Tutorial de Prolog

Paulo Victor de Aguiar
Luciani Regina da Costa Dantas
Claudio Cesar de Sá e Outros
Universidade do Estado de Santa Catarina – UDESC
Departamento de Ciência da Computação – DCC
Joinville – SC

10 de novembro de 2015

Indice

Motivação

Relembrando que:

- `Escutar, ver, e falar, trazem apenas lembranças.`
- Apenas praticando, o verdadeiro aprendizado ocorre.
- Dar uma consequência prática ao curso de LMA e mostrar uma ferramenta que usa provadores automáticos de teoremas lógicos.

Histórico

A Linguagem Prolog:

- As bases na lógica clássica
- Precisamente a Lógica de Primeira-Ordem (LPO)
- Os fundamentos da LPO, seguem numa notação em cláusulas, conhecida como **forma clausal**, precisamente, uma notação restrita das **cláusulas de Horn**
- Início da década de 70 sua primeira implementação
- Continua evoluindo, e é bem ativa até hoje
- Foi uma das primeiras ferramentas da Inteligência Artificial (IA)

Características do Prolog

- Sintaxe reduzida, poucos elementos
- Boa para fazer protótipos
- Desempenho é seu ponto negativo
- Contudo, muitas aplicações reais estão em Prolog há muitos anos
- Há muitas empresas que vivem de Prolog
- Há países com forte tradições em Prolog: Japão, UK, França, etc
- Ainda muito usado na academia

Requisitos

Retomando:

- Algum compilador e/ou interpretador Prolog instalado
- Prolog sugerido: `http://www.swi-prolog.org/`
- Ou ainda, podes fazer tudo via internet.
- Em algum navegador use:
  `http://www.tutorialspoint.com/execute_Prolog_online.php`
- Um editor de programas ... um **bom** de preferência!

Usando o Prolog

Primeiros Passos:

- Usando o Prolog em um terminal Linux, basta digitar o comando `swipl`, claro, com o SWI-PROLOG instalado
- Na console do swipl, digite o comando `consult(nome do arquivo)` para carregar o seu programa no interpretador Prolog
- URL: `http://http://www.swi-prolog.org/`

Ambiente de Programação

prolog : swipl - Konsole

Arquivo Editar Exibir Favoritos Configurações Ajuda

```
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Versio
Copyright (c) 1990-2013 University of Amsterdam, VU Am
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is
and you are welcome to redistribute it under certain c
Please visit http://www.swi-prolog.org for details.

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

?- emacs.
true.

?- % /home/ccs/Dropbox/cursos/lma/slides_prolog/prolog
d 0.00 sec, 11 clauses
?- x.
carlos,patricia,engenheiro
luis,lucia,medico
paulo,maria,advogado
true ;
carlos,patricia,engenheiro
luis,maria,medico
paulo,lucia,advogado
true ;
carlos,lucia,engenheiro
luis,patricia,medico
paulo,maria,advogado
true .
```

prolog : swipl

casais.pl [modified]

File Edit Browse Compile Prolog Pce Help

casais.pl [modified]

```
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.
```

Colourising buffer ... done, 0,00 seconds, 90 fragments Line: 35

Elementos Básicos

Elementos Básicos

Elementos:

- **Termo**: qualquer dado em Prolog é chamado termo. Tanto átomos quanto variáveis são termos.

- **Átomo**: é um nome de propósito geral ou de uma relação e é representado por uma sequência de letras e alguns caracteres como por exemplo: pai, gordo, amigo, 'João'.
  Eles começam sempre com letra minúscula ou entre aspas simples.

- **Variável**: similar ao átomo, porém começa com letra maiúscula (Ex: X, Y, Paulo, VICTOR). elas são como uma incógnita, ou seja, valor é desconhecido até ser descoberto.

Elementos Básicos

Números:

- Número inteiro: qualquer numero que não tenha o ponto flutuante.
  Ex: 1, 2, 3, 3960875698740

- Número float: números com o ponto flutuante e pelo menos uma casa decimal.
  Ex: 1.5 (correto), 2. (incorreto)

Elementos Básicos

Operadores de Comparação:

Operador	Símbolo	Exemplo
E	,	A , B
OU	;	A ; B
Negação	\+	\+A
Igualdade	==	A == B
Desigualdade	\==	A \== B

Exercícios

Classifique os termos abaixo como Variável, Átomo e Número:

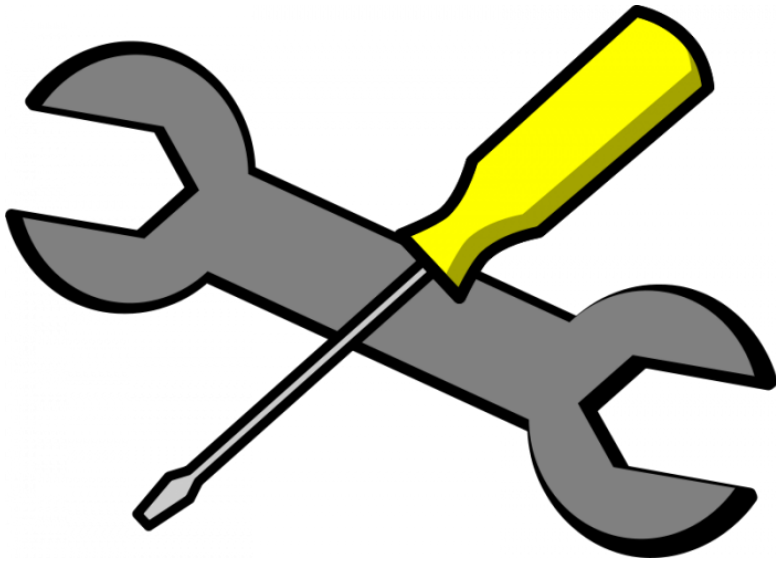
1. joão
2. 9
3. X
4. a
5. 0.5
6. 0.0.
7. '2'
8. Pedro

Exercícios

Classifique os termos abaixo como Variável, Átomo e Número:

1. joão - **Átomo**
2. 9 - **Número Inteiro**
3. X - **Variável**
4. a - **Átomo**
5. 0.5 - **Número Float**
6. 0.0. - **Inválido**
7. '2' - **Átomo**
8. Pedro - **Variável**

Estrutura de Programa



Estrutura de Programa

Basicamente:

Um programa Prolog constitui-se de uma coleção de **fatos** (base de dados) e **regras** (relações ou predicados lógicos), esses itens descrevem um problema e sua descrição é avaliada por um interpretador.

Fatos

Conceito:

Os fatos de Prolog permitem a declaração de átomos e termos de um domínio satisfazem os predicados (relações lógicas).

Por exemplo, pode-se definir o predicado `homem(X)` e utilizar este para definir quais elementos do domínio possuem tal predicado, nesse caso X é homem.

Exemplos:

```
1 homem(pedro).
2 homem(joao).
3 mulher(maria).
4 mulher(teresa).
```

Fatos

Sintaxe:

- Os nomes dos predicados e dos objetos sempre devem começar com letra minúscula.
- Os objetos são escritos dentro de parênteses.
- Todo fato é terminado com um ponto final.
- A ordem dos objetos é importante:

```
pai(carlos, pedro).
é diferente de:
pai(pedro, carlos).
```

Questões

Seja o código:

```
pai(carlos, pedro).
pai(carlos, paulo).
```

Uma questão (*query*) é dada por:

```
?- pai(carlos, X).
X = pedro ;
X = paulo ;
false
```

Regras

Conceito:

- As regras em Prolog são descrições de predicados condicionais.
- Por exemplo, pode-se definir o predicado `pai(X)`, que significa `X é pai`, utilizando uma regra do tipo:

```
pai(X) :- homem(X).
```

- Assim, para `X ser pai`, se somente se, `X for homem`!

