Outro Exemplo: Os Trís Mísicos	34
15	Apresentação do Prolog	36
16	Quanto ao Eclipse	42
17	Ambiente Gráfico de Programação: <i>TKECLiPSe</i>	44
18	O <i>Tracer</i> para Auto-aprendizagem	45
19	Sintaxe do Prolog	46



20 Resumindo Via Exemplo	65
21 Exercícios de <i>Warmup</i>	68
22 Semântica Operacional	71
23 Recursividade	76
24 Iteração Clássica	77
25 Recursão Ilustrada	78
26 Recursão Ilustrada 2	79
27 Quase Tudo São Flores	80
28 Um Outro Clássico: Os Ancestrais	87
29 Seu Processamento	90
30 Functores	100



4/176



Back

Close

31	Definições dos Functores	101
32	Listas	107
33	Resumindo o Fluxo do Cálculo Recursivo	108
34	O Fluxo Operacional das Listas e Análogo	109
35	Uma Lista Genérica	110
36	A Sintaxe das Listas	111
37	Definições sobre Listas	112
38	Listas: Uma Auto-Definição	117
39	Problemas de Buscas em IA	141
40	Reusando o Conhecimento de Listas e Functores	145
41	Resolvendo com Busca em Profundidade	146



5/176



Back

Close

42	Resolvendo com Busca em Largura	149
43	Dicas de Programação	155
44	Predicados <i>Miljo-na-roda</i>	157
45	Predicados <i>Baixaria</i>	159
46	Gerando Programas Executíveis (ou quase com o SWI-Prolog)	160
47	Operações Especiais	167
48	Programando com “ <i>Elegância</i> ”	171
49	Sites Interessantes	173
50	Alguns Bons Livros	175
51	Sugestões	176



6/176



Back

Close



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, ...



Back

Close



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, faça 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 *ECLⁱPS^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.



Back

Close

- 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.*



9/176



Back

Close



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 *ECLiPSe*: <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: 12 de março de 2014



Back

Close



Instalando o *ECLⁱPS^e* no Linux

Seguindo os passos abaixo numa console Linux:

Faça os downloads:

.....

```
mkdir clp_eclipse
mv eclipse_misc.tgz clp_eclipse/
mv eclipse_bas
```

.....

```
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
```



Back

Close



13/176

```
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 caminho de onde está; o seu Java



Back

Close

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:



14/176



Back

Close

```
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 caminho 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



15/176

tkeclipse

eclipse

◀

▶

Back

Close



Back

Close



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 *ECLⁱPS^e*, objeto de investigação da segunda parte deste curso.



Back

Close



18/176

Características

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



Back

Close



Resolvendo um Problema

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

Fui hãŁj uma festa e apresentado hãŁj trãŁjs casais. Os maridos tinham profissãŁjes e esposas distintas. ApãŁjs alguns goles me confundi quem era casado com quem, e as profissãŁjes. Apenas lembro de alguns fatos, entãŁjo me ajude descobrir quem sãŁjo estes casais, com base nos seguintes dados:

1. O mãŁjdico ãŁ casado com a Maria;
2. O Paulo ãŁ advogado;
3. PatrãŁjcia nãŁjo ãŁ casada com Paulo;
4. Carlos nãŁjo ãŁ mãŁjdico.



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

Modelagem

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

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

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

O problema se resume em encontrar: $((H_1,M_1,P_1), (H_2,M_2,P_2), (H_3,M_3,P_3))$

Codificando a Descrição Acima

```
1 /*  
2 Os demais nomes de nossos personagens são:
```



20/176



Back

Close



```
3  H = { carlos (c) , luiz (l) , paulo (p) }
4  M = { maria , lucia , patricia }
5  P = { advogado (a), medico (m), engenheiro (e) }
6
7  Modelagem: (H,M,P) logo ((H1,M1,P1), (H2,M2,P2), (H3,M3,P3))
8  */
9
10 /* aqui comeca o codigo */
11
12 prof(advogado).
13 prof(medico).
14 prof(engenheiro).
15 esposa(patricia).
16 esposa(lucia).
17 esposa(maria).
18
19 x :- deduz(X, Y, Z), write(X), nl, write(Y), nl, write(Z), nl.
20 deduz(( carlos ,M1,P1), (luis ,M2,P2), (paulo ,M3,advogado)) :-
21
22     esposa(M1),
23     esposa(M2),
24     esposa(M3),
25     prof(P1),
26     prof(P2),
27     M3 \== patricia ,
28     P1 \== medico ,
29     P1 \== advogado ,
30     P2 \== advogado ,
```





22/176

```
31 P1 \== P2,
32 M1 \== M2,
33 M1 \== M3,
34 M2 \== M3.
35
36 /* A restriccao a) nao foi atendida:
37 %% O medico eh casado com a Maria
38 logo o programa acima deve sofrer uma modificacao:
39 basta reler as hipoteses e voce vai concluir que o
40 programa se reduz a:
41 */
42
43 y :- deduz2(X, Y, Z), write(X), nl, write(Y), nl, write(Z), nl.
44
45 deduz2((carlos,M1,P1), (luis,maria,medico), (paulo,M3,advogado)) :-
46     esposa(M1),
47     esposa(M3),
48     prof(P1),
49     M3 \== patricia,
50     P1 \== medico,
51     P1 \== advogado.
52
53
54 /*?- ['casais.pl'].
55 % casais.pl compiled 0.00 sec, 0 bytes
56
57 Yes
58 ?- x.
```



Back

Close



```
59 carlos , patricia , engenheiro
60 luis , lucia , medico
61 paulo , maria , advogado
62
63 Yes
64 ?- y.
65 carlos , patricia , engenheiro
66 luis , maria , medico
67 paulo , lucia , advogado
68
69 Yes
70 ?-
71 */
```

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.



Back

Close

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 \equiv medico.
5. Altere e/ou inclua algumas regras, afim de restringir a quantidade de respostas.



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



25/176



Back

Close



Explicando o Fluxo da Máquina Prologiana

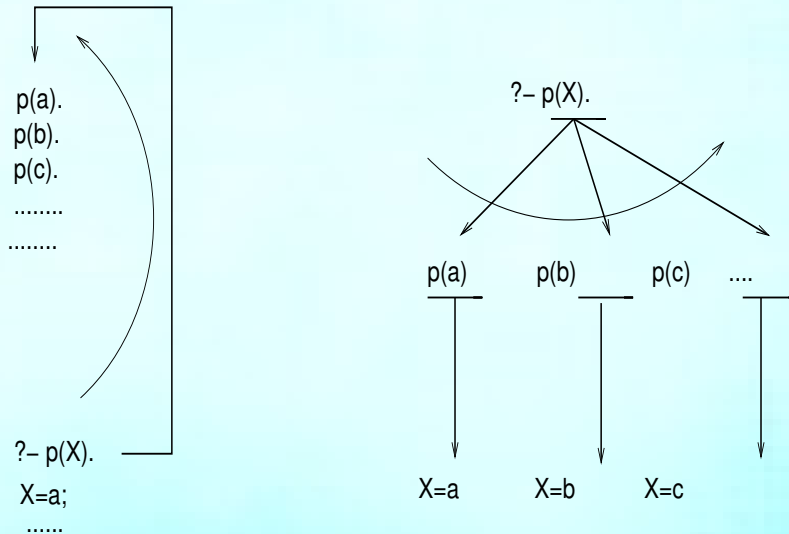
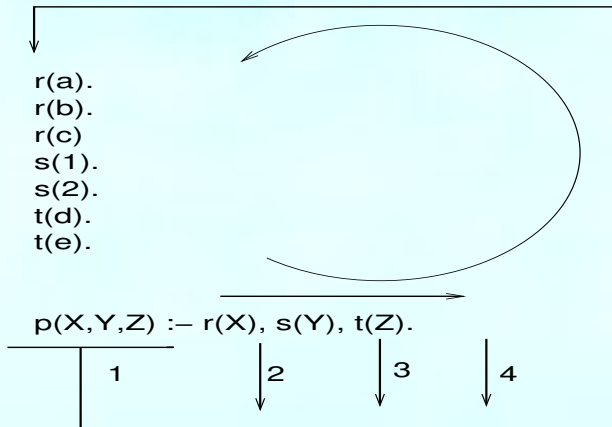


Figura 1: Fluxo b sico de um predicado em Prolog



Um Predicado Composto



?- $p(X,Y,Z).$
 $X=a$
 $Y=1$
 $Z=d$

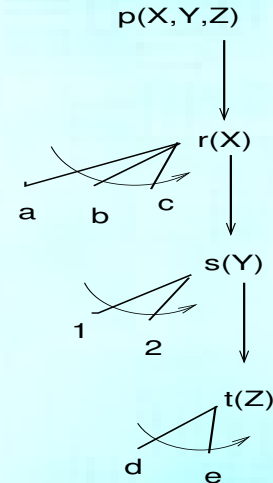


Figura 2: Fluxo b sico de um predicado composto em Prolog



Back

Close

Implementando estas Figuras



28/176

```
1  r(a).
2  r(b).
3  r(c).
4  s(1).
5  s(2).
6  t(d).
7  t(e).
8  p(X,Y,Z) :- r(X), s(Y), t(Z).
9  saida :- p(X, Y, Z),
10          format('X:~w~\tY:~w~\tZ:~w~', [X, Y, Z]) .
11 /*
12 ?- saida.
13 X: a      Y: 1      Z: d
14 true ;
15 X: a      Y: 1      Z: e
16 .....
17 X: c      Y: 2      Z: d
18 true ;
19 X: c      Y: 2      Z: e
20 true.
21 */
```

Listing 2: Semântica operacional do Prolog



Back

Close

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!



29/176



Back

Close



Implementando em ECL^iPS^e

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

```
1  /* :-lib(ic_symbolic). */
2  :- lib(ic).
3
4  prof(1, medico).
5  prof(3, advogado).
6  prof(2, engenheiro).
7  m(1, patricia).
8  m(2, lucia).
9  m(3, maria).
10
11 x :- deduz((carlos,M1,P1), (luis,M2,P2), (paulo,M3,advogado)),
12      m(M1,E1), m(M2,E2), m(M3,E3),
13      prof(P1,P1_out), prof(P2,P2_out),
14      writeln("A saída dada por: "),
15      writeln((carlos,E1,P1_out)),
16      writeln((luis,E2,P2_out)),
17      writeln((paulo,E3,advogado)).
18
19 deduz((carlos,M1,P1), (luis,M2,P2), (paulo,M3,advogado)) :-
```



Back

Close



```
20 [M1,M2,M3] :: 1 .. 3,  
21 [P1,P2]   :: 1.. 2,  
22 P2 #= 1,  
23 M2 #= 3,  
24 M3 #\= 1,  
25 P1 #\= 1,  
26 P1 #\= P2,  
27 alldifferent ([M1,M2,M3]).
```

Listing 3: 3 casais e profissões dos maridos





Ambiente de Programação: *ECLⁱPS^e* console

```
claudio@g_cantor: ~/principal/cursos_disciplinas/ecoa
File Edit View Terminal Tabs Help
claudio@g_cantor:~/principal/cursos_disciplinas/ecoa$ eclipse
ECLiPSe Constraint Logic Programming System [kernel]
Kernel and basic libraries copyright Cisco Systems, Inc.
and subject to the Cisco-style Mozilla Public Licence 1.1
(see legal/cmpl.txt or www.eclipse-clp.org/licence)
Source available at www.sourceforge.org/projects/eclipse-clp
GMP library copyright Free Software Foundation, see legal/lgpl.txt
For other libraries see their individual copyright notices
Version 6.0 #80 (i386_linux), Mon Mar 23 03:29 2009
[eclipse 1]: 6 > 7.

No (0.00s cpu)
[eclipse 2]: X = 17, (X > 17 -> write(X); write("nao")).
nao
X = 17
Yes (0.00s cpu)
[eclipse 3]: writeln(" boa noite ").
boa noite

Yes (0.00s cpu)
[eclipse 4]: █
```

Figura 3: A console básica do *ECLⁱPS^e* (análoga aos Prologs)



Ambiente de Programação: *TKECLiPSe*



33/176

```
File Query Tools Help
Query Entry
eclipse  [Download] [Upload]
run more Yes make interrupt

Results
Abort
?- x.
Abort
?- x.
Abort
?- x.
Yes (0.00s cpu)

Output and Error Messages
type error in printf("\n (carlos, %s , %s)", patricia, engenheiro)
plr_casais.ecl compiled 2584 bytes in 0.01 seconds
type error in printf("\n (carlos, %w , %w)", patricia, engenheiro)
plr_casais.ecl compiled 2584 bytes in 0.00 seconds
type error in printf("\n %u , %u", patricia, engenheiro) in modu.
File plr_casais.ecl, line 13: Singleton variable E2
File plr_casais.ecl, line 13: Singleton variable E3
File plr_casais.ecl, line 14: Singleton variable P2_out
plr_casais.ecl compiled 2424 bytes in 0.01 seconds
carlos, patricia, engenheiro
plr_casais.ecl compiled 2708 bytes in 0.01 seconds
calling an undefined procedure printf("\n A saida e dada por : \n
plr_casais.ecl compiled 2708 bytes in 0.01 seconds
calling an undefined procedure printf(" A saida e dada por : "> ir
calling an undefined procedure printf(" A saida e dada por : "> ir
plr_casais.ecl compiled 2720 bytes in 0.02 seconds
A saida e dada por :
carlos, patricia, engenheiro
luis, maria, medico
paulo, lucia, advogado
```

Figura 4: O *ECLiPSe* com a biblioteca TK



Um Outro Exemplo: Os Trilés Músicos

“Trilés músicos, João, Antônio e Francisco, tocam harpa, violoncelo e piano. Contudo, não se sabe quem toca o quê. Sabe-se que o Antônio não é o pianista. Mas o pianista ensaia sozinho é Terê. O João ensaia com o Violoncelista é as Quintas. Quem toca o quê?”

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

```
1 instrumento(harpa).
2 instrumento(violoncelo).
3 instrumento(piano).
4 dia(3).
5 dia(5).
6
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),
```



Back

Close



```
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).
```

Filizil, certo? Veja a execuifilizo:

```
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 seria melhorado e como o anterior refeito com recursos da PLR.



Back

Close



Apresentação do Prolog

- Prolog é uma linguagem de programação implementada sobre um paradigma lógico (Lógica de 1ª. 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 John Wiley & Sons, 2000.



Características

- Manipula símbolos (objetos) por natureza, logo $7 + 13$ tem várias conotações;
- Um ítemo, 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



37/176



Back

Close

de compiladores, etc.



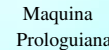
38/176



Back

Close

- ✓ 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.








Exemplos e Como Trabalhar

- ✓ Alíxi-m do Prolog ou do Eclipse instalado, ïxi necessïxi-rio ter um editor de texto padrixi-jo ASCII. Aquele de sua preferïxi-jncia.
- ✓ Qualquer um desde que apresente o nixi-mero das linhas no texto.
- ✓ Caso a escolha seja um Prolog com interface grïxi-fica tipo XPCE (<http://www.swi-prolog.org/>), o qual funciona muito bem sob o Linux, tal editor ïxi acionado pelo comando:
?- edit(file('nome do pgm.pl')).

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

✓

Veja os exemplos iniciais:

?- 2 > 3.

No

?- 2 == 2.

Yes

?- mae(eva, abel).



40/176



Back

Close

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 masculino */
```

```
X = joao
```

Yes

```
?- homem( x ). /* x masculino */
```

No

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



41/176



Back

Close



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 ao **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 digitar, ou carregue um arquivo para execução na opção **Compile File**.
- ✓ Veja na janela de **Output**. Se nenhum erro, estamos indo bem.



- ✓ Viŕ 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 ŕ minimalista.



43/176



Back

Close



44/176

Ambiente Gráfico de Programação: *ECLiPSe*

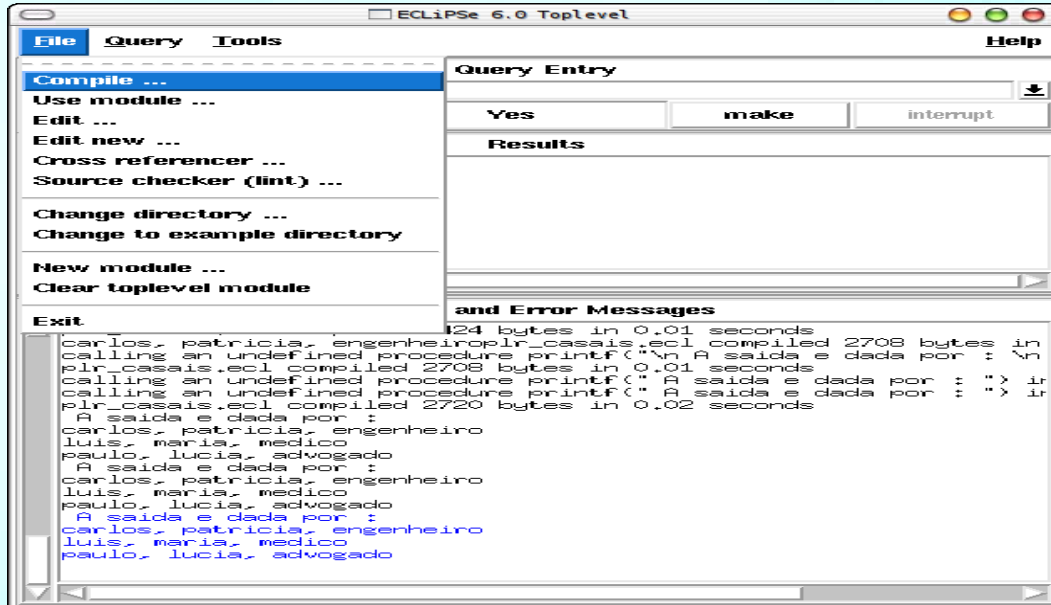


Figura 6: O *ECLiPSe* gráfico – a aba File

O Tracer para Auto-aprendizagem



45/176

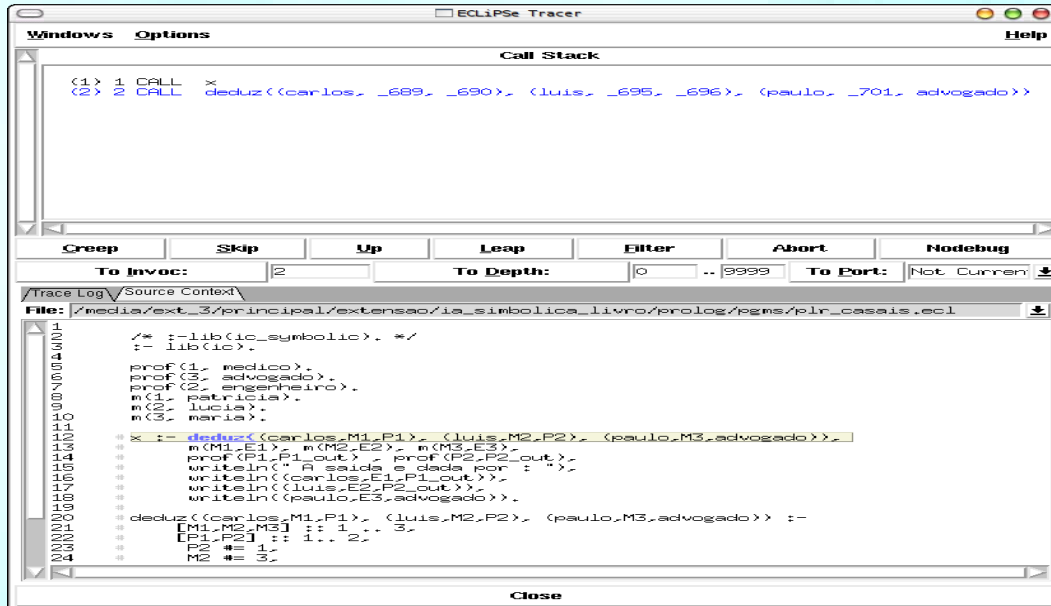


Figura 7: O depurador gráfico do *ECLiPSe*



Back

Close



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 identificadas novamente no programa inicial.