Regras

Exemplo:

- A regra significa que X é pai se X for homem. O usuário interage com o programa utilizando consultas (queries). Por exemplo, sejam os fatos:

```
1 homem(pedro).
2 homem(joao).
3 mulher(maria).
4 mulher(teresa).
5 pai(X) :- homem(X).
```

- O usuário pode realizar a consulta `pai(X)`:

Execução:

```
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 7.2.1)
Copyright (c) 1990-2015 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

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

?- consult('pessoas.pl').
true.

?- pai(X).
X = pedro ;
X = joao.

?-
```

Fluxo da Máquina *Prologuiana*

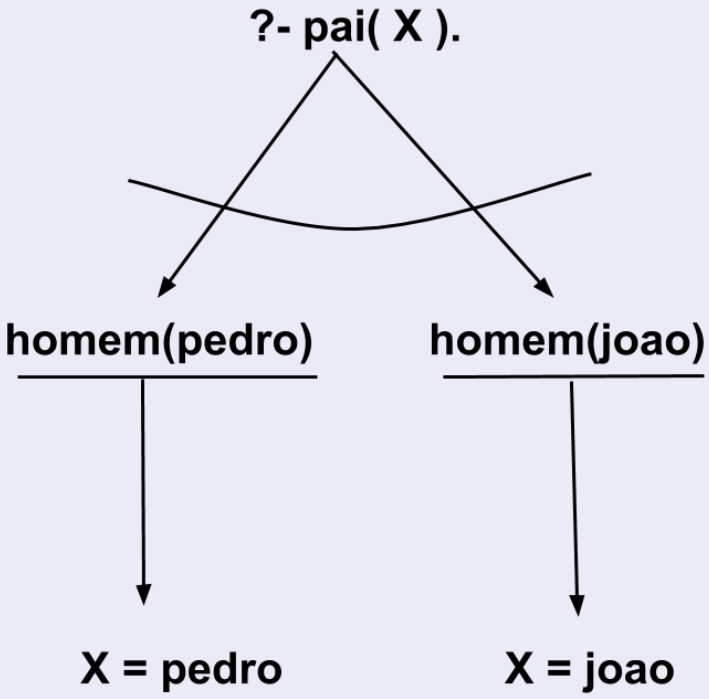


Figura: Fluxo básico de um predicado em Prolog.

Exemplo de Regra

```
dados(X,Y,Z) :-
        nome(X),
        idade(Y),
        altura(Z).

nome(marcia).
nome(antonio).
nome(fabricia).

idade(18).
idade(20).

altura(1.71).
altura(1.85).

pessoas :-
    dados(X,Y,Z),
    format('\n X: ~w  \t Y: ~d \t  Z: ~f ',[X,Y,Z]),
    fail.

pessoas:-
     format('\n Nao hah mais dados !!!!!\n', []).
/* ****************************************** */
```

Funcionamento de uma Regra

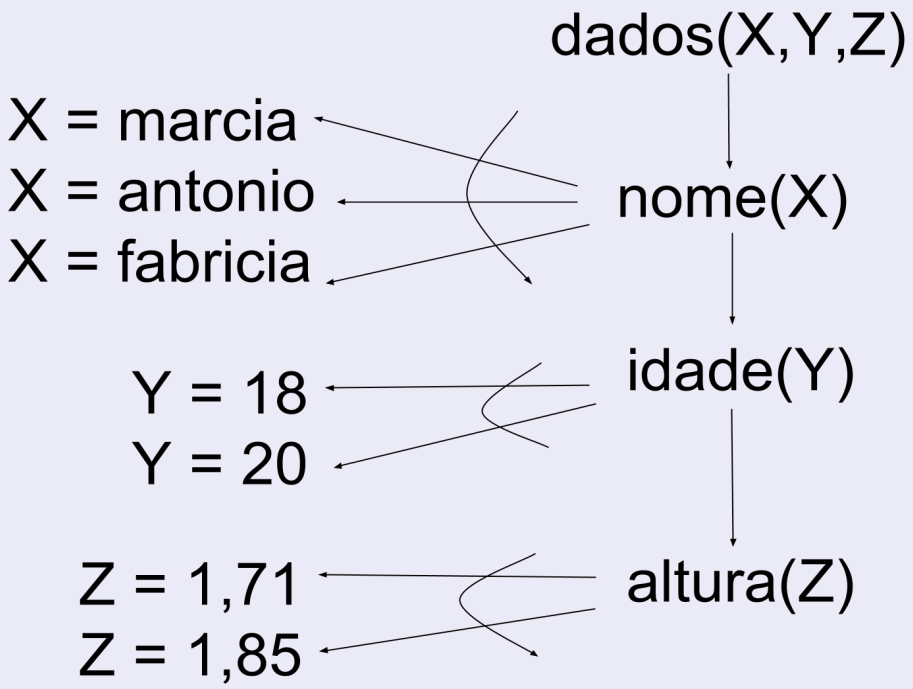


Figura: Fluxo básico de uma regra em Prolog

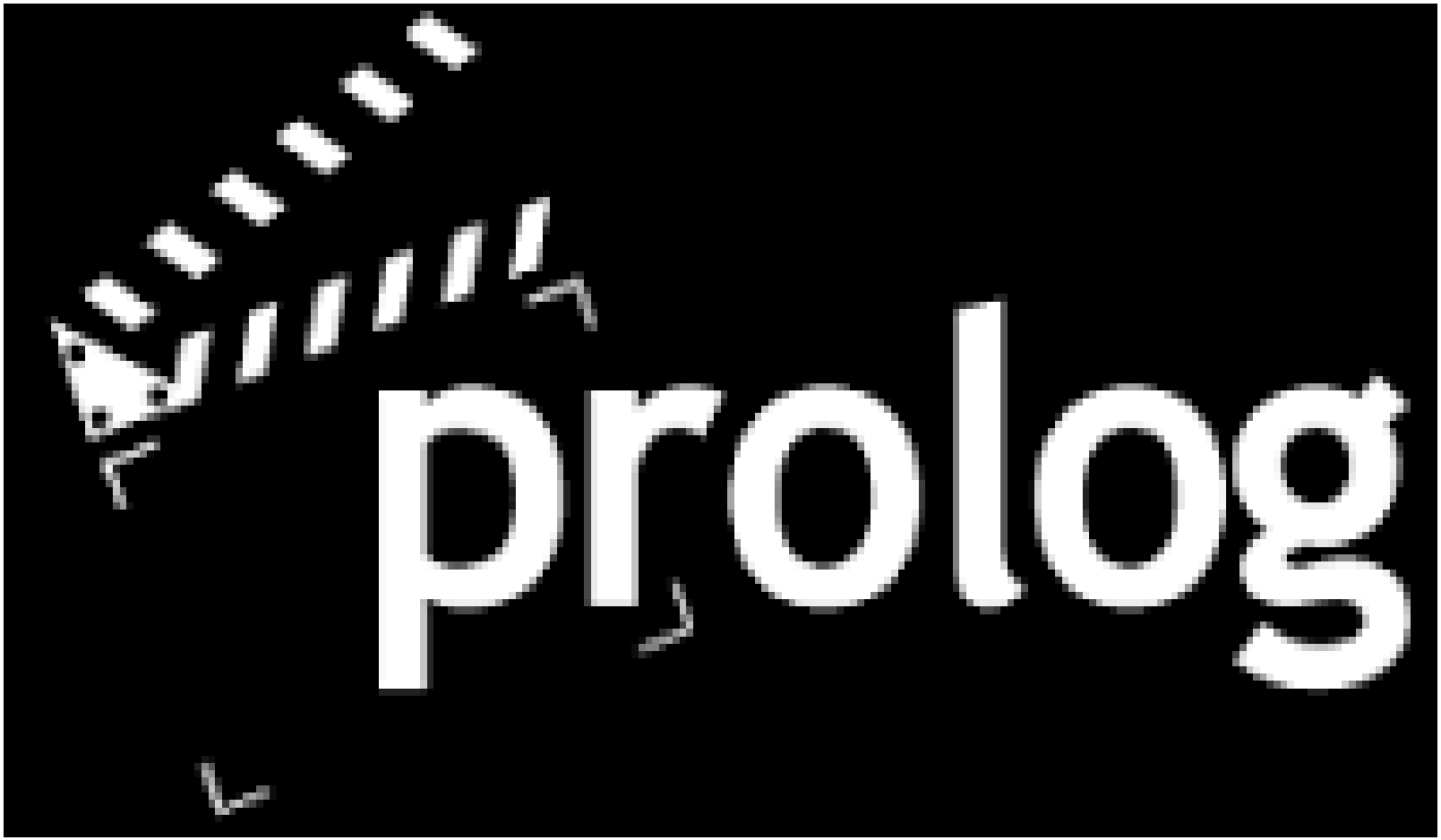
Execução:

```
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 7.2.1)
Copyright (c) 1990-2015 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

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

?- consult('exemplo-01.pl').
true.

?- pessoas.

 X: antonio      Y: 18    Z: 1.710000
 X: antonio      Y: 18    Z: 1.850000
 X: antonio      Y: 20    Z: 1.710000
 X: antonio      Y: 20    Z: 1.850000
 X: maria        Y: 18    Z: 1.710000
 X: maria        Y: 18    Z: 1.850000
 X: maria        Y: 20    Z: 1.710000
 X: maria        Y: 20    Z: 1.850000
 X: fabricia     Y: 18    Z: 1.710000
 X: fabricia     Y: 18    Z: 1.850000
 X: fabricia     Y: 20    Z: 1.710000
 X: fabricia     Y: 20    Z: 1.850000
 Nao hah mais dados !!!!!
true.

?-
```