1o. tipo: são as **questões** ou *goals* (*query*), isto é, uma pergunta à uma base de conhecimento
 ... “*a ideia 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
?-
Yes
```



Back

Close

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

```
?- listing(homem).
```

```
homem(joao).
```

```
homem(jose).
```

```
homem(jedro).
```

Yes

3o. tipo: sãŁjo as **regras** ... que “*aproximadamente*” sãŁjo verdades ou teoremas condicionais, isto ĩŁ: *necessitam de uma prova lĩŁgica!*

```
?- listing(mortal).
```

```
mortal(A) :- homem(A).
```

```
/* obs. o sãŁmbolo :- ĩŁ o implica ao contrãŁrio
```

```
/* homem --> mortal
```

```
leia-se ... para demonstrar que algum X ĩŁ
```

```
mortal, preciso demonstrar e provar que A ĩŁ
```

```
um homem */
```

Yes



47/176

<- */



Back

Close

Resumindo o Prolog

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



48/176



Back

Close

Detalhando a Sintaxe

➔ Há alguns detalhes de como se montam esses predicados *prologianos*, os mais trabalhosos são os das regras. 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:

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. e. 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 “*pit*” da regra;



49/176



Back

Close



50/176

- Logo a regra possui uma “*cabeçalho*”, que é o predicado “*irmão*”;
- O corpo é composto pelos predicados: “*pai*” e “ $\backslash == /$ ”.
- 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;



Back

Close

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



51/176



Back

Close



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 da **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?



53/176



Back

Close

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 sintá-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.



55/176



Back

Close

Alguns comandos para usar no *ECLiPS^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 reserved.
```

```
For help, use ?- help(Topic). or ?- apropos(Word).
```

```
Yes
```

```
?-
```

- Idem ao *eclipse*. Veja o passo-a-passo para instalá-lo 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 est ; 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(padriço_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



Back

Close



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
```



Back

Close



60/176

```
?- notrace.
```

```
e
```

```
?- nodebug. /* ou Ctrl-C para sair do modo debug */
```

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

```
==> <abort.> sai do debug/trace ou da pendencia...
```

```
ou <Ctrl 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:
```



Back

Close

'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.

- Niŕjo esqueŕja que:

?- trace, t.

e

?- spy(t), t.

siŕjo equivalentes !



61/176



Back

Close

Outros Detalhes

- Correções automáticas:

```
?- pratos(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,



62/176



Back

Close

ou seja, torna-a sempre falsa; esse “*fail*” foriçja o mecanismo “*backtracking*”.



63/176



Back

Close



64/176

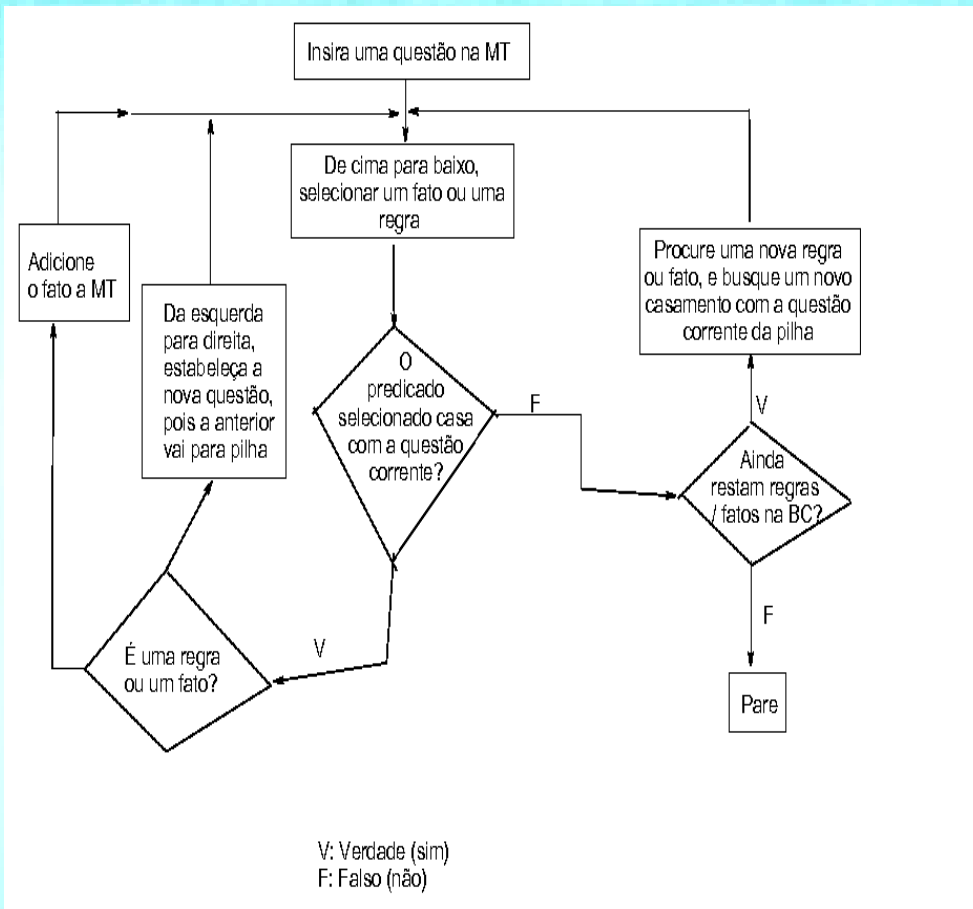


Figura 8: Fluxo *Aproximado* de Inferências em Prolog





Resumindo Via Exemplo

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

```
1  x(7).  
2  x(5).  
3  x(3).  
4  
5  par(Xpar) :- x(N1), x(N2), N1 \= N2,  
6               Xpar is (N1+N2),  
7               write(N1) , write(' .... '),  
8               write(N2) , write(' .... '),  
9               write(Xpar) ,  
10              nl ,  
11              fail .  
12  
13 ?- par(N).
```

Cuja árvore de busca é dada pela figura 20:





66/176

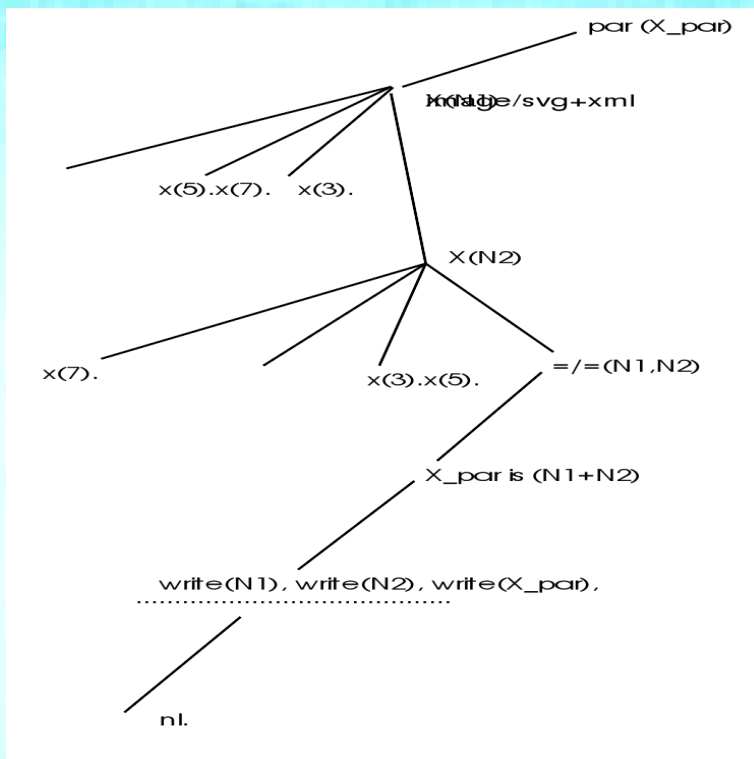


Figura 9: Árvore de busca para regra par

➔ Aconselho estudar cada ponto deste exemplo, pois o mesmo resume todo conhecimento até o momento sobre Prolog. Avance



Back

Close

İki prefixima seilijio, apenas se este exercicio for dominado em detalhes.



67/176



Back

Close



Exercícios de *Warmup*

1. Construa a árvore genealógica 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 resultados? Escreva os seus significados, tomando por base os conceitos de termos ou átomos, *matching* (casamento), e unificação:

```
1 ?- 2 > 3.  
2 ?- >(2, 3).  
3 ?- 2 == 2.  
4 ?- a(1, 2) = a(X, X).  
5 ?- a(X, 3) = a(4, Y).  
6 ?- a(a(3, X)) = a(Y).  
7 ?- 1+2 = 3.
```



Back

Close



```
8 ?- X = 1+2.  
9 ?- a(X, Y) = a(1, X).  
10 ?- a(X, 2) = a(1, X).  
11 ?- 1+2 = 3  
12 ?- X + 2 = 3 * Y.  
13 ?- X+Y = 1+2.  
14 ?- 1+Y = X + 3.  
15 ?- pai(X, adao) = pai(abel, Y).  
16 ?- X+Y = 1+5, Z = X.  
17 ?- 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 *backtracking*?



70/176



Back

Close



Semântica Operacional

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

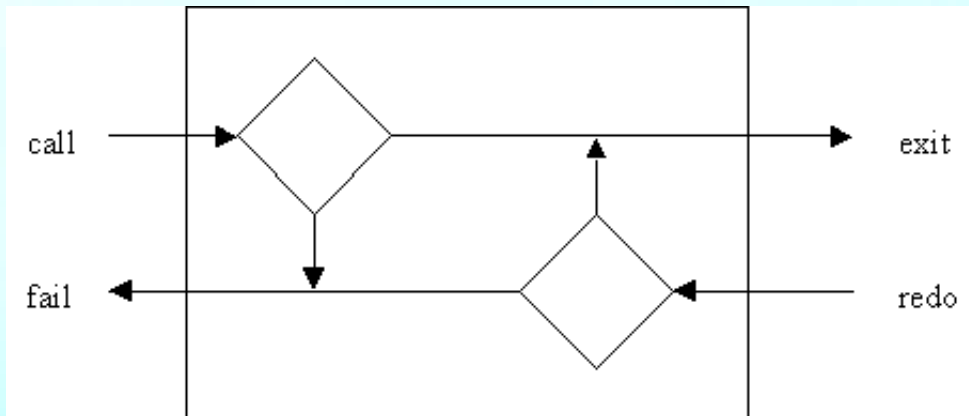


Figura 10: Um fluxo operacional de um predicado típico



➤ Retirado de [http:](http://grack.com/downloads/school/enel553/report/prolog.html)

[//grack.com/downloads/school/enel553/report/prolog.html](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 exhaustivamente 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**.



72/176



Back

Close

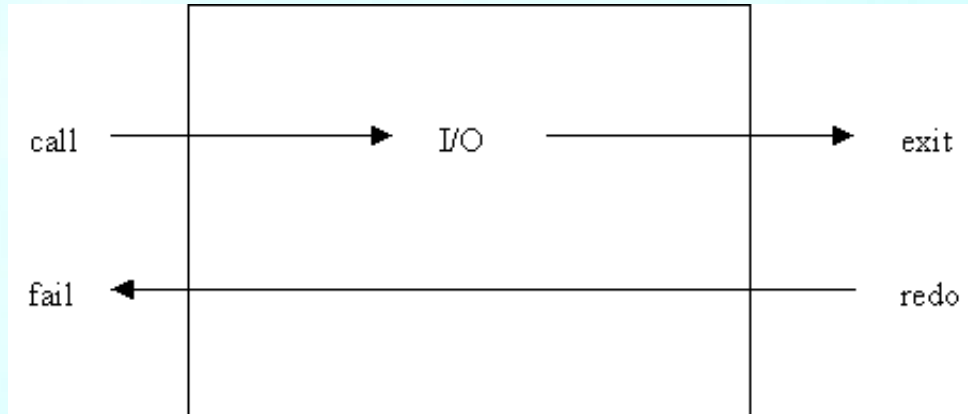


Figura 11: Um fluxo operacional de um predicado de E/S - sem redo

Exemplo: o predicado `write/1`



Back

Close

Um predicado que sempre falhe



74/176

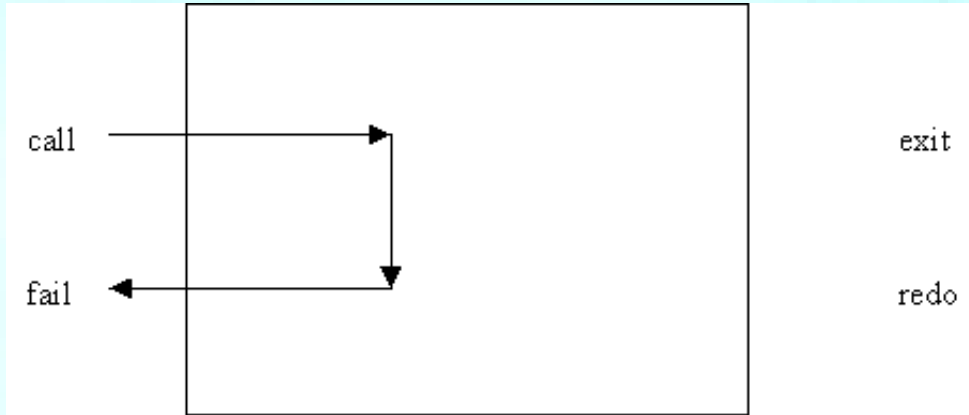


Figura 12: Um fluxo operacional de um predicado com retorno em fail

Exemplo: o predicado fail/0



Back

Close



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:



Back

Close

Iteração Clássica



77/176

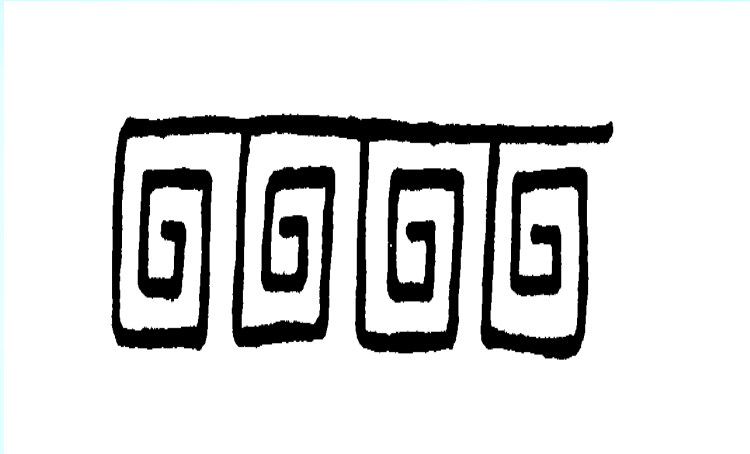


Figura 14: Uma iteração ilustrada



Back

Close

Recursiļjo Ilustrada



78/176

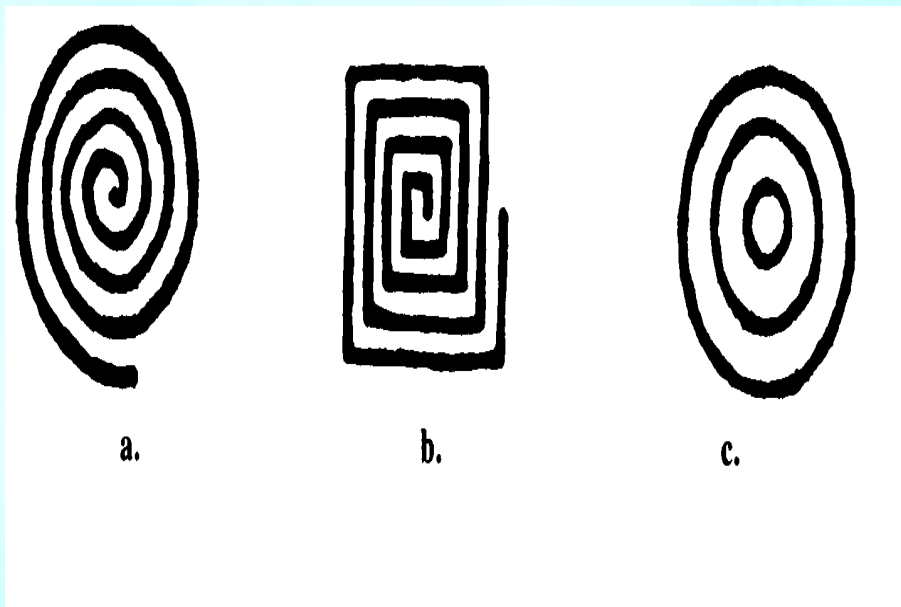


Figura 15: Recursiļjo ilustrada



Recursiļjo Ilustrada 2



79/176

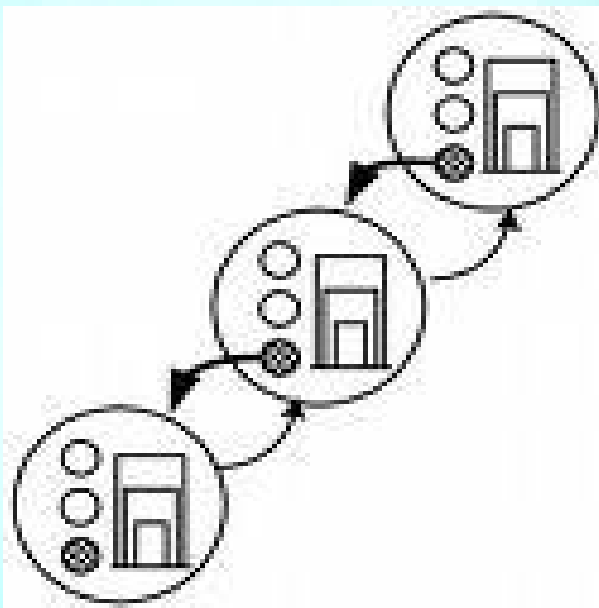


Figura 16: Recursiļjo ilustrada 2



Quase Tudo Siflo Flores



80/176

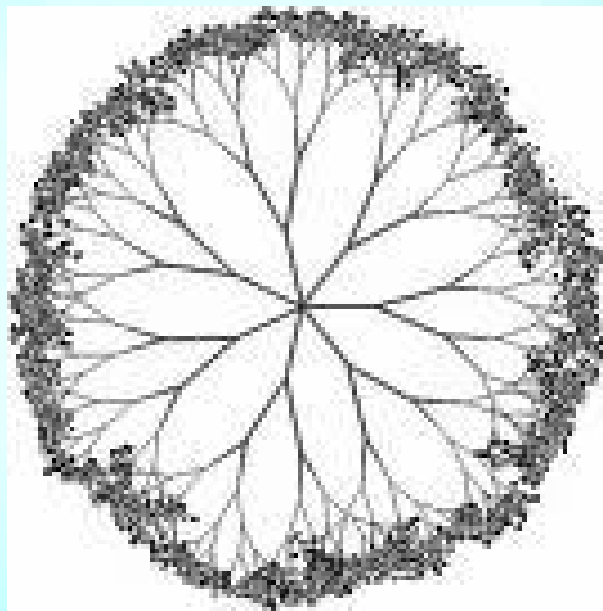


Figura 17: Recursiflo ilustrada – flores



Back

Close



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

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

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

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

O que é um fato verdadeiro pois:

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

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).



Back

Close

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 recursiva, 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.



82/176



Back

Close

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