Resumo até o momento:

- Precisamos de um editor de programas e um compilador
- 3 elementos compõem a sintaxe do Prolog: **fatos**, **regras** e **questões**
- **Fatos**: expressam dados conhecidos ou verdades incondicionais
- **Regras**: expressam conhecimentos genéricos, como regras gerais dos fatos.
- Um conhecimento é uma uma verdade condicional, isto é, uma regra ou fato que precisa ser provado
- **Questões**: buscam fazer a demonstrações das verdades de regras e fatos
- Pratique!
- Voce já pode criar e resolver muitos problemas difíceis e interessantes
- Exemplos: `https://github.com/claudiosa/prolog`

Pausa



Functores ou Funções Lógicas

Nesta vídeo-aula

- Conceito sobre Functores
- Dois exemplos
- Pré-requisito: **fatos**, **questões** e **regras**
- Ao final: *prontos* para resolverem os problemas do sítio `http://rachacuca.com.br/`
- Todos os códigos aqui discutidos estão: `https://github.com/claudiosa/prolog`
- A apresentação: `http://www2.joinville.udesc.br/~coca/index.php/Main/LogicaMatematica`

Functores

Conceito:

- Functores são termos compostos utilizados para representar funções com termos combinados entre si.
- Assim, o functor é um termo da relação e internamente há outros termos e átomos.
- Uma estratégia de contornar o **V**erdadeiro ou **F**also da LPO e/ou do Prolog
- Exemplos:

1. `pai(pedro, joao)`: Pedro é o pai do João
2. mas `pai(pai(pedro), filho(joao))`: é muito diferente
3. Pois: $pai(pedro) \mapsto adao$ e $filho(joao) \mapsto abel$

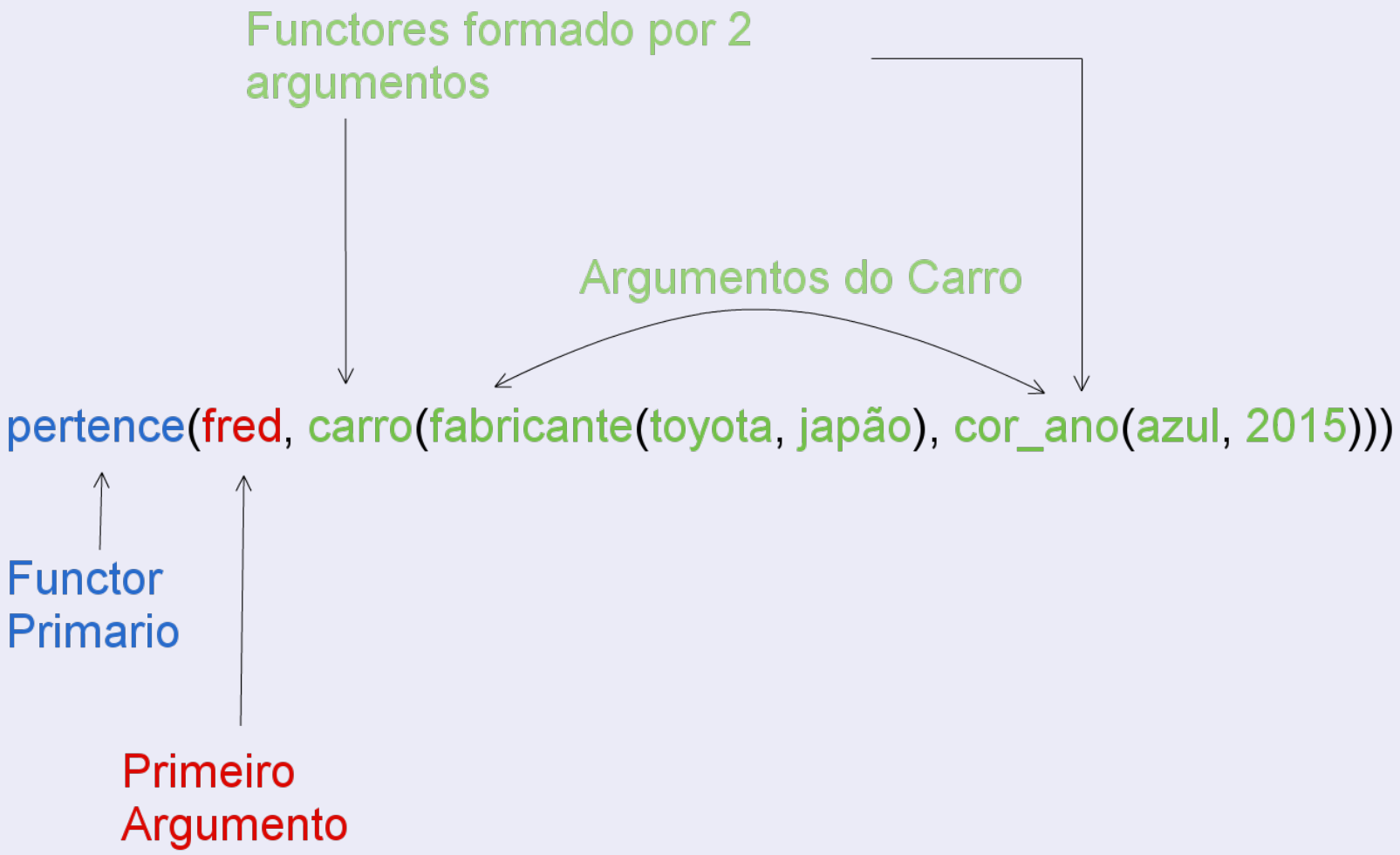


Figura: Um exemplo de functor – um termo composto.