```
1 s(1,1) :- true.           /* REGRA #1 */
2 s(N, S) :-                 /* REGRA #2 */
3     N > 1,
4     Aux is (N-1),
5     format('~\n N: ~w \t AUX: ~w \t PARCIAL: ~w \t S: ~w',
6             [N, Aux, Parcial, S]),
7     s(Aux, Parcial),
8     format('~\n ==> Apos o casamento da REGRA #1: '),
9     S is (N + Parcial),
10    format('~\n N: ~w \t AUX: ~w \t PARCIAL: ~w \t S: ~w',
11            [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).



83/176





84/176

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



Back

Close

A execu    o na console:



85/176

```
1 ?- s(5,X).
2
3 N: 5      AUX: 4      PARCIAL: _G254      S: _G181
4 N: 4      AUX: 3      PARCIAL: _G269      S: _G254
5 N: 3      AUX: 2      PARCIAL: _G284      S: _G269
6 N: 2      AUX: 1      PARCIAL: _G299      S: _G284
7 ==> Apos o casamento da REGRA #1:
8 N: 2      AUX: 1      PARCIAL: 1          S: 3
9 ==> Apos o casamento da REGRA #1:
10 N: 3      AUX: 2      PARCIAL: 3          S: 6
11 ==> Apos o casamento da REGRA #1:
12 N: 4      AUX: 3      PARCIAL: 6          S: 10
13 ==> Apos o casamento da REGRA #1:
14 N: 5      AUX: 4      PARCIAL: 10         S: 15
15 X = 15
```



Back

Close

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.



86/176



Back

Close



Um Outro Clássico: Os Ancestrais

O ancestral de X , é o mínimo o pai de X . Logo, um avô de 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:

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



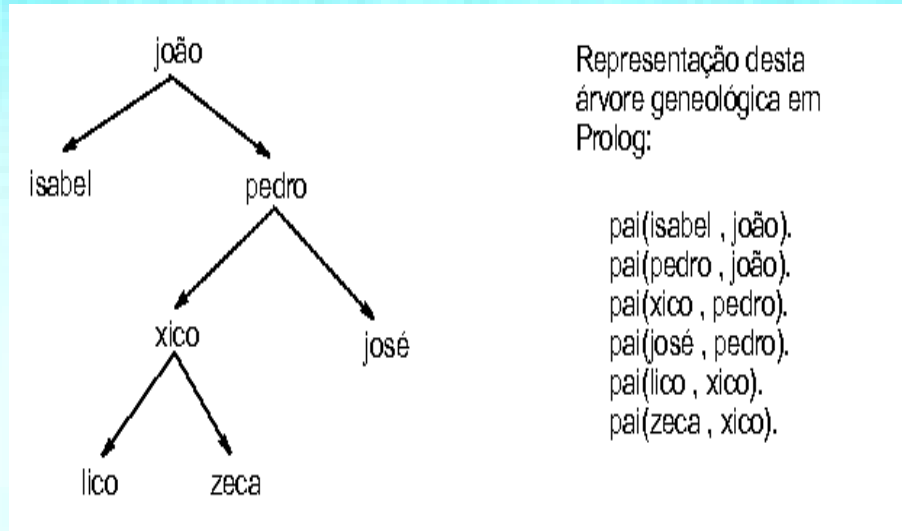


Figura 18: Uma árvore que representa os descendentes de alguém

Este código em Prolog, agora em inglês:

```
1 parent(john , paul) .      /* paul is john's parent */  
2 parent(paul , tom) .      /* tom is paul's parent */  
3 parent(tom , mary) .      /* mary is tom's parent */  
4 ancestor(X,Y) :- parent(X,Y).  
5 /* If Y is a parent of X, then Y is an ancestor of X */  
6  
7 ancestor(X,Y) :- parent(X,Z) , ancestor(Z,Y).
```



Back

Close

8
9

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





90/176

Seu Processamento

Calculando ?- ancestor(john,tom).

```
1 CALL ancestor(john,tom).
2     CALL parent(john,tom).
3     FAIL parent(john,tom).
4     CALL parent(john,Z).
5         TRY Z=paul
6         CALL ancestor(paul,tom).
7             CALL parent(paul,tom).
8             SUCCEEDS parent(paul,tom).
9             SUCCEEDS ancestor(paul,tom).
10        SUCCEEDS with Z=paul
11    SUCCEEDS ancestor(john,tom).
```

Repita o exemplo com nomes conhecidos de sua família.



Back

Close

Outros Exemplos de Recursividade

Um simples sistema de menu : alíjım do uso da recursividade, este exemplo mostra o que fazer quando em aparentemente nãıjıo hãıjı predicados disponãıjıveis ao que se deseja; ou seja: “ $X \leq 3$ ” nãıjıo existe, mas como opãıjıjıjıo equivalente tem-se:

- “ $X \leq 3$ ” ;
- “ $\backslash + (X \geq 3)$ ”.

```
menu(0).
menu(_) :-
    repeat,
    write('.....'), nl,
    .....
    write('.....'), nl,
    write(' DIGITE A OPCA0: '),
    read(X),
    X >= 0,
    \+(X >= 3), /* isto ãıjı: X <= 3 */
    /* X \== 0 ãıjı equivalente a: \+(X == 3) */
    acao(X),
    menu(X).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
acao(0). /* continua acoes etc */
```



91/176



Back

Close

```
acao(1).  
acao(2).  
acao(3).
```



92/176



Back

Close

Clássico “Sócrates” com fail :



93/176

```
1      human(socrates).          % facts about who is human
2      human(aristotle).
3      human(plato).
4      god(zeus).                % and who is a god
5      god(apollo).
6      mortal(X) :- human(X).
7      % a logical assertion that X is mortal if X is
8
9  Fazendo as perguntas:
10     ?-mortal(plato).           % is Plato mortal?
11     yes
12     ?-mortal(apollo).         % is apollo mortal?
13     no
14     ?-mortal(X).              % for which X is X mortal?
15     X = socrates ->;
16     X = aristotle ->;
17     X = plato ->;
18     no                         %human
19
20  Evitando este último "no", usando a recursividade:
21
22  mortal_report :-
23      write('Mostre todos mortais conhecidos:'), nl, nl,
24      mortal(X),
```



Back

Close



94/176

```
25      write(X), nl,  
26      fail.  
27 mortal_report.    /* ou::      mortal_report :- true. */  
28  
29 Entiço:  
30     ?- mortal_report.  
31     Mostre todos mortais conhecidos:  
32     socrates  
33     aristotle  
34     plato  
35     yes
```

Veja a seqüência dos fatos e da saída.



Back

Close

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

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

O que é um fato verdadeiro pois:

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

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:

$fatorial(N, Fat) :: Nant = (N-1), fatorial(Nant, Fat_Nant) \text{ e } Fat = (N * Fat_Nant).$



95/176



Back

Close



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)	#	?..... →	↖.....



Back

Close



“*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.

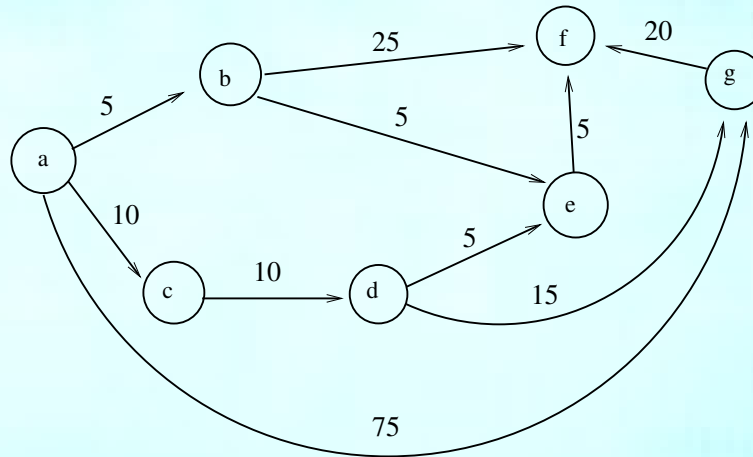


Figura 19: Um grafo que define custos entre as cidades - unidirecional

Para expressar a ideia 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 \Leftrightarrow 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  
?-
```



98/176



Back

Close

Triangulo de Astericos : ir para os exercíçios ...



99/176



Back

Close



Funtores

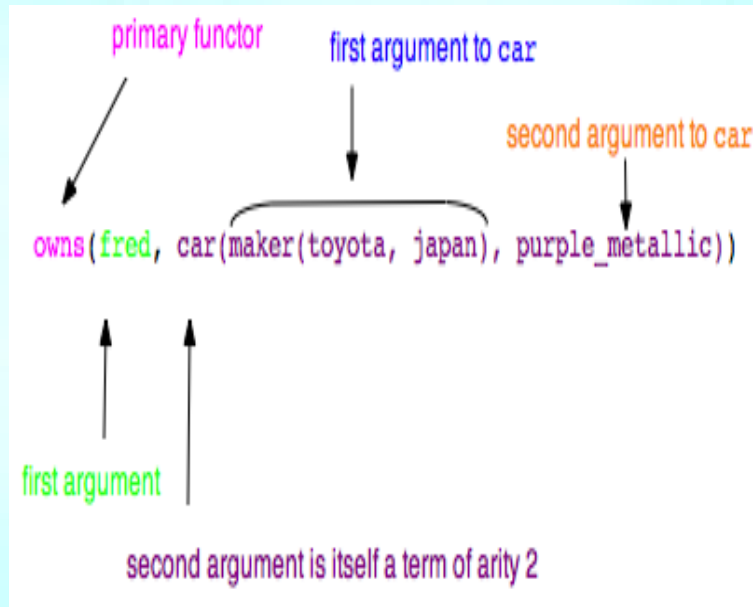


Figura 20: Um exemplo de functor – um termo composto



Back

Close



Definições dos Funtores

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

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

➔ Um domínio de objetos mapeados é 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)))  
    ) ).
```



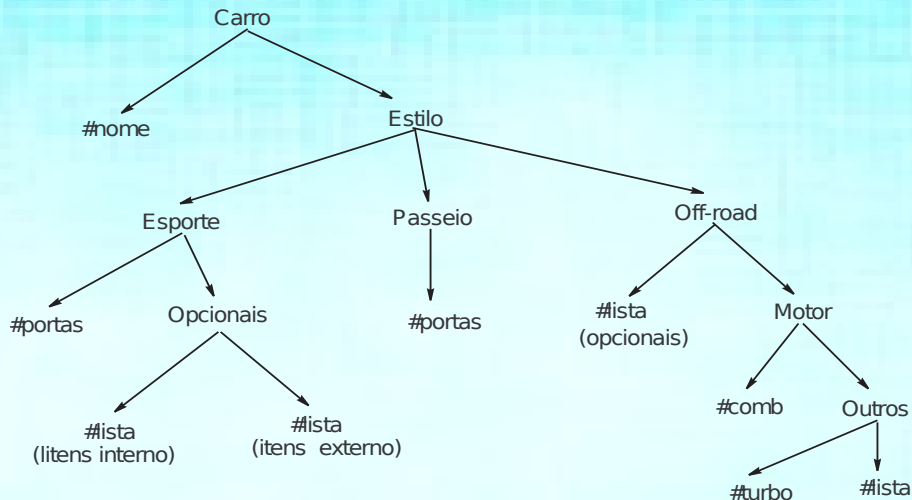


Figura 21: Representando o conceito hierárquico com um functor

Motivação

A principal observação do exemplo acima é a hierarquia “*natural*” e estabelecida, para representar um conhecimento. Tal re-



Back

Close



presenta uma linguagem conhecida por “*frames*”, que incorpora a maioria dos conceitos da programação orientada-objetos. Em resumo tem-se:

1. Construção de fatos genéricos (próximo ao conceito de classes, instâncias, etc.);
2. Quantidade variável de argumentos, leva a 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 funtores podem ser vistos como uma estruturando os dados sob formas dos “*REGISTROS*” convencionais.

Exemplo

Seja o programa abaixo:

```
artista("Salvador Dali", dados(pintor(surrealista), regioao( "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? */
```



Back

Close

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



106/176



Back

Close



Listas

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

[Back](#)[Close](#)



Resumindo o Fluxo do Cálculo Recursivo

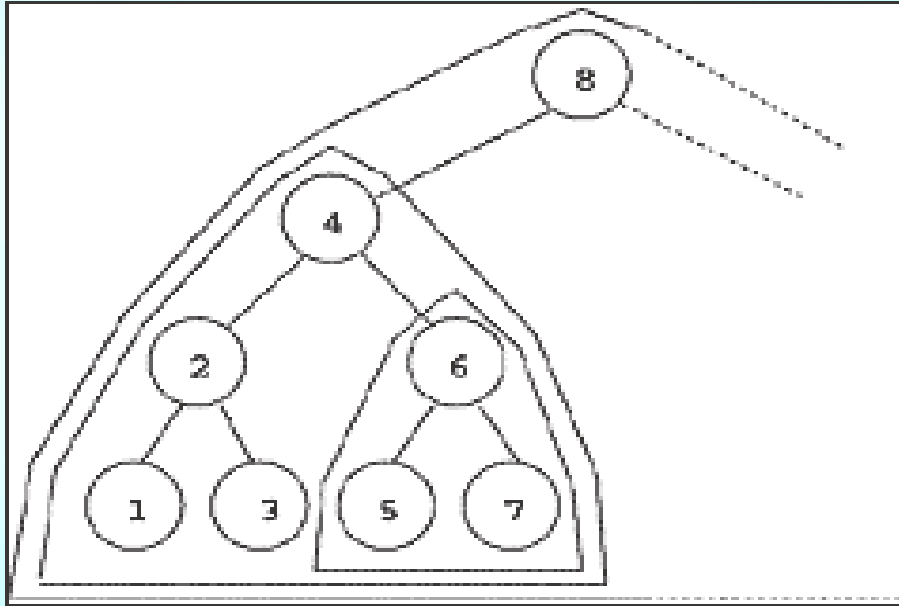


Figura 22: Cálculo Recursivo 1



Back

Close

O Fluxo Operacional das Listas ï£­ Anï£­logo



109/176

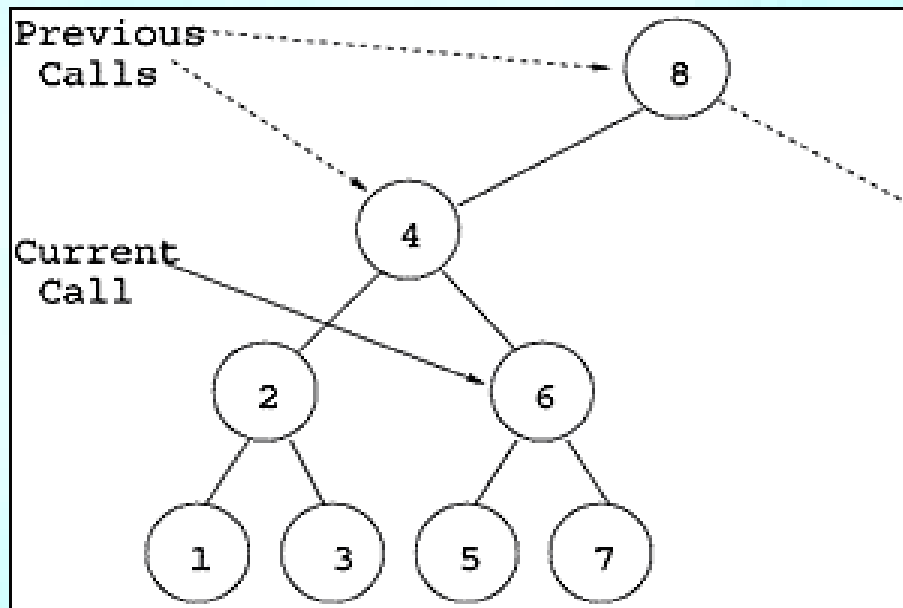


Figura 23: Cï£­lculo Recursivo 2



Back

Close



Uma Lista Genélica

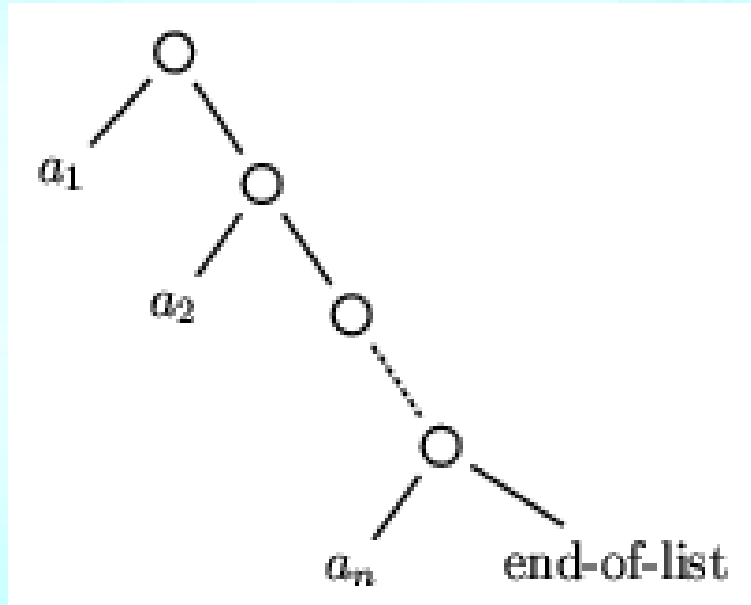


Figura 24: Uma lista em Prolog



Back

Close



A Sintaxe das Listas

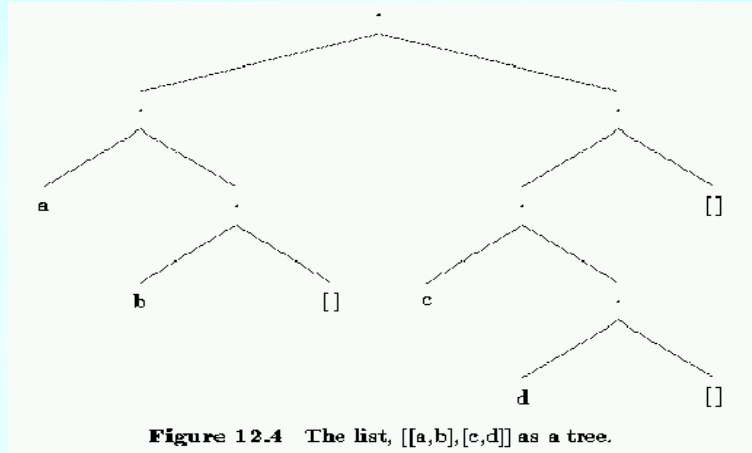


Figura 25: Lista1 ilustrada

➡ Após as definições e exemplos volte a esta figura



Back

Close



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 functor¹ (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.