Código:

```
carros :-
  pertence(X,Y,Z)

pertence(fred,
  carro(fabricante(toyota, japao)),
  ano_cor(2014, azul)).

pertence(romi,
  carro(fabricante(bmw, alemanha)),
  ano_cor(2015, vermelho)).

pertence(claudio,
  carro(fabricante(vw, brasil)),
  ano_cor(2012, prata)).
```

Execução:

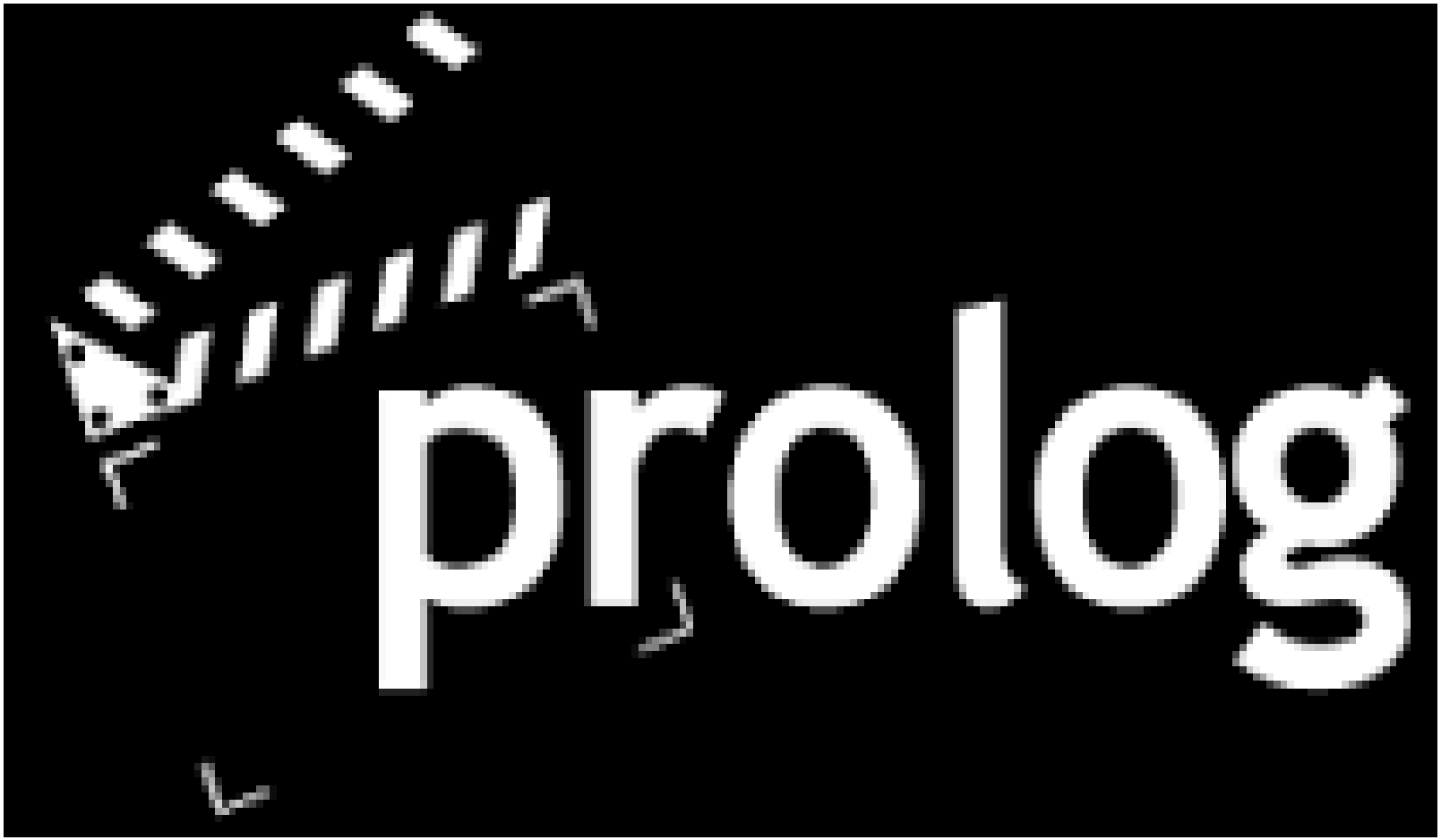
```
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 7.2.1)
Copyright (c) 1990-2015 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

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

?- consult('exemplo-02.pl').
true.

?- carros.

 X: fred         Y: carro(fabricante(toyota,japao))       Z: ano_cor(2014,azul)
 X: romi         Y: carro(fabricante(bmw,alemanha))       Z: ano_cor(2015,vermelho)
 X: claudio      Y: carro(fabricante(vw,brasil))          Z: ano_cor(2012,prata)
 Nao hah mais dados !!!!!
true.

?-
```

Execução com Condicional:

```
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 7.2.1)
Copyright (c) 1990-2015 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

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

?- consult('exemplo-02.pl').
true.

?- pertence(X,Y,ano_cor(A,C)) , A > 2014.
X = romi,
Y = carro(fabricante(bmw, alemanha)),
A = 2015,
C = vermelho
```

Figura: A pergunta foi feita com **uma condição**: `A > 2014`

Pausa



Exercícios

Identifique o que os fatos representam:

1. `animal(gato)`
2. `veiculo(moto)`
3. `cor(preta)`
4. `mãe(maria,bia)`
5. `móvel(Sofá)`

Exercícios

Identifique o que os fatos representam:

1. `animal(gato)` - **gato é um animal**
2. `veiculo(moto)` - **moto é um veiculo**
3. `cor(preta)` - **preto é uma cor**
4. `mãe(maria,bia)` - **maria é mãe da bia**
5. `móvel(Sofá)` - CUIDADO **não significa que sofá é um movel porque Sofá é uma variável**

Recursividade

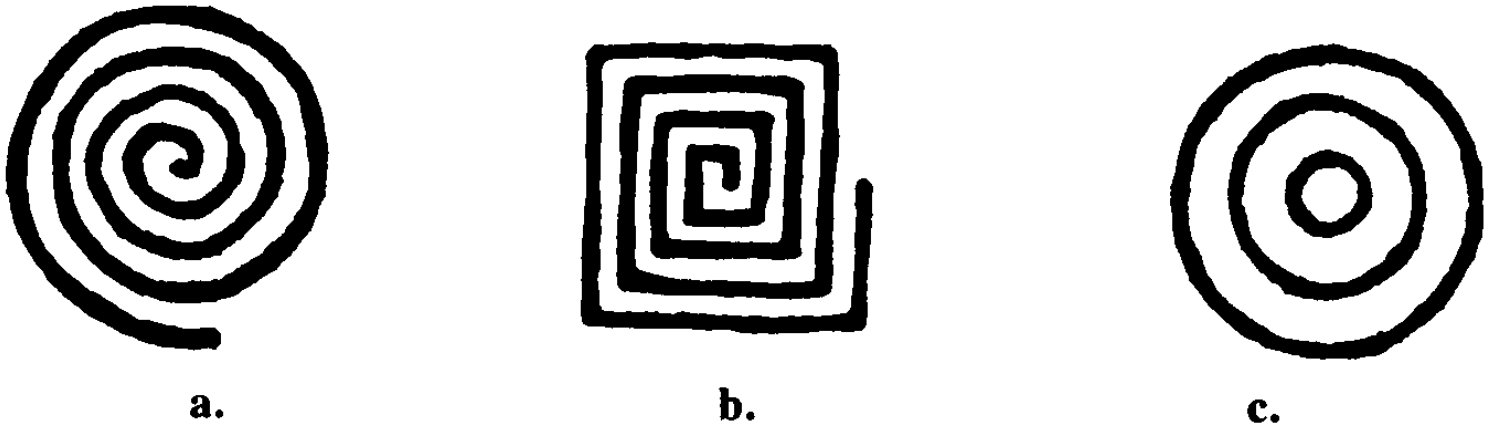


Figura: Recursão ilustrada

Conceito:

- A recursão é um conceito importante da linguagem Prolog.
- Esta permite expressar problemas complexos de uma maneira simples.
- Um exemplo é a relação `descendente(X,Y)` "*Y é um descendente de X*".
- Esta definição utiliza a relação `genitor` que é uma descendência direta, ou seja, quando `X` é genitor de `Y`, é representada com a seguinte regra:

```
descendente(X,Z) :- genitor(X,Z).
```

Outros Casos:

- Outros casos de descendência, que não uma descendência direta, poderiam ser utilizadas seguintes regras:
  1. `descendente(X,Z) :- genitor(X,Y) , genitor(Y,Z).`
  2. `descendente(X,Z) :- ancestral(Z,X).`
  3. Claro: o conceito de `ancestral` é o oposto de `descendente`

- Usando recursão é possível obter uma solução simples. Para isso, é necessário definir a seguinte afirmação: `X` é um descendente de Z se existe um Y, tal que, X seja genitor de Y e Y seja um descendente de Z, a seguinte regra descreve isso:
  `descendente(X,Z) :- genitor(X,Y) , descendente(Y,Z).`

Problema: Árvore Genealógica

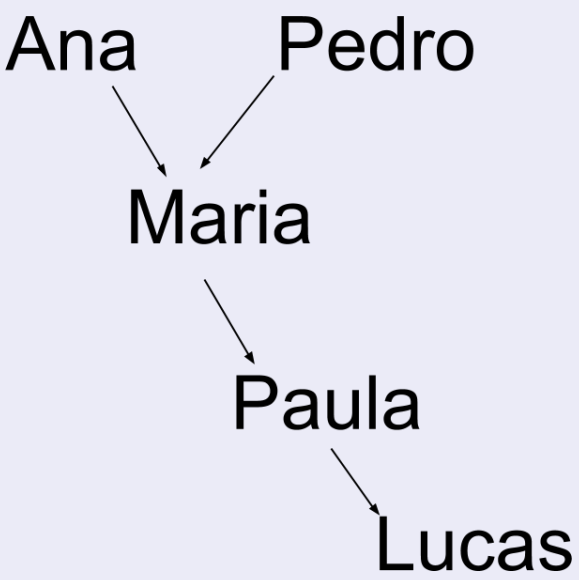


Figura: Árvore Genealógica

Programa:

- O programa completo que descreve as relações familiares discutidas nesta seção pode ser observado a seguir:

```
1  ancestral(ana,maria).
2  ancestral(pedro,maria).
3  ancestral(maria,paula).
4  ancestral(paula,lucas).
5  mulher(ana).
6  mulher(maria).
7  mulher(paula).
8  homem(pedro).
9  homem(lucas).
10 prole(Y,X) :- ancestral(X,Y).
11 mae(X,Y) :- ancestral(X,Y), mulher(X).
12 avos(X,Z) :- ancestral(X,Y), ancestral(Y,Z).
13 descendente(X,Z) :- ancestral(X,Z).
14 descendente(X,Z) :- ancestral(X,Y), descendente(Y,Z).
```

Testando Programa:

```
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 7.2.1)
Copyright (c) 1990-2015 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

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

?- consult('arvore.pl').
true.

?- descendente(ana,X).
X = maria ;
X = paula ;
X = lucas ;
false.

?-
```

Resultado:

- A consulta a seguir um exemplo sobre a relação de descendência, nela exibe todos os descendentes de Ana:

```
?- descendente (ana,X).
X = maria;
X = paula;
X = lucas;
false.
```

Soma Recursiva

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

Este problema é reformulado sob uma visão matemática, mais especificamente, pela *indução finita* dada por:

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

Soma Recursiva

O que é um fato verdadeiro pois:

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

Como o procedimento é recursivo, é necessário encontrar a definição da "**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).

Soma Recursiva

```
s(1,1) :- true.                    /* regra #1 */
s(N, S) :- N > 1,            /* regra #2 */
        Aux is (N-1),
        format('\n N: ~w \t AUX: ~w
                \t PARC: ~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 PARC: ~w \t S: ~w',
                [N, Aux, Parcial, S]).
```

Execução s(7,7):

```
?- s(7,7).

 N: 7    AUX: 6          PARC: _G1706    S: 7
 N: 6    AUX: 5          PARC: _G1718    S: _G1706
 N: 5    AUX: 4          PARC: _G1730    S: _G1718
 N: 4    AUX: 3          PARC: _G1742    S: _G1730
 N: 3    AUX: 2          PARC: _G1754    S: _G1742
 N: 2    AUX: 1          PARC: _G1766    S: _G1754
 ==> Apos o casamento da  REGRA #1:
 N: 2    AUX: 1          PARC: 1         S: 3
 ==> Apos o casamento da  REGRA #1:
 N: 3    AUX: 2          PARC: 3         S: 6
 ==> Apos o casamento da  REGRA #1:
 N: 4    AUX: 3          PARC: 6         S: 10
 ==> Apos o casamento da  REGRA #1:
 N: 5    AUX: 4          PARC: 10        S: 15
 ==> Apos o casamento da  REGRA #1:
 N: 6    AUX: 5          PARC: 15        S: 21
 ==> Apos o casamento da  REGRA #1:
false.

?-
```

Execução s(5,x):

```
?- consult('soma.pl').
true.

?- s(5,X).

 N: 5     AUX: 4            PARC: _G1718     S: _G1647
 N: 4     AUX: 3            PARC: _G1730     S: _G1718
 N: 3     AUX: 2            PARC: _G1742     S: _G1730
 N: 2     AUX: 1            PARC: _G1754     S: _G1742
 ==> Apos o casamento da  REGRA #1:
 N: 2     AUX: 1            PARC: 1          S: 3
 ==> Apos o casamento da  REGRA #1:
 N: 3     AUX: 2            PARC: 3          S: 6
 ==> Apos o casamento da  REGRA #1:
 N: 4     AUX: 3            PARC: 6          S: 10
 ==> Apos o casamento da  REGRA #1:
 N: 5     AUX: 4            PARC: 10         S: 15
X = 15
```

Exercício:

- Crie fatos e uma regra e faça uma consulta no swi-prolog para ver o resultado.