114/176



Back

Close

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.

```

1  [ a, b, c, d ] == X
2  [ X | b, c, d ] == [ a, b, c, d ]
3  [ a | b, c, d ] == [ a, b, c, d ]
4  [ a, b | c, d ] == [ a, b, c, d ]
5  [ a, b, c | d ] == [ a, b, c, d ]
6  [ a, b, c, d | [] ] == [ a, b, c, d ]
7  [] == X
8  [ [ a | b, c, d ] ] == [ [ a, b, c, d ] ]
9  [ a | b, c, [ d ] ] == [ a, b, c, [ d ] ]
10 [ _ | b, c, [ d ] ] == [ a, b, c, [ d ] ]
11 [ a | Y ] == [ a, b, c, d ]
12 [ a | _ ] == [ a, b, c, d ]
13 [ a, b | c, d ] == [ X, Y | Z ]

```



115/176



Back

Close

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

```
1 [ a , b | [ c , d ] ] \== [ a , b , c , d ]
2 [ [ a , b , c , d ] ] \== [ a , b , c , d ]
3 [ a , b , [ c ] , d , e ] \== [ a , b , c , d , e ]
4 [ [ [ a ] | b , c , d ] ] \== [ [ a , b , c , d ] ]
```

Enfim, os tipos de casamentos de objetos de uma lista, seguem 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, reescrevendo em Prolog tal definição é dada por:

```
#1 if_uma_lista( [ ] ).          /* 1a. premissa */
#2 if_uma_lista( [X | T ] ) :- if_uma_lista( T ). /* 2a.
?- if_lista( [a,b,c] ).
yes
```



Um “*mapa de memŁjria aproximado*” ŁŁj dado por:

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

Basicamente, quase todas operaŁŁjŁŁjes com listas possuem regras anŁŁjlogas a definiŁŁjŁŁjo acima. O exemplo anterior serve apenas para identificar que o objeto: [a,b,c,d], ŁŁj uma lista.



118/176



Back

Close

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 comprimento([ ], 0).  
#2 comprimento([X | T], N):- comprimento(T, N1), N is N1+1.  
  
?- comprimento([a, b, c, d], X).  
X = 4
```

Um “*mapa de memória aproximado*” é dado por:





120/176

	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	—	—	—	$\nwarrow 0$



Back

Close



Concatenar ou união de duas listas: Em inglês este predicado² 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).
```

```
0 ‘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	$L \equiv [X \mid L3]$
<code>uniao([a,c,e],[b,d],L)</code>	#2	a	[c,e]	[b,d]	<i>[c,e,b,d]</i>	<i>[a,c,e,b,d]</i>
<code>uniao([c,e],[b,d],L)</code>	#2	c	[e]	[b,d]	<i>[e,b,d]</i>	\nwarrow <i>[c,e,b,d]</i>
<code>uniao([e],[b,d],L)</code>	#2	e	[]	[b,d]	<i>[b,d]</i>	\nwarrow <i>[e,b,d]</i>
<code>uniao([], [b,d],L)</code>	#1	–	–	[b,d]	–	\nwarrow <i>[b,d]</i>

²A 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 “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]	-



122/176



Back

Close



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



Back

Close

Verifica se um dado objeto pertence a 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]  
.....
```

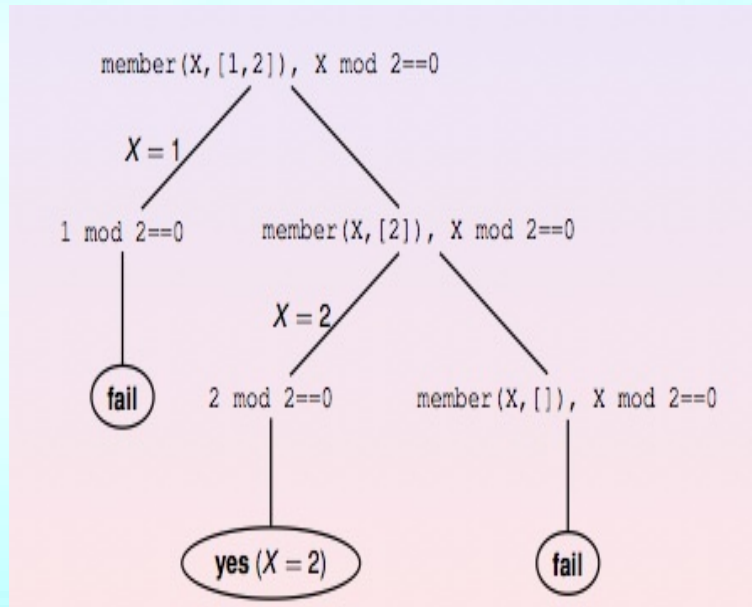


Figura 26: Um exemplo com member



Reflita sobre a variedade de usos deste predicado



126/176



Back

Close

Adiciona um objeto em uma lista: Neste exemplo, um objeto é adicionado a lista sem repetição caso este 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]).
```



127/176



Back

Close



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, fornece 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 (primitivo) 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 ).
```



130/176



Back

Close

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).
```



131/176



Back

Close

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.



132/176

```
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.



Back

Close

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 */
```



133/176



Back

Close

```
testa(27) :- !, fail.          /* Escape */  
testa( _ ) :- true.
```

Hilf; outros casos com o aterramento depois da regra geral.



134/176



Back

Close

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

```
1 del_X_all(X, [], []).
2 del_X_all(X, [X|L], L1) :- del_X_all(X,L,L1).
3 del_X_all(X, [Y|L1], [Y|L2]) :- del_X_all(X,L1,L2).
4
5
6 ?- del_X_all(3, [3,4,5,3,3,7,3],X).
7
8 X = [4, 5, 7]
9
10 Yes
11 ?- del_X_all(8, [3,4,5,3,3,7,3],X).
12
13 X = [3, 4, 5, 3, 3, 7, 3]
14
15 Yes
16 ?- del_X_all(3, [3],X).
17
18 X = []
19
20 Yes
21 ?- del_X_all(3, [],X).
22
```





136/176

```
23 X = []
24
25 Yes
26 ?- del_X_all(X, [3,4],Y).
27
28 X = 3
29 Y = [4] ;
30
31 X = 4
32 Y = [3] ;
33
34 X = _G189
35 Y = [3, 4] ;
36
37 No
38 ?-
39 ?- del_X_all(X, [3,4],[3]).
40
41 X = 4
42
43 Yes
```

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



Back

Close

funcionalidade.



137/176



Back

Close



Permutação: Alguns predicados são difíceis em qualquer linguagem de programação. Um destes é a permutação a qual é útil vários problemas. O predicado `exclui_1a` exclua 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]
```



20 [5 , 9 , 7]

21 [9 , 5 , 7]

22 [9 , 7 , 5]

23

24 No

25 ?—



139/176



Back

Close

Particípio:



140/176



Back

Close



Problemas de Buscas em IA

- Faça muitos exercícios sobre listas e funtores
- Combinando os funtores 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 site.
- Retrata exatamente/precisamente os problemas de buscas em geral.



Niŕcleo Miŕgico

➔ Reflita sobre este cıŕdigo, hiŕ muito conhecimento embutido.



142/176

```
1 solve(P) :-
2     start(Start),
3     search(Start,[Start],Q),
4     reverse(Q,P).
5
6 search(S,P,P) :- goal(S), !.           /* done */
7 search(S,Visited,P) :-
8     next_state(S,Nxt),                 /* generate next state */
9     safe_state(Nxt),                   /* check safety */
10    no_loop(Nxt,Visited),               /* check for loop */
11    search(Nxt,[Nxt|Visited],P).        /* continue searching... */
12
13 no_loop(Nxt,Visited) :-
14     \+member(Nxt,Visited).
```



Back

Close

Continuando com o *NiŁjcleo MiŁjgico*



143/176

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

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



Back

Close

Resumindo esta Idéjia em uma Figura

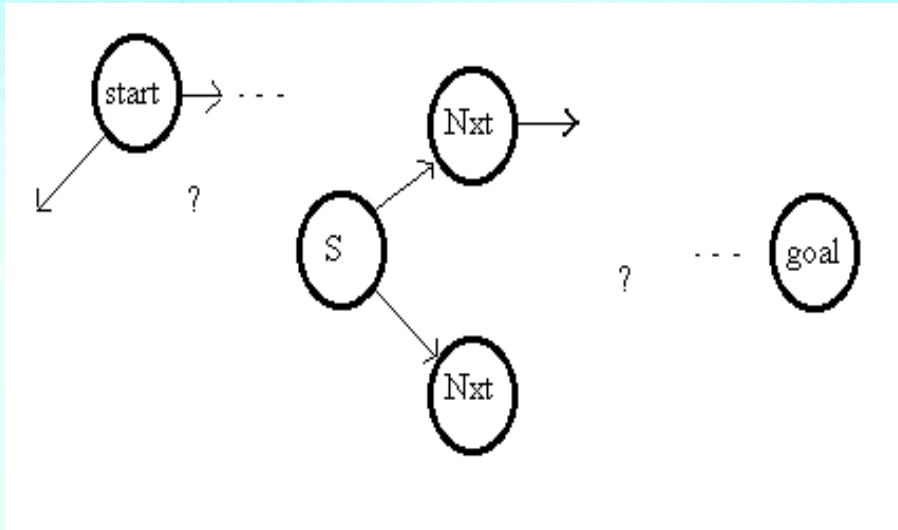


Figura 27: Níveis iniciais e finais do problema

➡ Volte aos capítulos e reflita mais uma vez!