Fatorial

Definição:

- 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:

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

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 ).
```

Exercício:

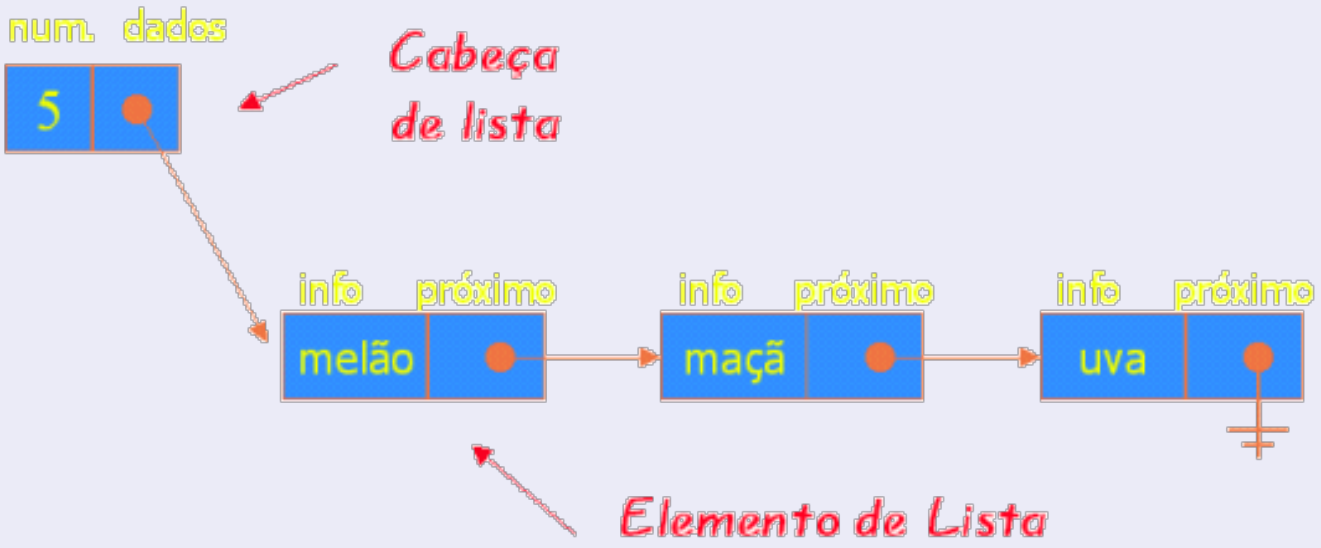
Complete a tabela abaixo, para descrição do fatorial:

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)	#	.....	.....	?..... →	↖.....

FALTA AINDA

1. Paulo: corrigi até aqui ....
2. Há outros exemplos de recursividade na apostila original, prontos, como o fatorial ou do ancestral... que está muito legal. Siga o modelo aplicado.
3. TODAS figuras devem ser em formato VETORIAL: pdf ou svg
4. png ... não é vetorial.... salve em pdf ou svg ou use um conversor como: `http://vectormagic.com/support/file_formats`
   Cuidar mais com a Língua Portuguesa .. há muitos erros no texto de acentuação Básica
5. A figura da próxima página é inadmissível
6. Não apague estas observações

Listas


num. dados
5
Cabeça de lista
info
próximo
melão
info
próximo
maçã
info
próximo
uva
Elemento de Lista

Listas

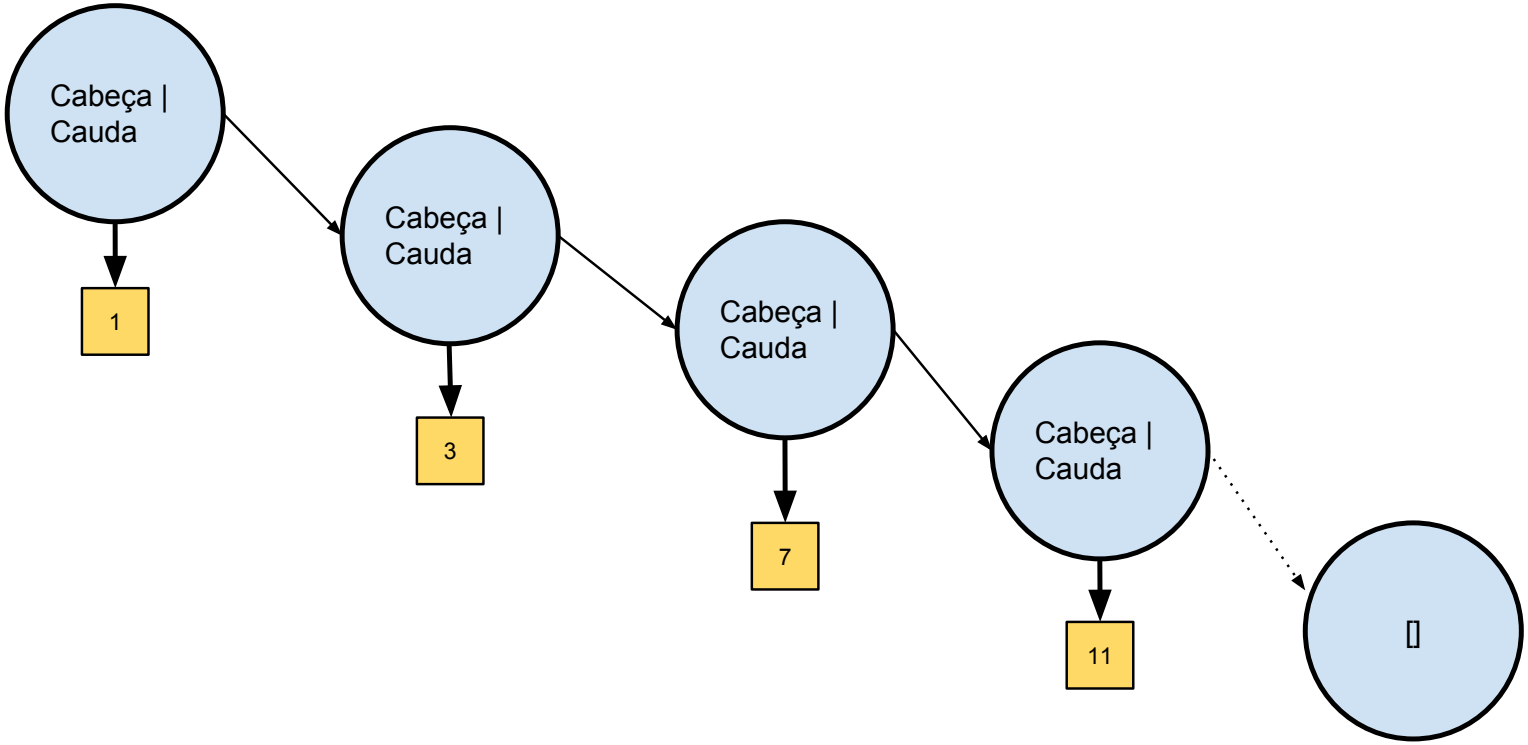


Figura: [1, 3, 7, 11]

Listas

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

Fluxo Operacional das Listas

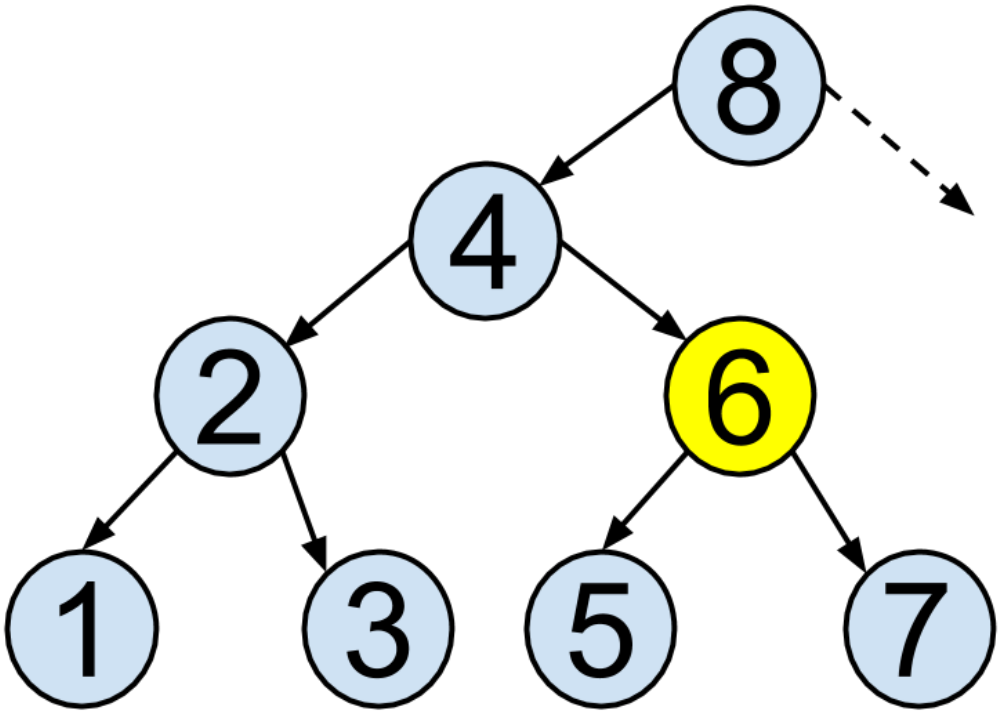


Figura: Cálculo Recursivo 2

Definições

Uma lista é:

- Uma estrutura de dados que representa uma coleção de objetos homogêneos;
- Uma sequência de objetos;
- Um tipo particular de functor [a] (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;

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

Exemplos:

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

```
letras( [ a, b, c, d ] ).
          ^
          |
          |
  cabeca 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 também é uma lista.

Exemplos:

Operador Pipe: define quem é a cabeça da cauda da lista. Ele é 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:

Exemplos de "casamentos": os exemplos abaixo definem como ocorrem os casamentos entre variáveis e listas. 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 ]
```

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, Reescrenvendo em Prolog tal definição é dada por:

```
1  eh_uma_lista( [ ] ).
2  eh_uma_lista( [X | T ] ) :-
3                  eh_uma_lista( T ).
4
5  ?- eh_lista( [a,b,c] ).
6  yes
```

Mapa de Memória

	Regra	X	T
eh_uma_lista([a,b,c])	#2	a	[b,c]
eh_uma_lista([b,c])	#2	b	[c]
eh_uma_lista([c])	#2	c	[]
eh_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 lista

As regras sobre listas são diversas e elegantes. Apenas exercitando há que se cria a destreza para resolve-las. 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:

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

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

Mapa de Memória

	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[a] é conhecido como "*append*", e em alguns Prolog's pode estar embutido como predicado nativo:

```
uniao([],X,X).
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
```

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

Mapa de Memória

	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]*

Dividir uma lista em duas outras listas:

Lista inicial "em [X,Y | L]", em uma lista

```
divide([], [], []).
divide([X], [], [X]).
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]
```

Mapa de Memória:

	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 ).
```

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 os seguintes resultados:

Consultas:

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

?- member(X,[4,5,3]).
X = 4 ;
X = 5 ;
X = 3 ;
false.

?- member(3,X).
X = [3|_G1266] ;
X = [_G1265, 3|_G1269] ;
X = [_G1265, _G1268, 3|_G1272] ;
X = [_G1265, _G1268, _G1271, 3|_G1275] ;
X = [_G1265, _G1268, _G1271, _G1274, 3|_G1278] ;
X = [_G1265, _G1268, _G1271, _G1274, _G1277, 3|_G1281] ;
X = [_G1265, _G1268, _G1271, _G1274, _G1277, _G1280, 3|_G1284] ;
```

Verifica se um dado objeto pertence há uma lista:

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

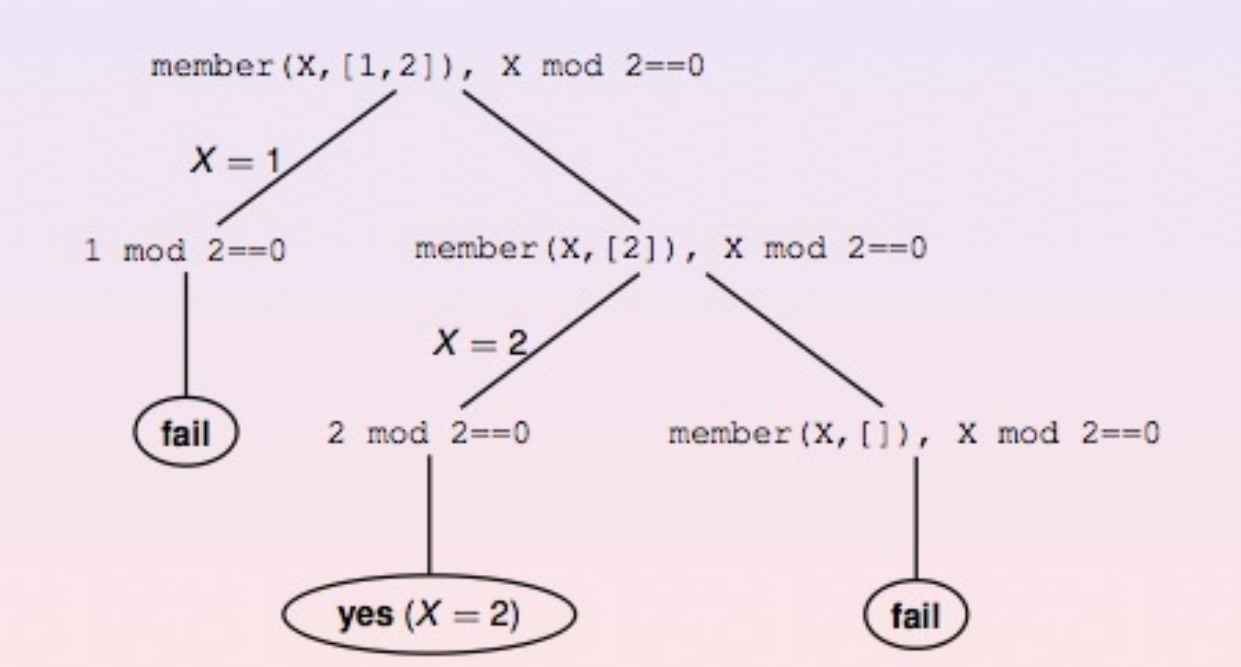


Figura: Um exemplo com member

Exercício:

Reflita sobre a variedade de usos deste predicado.

Adiciona um objeto em uma lista:

- Novamente, em alguns Prolog's, este predicado pode estar embutido, confira:
- Neste exemplo, um objeto é adicionado a lista sem repetição caso este jé 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 ) ,
5                    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]).
```

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.

```
1  subs(A,B) :- sub(A,B,A).
2  sub([],_,_).
3  sub([A|B] , [C|D] , Lcopia) :- A == C,
4                                 sub( B, D, Lcopia).
5  sub([A|_] , [C|D] , Lcopia) :- A \== C,
6                                 sub(Lcopia,D, Lcopia).
```

- Como exercício, faça este predicado utilizando dois append.

Verifica se uma lista está contida em outra lista:

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),
                      le_aux( Resto ).
```

Condição de Parada:

```
le_aux( [ ] ) :- !.
testa(13) :- !, fail.
testa(10) :- !, fail.
testa(27) :- !, fail.
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).
```

Consultas:

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

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

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

?- del_X_all(3, [],X).
X = [] .
```

Removendo um item da lista:

```
?- del_X_all(X , [3 ,4] , Y ).
X = 3,
Y = [4] ;
X = 4,
Y = [3] ;
Y = [3, 4].

?- del_X_all(X , [3 ,4] ,[3]).
X = 4 ;
```

Observação:

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 ocorrências de um termo na lista, enquanto o `del_X_all`, visto anteriormente, exclui todas ocorrências.

```
permutar([], []).
permutar([X|L], Lpermutada):-
        permutar(L, L1),
        exclui_1a(X, Lpermutada, L1).

exclui_1a(X, [X|L], L).
exclui_1a(X, [Y|L], [Y|L1]):-
                exclui_1a(X, L, L1).
init :- permutar([5,7,9],X), nl ,
                write(X), fail.
```

Permutação:

```
init.
[5, 7, 9]
[7, 5, 9]
[7, 9, 5]
[5, 9, 7]
[9, 5, 7]
[9, 7, 5]
No
```

Problemas Clássicos de Inteligência Artificial

- 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.