Reusando o Conhecimento de Listas e Functores



145/176

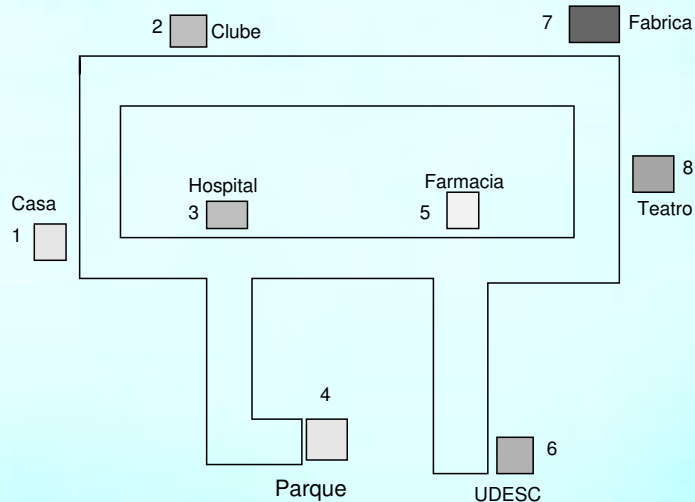


Figura 28: Um mapa – grafo cli£ssico



Resolvendo com Busca em Profundidade

➔ Acompanhe as explicações dos códigos em sala de aula!

```
1  /*
2  BUSCA em PROFUNDIDADE – MAPA da CIDADE
3  */
4
5  inicio(Sol_invertida) :-
6      no_origem(No_inicial),
7      busca(No_inicial, [No_inicial], Solucao),
8      reverse(Solucao, Sol_invertida),
9      write(Sol_invertida),
10     no_destino(Y), nl,
11     write('Onde o no de origem do mapa era: '), write(No_inicial) nl,
12     write('E o no de destino do mapa era: '), write(Y).
13
14 inicio('Nao ha mais solucoes'):- !.
15     /* para terminar com yes */
16     /******
17
18     /* Nós iniciais e finais do problema */
19     no_origem(casa).
20     no_destino(teatro).
```



Back

Close



```
21 /*****
22  *  Representação de um MAPA QUALQUER  *
23  *****/
24 /* MODELAGEM do Problema ==> parte mais difícil */
25 a(casa , hospital).
26 a(casa , clube ).
27 a(hospital , parque ).
28 a(hospital , farmacia ).
29 a(farmacia , udesc ).
30 a(teatro , farmacia ).
31 a(fabrica , clube ).
32 a(fabrica , teatro ).
33
34
35 /* para não ter que reescrever que:
36    a(fabrica , teatro) \== a(teatro , fabrica ).
37 */
38
39 conexao(X,Y) :- a(X,Y).
40 conexao(X,Y) :- a(Y,X).
41
42 /*****
43
44 %% cond. de parada .... o nó corrente é o nó desejado = nó final
45 busca( No_final , [ No_final | Caminho] , [No_final | Caminho]):-
46     no_destino(No_final).
47 %% termina a busca ...
48
```





148/176

```
49 busca(Noh, [Noh|Caminho], Solucao):-
50     conexao(Noh, Novo_nodo), /* proximo no = estado */
51     not( member(Novo_nodo,[Noh|Caminho])),
52     /* verifica se não foi visitado */
53     busca( Novo_nodo, [Novo_nodo, Noh | Caminho ], Solucao).
54
55 /******
56 /*
57 ?- inicio(X).
58 [casa, hospital, farmacia, teatro]
59 Onde o no de origem do mapa era: casa
60 E o no de destino do mapa era: teatro
61
62 X = [casa, hospital, farmacia, teatro] ;
63 [casa, clube, fabrica, teatro]
64 Onde o no de origem do mapa era: casa
65 E o no de destino do mapa era: teatro
66
67 X = [casa, clube, fabrica, teatro] ;
68
69 X = 'Nao hah mais solucoes' ;
70
71 No
72 ?-
73 */
```

Listing 5: Grafo da Cidade



Back

Close



Resolvendo com Busca em Largura

➔ Acompanhe as explicações dos códigos em sala de aula!

```
1  /*
2  BUSCA EM LARGURA – PROBLEMA DO MAPA
3  */
4  /******
5   * Representação de um MAPA/GRAFO QUALQUER *
6   *****/
7  /* MODELAGEM do Problema ==> parte mais difícil */
8  a(casa , hospital).
9  a(casa , clube ).
10 a(hospital , parque).
11 a(hospital , farmacia ).
12 a(farmacia , udesc).
13 a(teatro , farmacia ).
14 a(fabrica , clube).
15 a(fabrica , teatro).
16
17 /*
18
19
20
```

casa

/ \

hospital clube



Back

Close



150/176

```

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
```





```
005  Tempo total: 0.0000000000  msec
006  X = [casa , hospital , parque] .
007
008  ?- todas_solucoes .
009
010
011  CAMINHO PERCORRIDO: [casa , hospital , parque]
012  Nó de origem do mapa era: : casa
013  Nó de destino do mapa era: parque
014  Total de estados: 3
015  Total de movimentos: 2
016  T1: 0.090000  T2: 0.090000  msec
017  Tempo total: 0.0000000000  msec
018
019  CAMINHO PERCORRIDO: [casa , clube , fabrica , teatro , farmacia , hospital , parque]
020  Nó de origem do mapa era: : casa
021  Nó de destino do mapa era: parque
022  Total de estados: 7
023  Total de movimentos: 6
024  T1: 0.090000  T2: 0.090000  msec
025  Tempo total: 0.0000000000  msec
026  RESUMO DAS SOLUCOES :
027  ==> [casa , hospital , parque]
028  ==> [casa , clube , fabrica , teatro , farmacia , hospital , parque]
029  ==> Nao hah mais solucoes
030  true .
031
032  ?-
```



Listing 6: Grafo da Cidade



154/176



Back

Close



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 *backtracking*. Exemplo é o `is`. Veja o que fazer para contornar, por exemplo:

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



- A cada prediç o constru do, teste! Trabalhe com o conceito de prototipa  o.



156/176



Back

Close



Predicados *Mi£jo-na-roda*

📖 Na console digite `?- help(nome_do_predicado)` para detalhes, nos seguintes predicados *mi£jo-na-roda*.

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



Back

Close

➡ 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 nospys faz uma depuração em um predicado em particular.

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



158/176



Back

Close



159/176

Predicados *Baixaria*

➤ if - then - else , atĩtu?

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

➤ for

➤ while



Back

Close



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

➔ O Prolog gera executíveis com velocidades compatíveis a linguagem C++, Delphi, e outras. Contudo, o SWI-Prolog pelo fato de ser um “*freeware*” (talvez), gera executíveis que traz consigo o seu interpretador. Ou seja, um executível 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 *ECLⁱPS^e* é mantido pela CISCO, explorado comercialmente por esta, mas é gratuito e livre.

[Back](#)[Close](#)

➔ A seguir irei mostrar 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 interprete;  
%% ao final
```



161/176



Back

Close

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 começará a processar. Neste exemplo, o predicado chama-se `inicio`.

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



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 executível foi gerado chamado “*saida.exe*”.



163/176



Back

Close



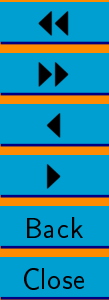
➡ Estes dois scripts s~ao equivalentes ao seguinte procedimento dentro do ambiente interpretado. Logo, este vai funcionar para outros SO's:

```
?- 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 executível chamado de “*nova_saida.exe*”. Leia com atenção o *help* deste predicado.

A interface com linguagens como C, Java e outros ã relativamente fãcil. Contudo, neste momento serã omitido.



Ainda sobre Gerar Executíveis

➔ 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:

```
1  /*
2  $ swipl --goal=main -t halt -o saida2 -c fatorial.pl
3  % fatorial.pl compiled 0.00 sec, 1,164 bytes
4  $ ./saida2
5  6
6  120
7  5040
8  $ swipl --stand_alone=true --goal=main -t halt -o saida1 -c fatorial.pl
9  % fatorial.pl compiled 0.00 sec, 1,164 bytes
10 $ ./saida1
11 6
12 120
13 5040
14 $ ls -al saida*
15 -rwxrwxrwx 1 root root 907617 Set  1 00:27 saida1
16 -rwxrwxrwx 1 root root 122822 Set  1 00:26 saida2
17 */
18
19 %%% swipl -stand_alone=true -g main -t halt -o saida.exe -c fatorial.pl
20 %%% nesta ordem...
```



165/176



Back

Close



166/176

```
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 , Partial ),
30                   Fat is ( X * Partial ).
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



Back

Close



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  
?-
```



Back

Close

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**.



168/176



Back

Close



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
```

```
?-
```



Back

Close

➡ 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!



170/176



Back

Close



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, está 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/>



174/176



Back

Close



Alguns Bons Livros

- Michael A. Convigton, Donald Nute, Andri  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;
- H  um t tulo de um livro de IA, que   “aproximadamente”  : “*Solving Complex Problems with Artificial Intelligence*”, cuja linguagem utilizada nas solu  es   o Prolog.



Sugestões

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



176/176



Back

Close