Resumindo esta idéia em uma figura:

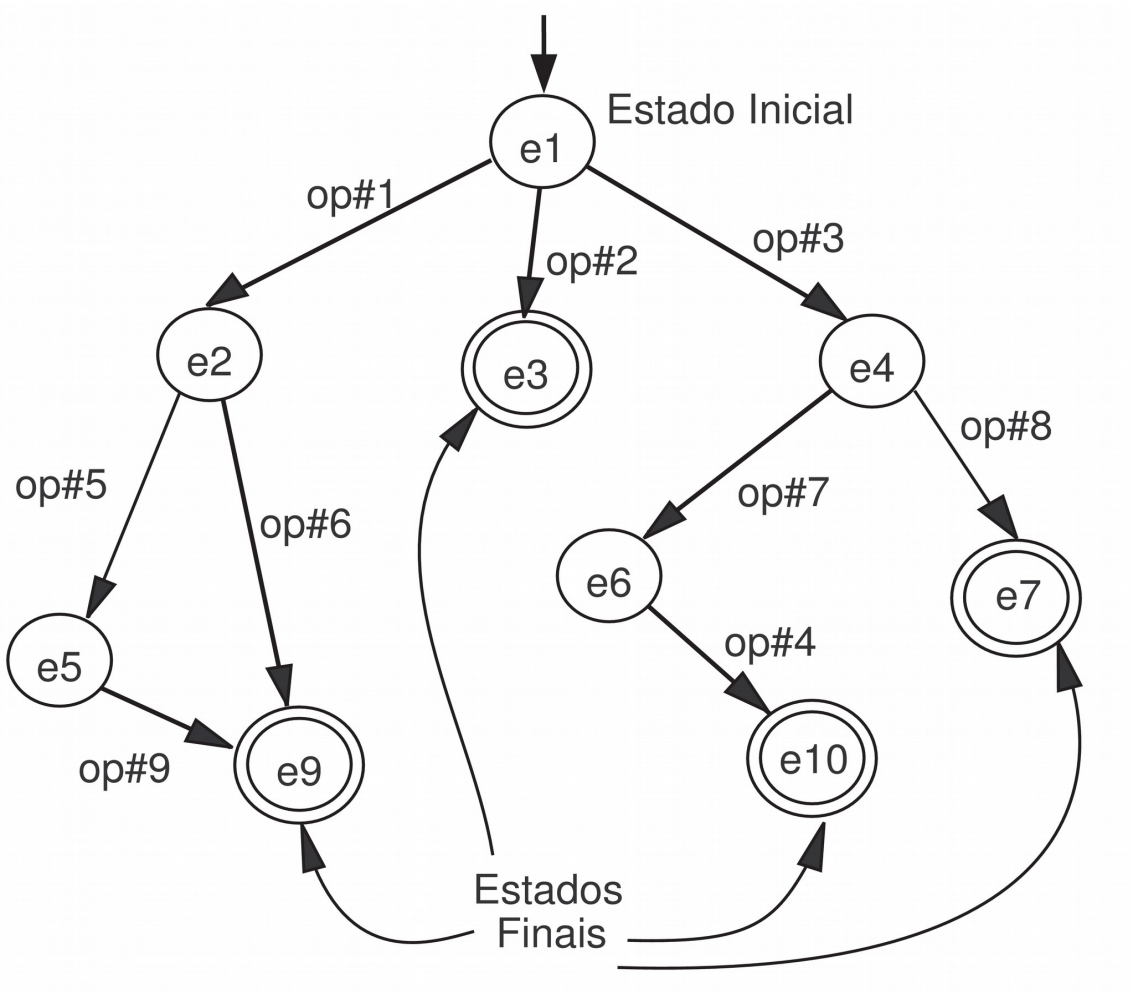


Figura: Nó inicial e nós finais do problema

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), !.
search(S,Visited,P) :-
     next_state(S,Nxt),
     safe_state(Nxt),
     no_loop(Nxt,Visited),
     search(Nxt,[Nxt|Visited],P).

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>.

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 de código *Prologuiano*

Reusando o Conhecimento de Listas e Functores

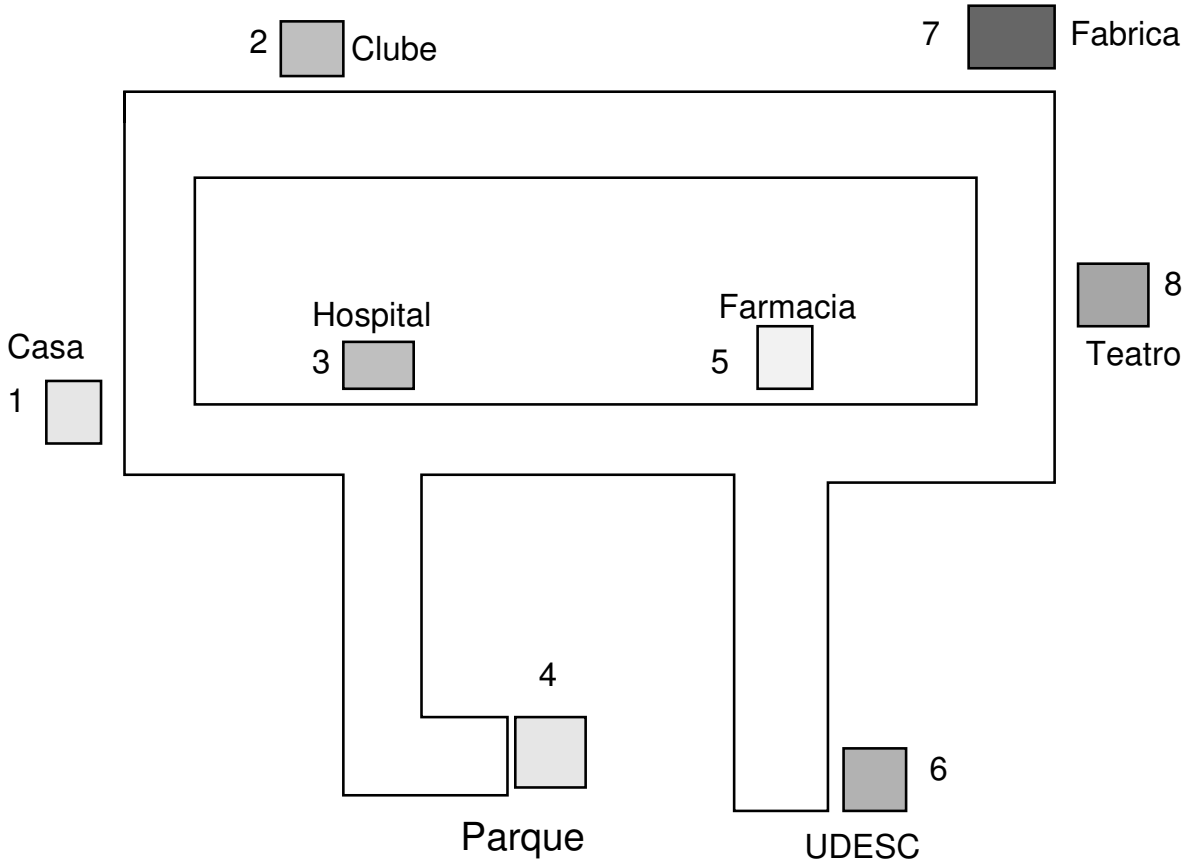


Figura: Um mapa - grafo clássico

Link dos códigos

Github:

- https://github.com/claudiosa/Prolog/blob/master/mapa_cidade_largura.pl
- https://github.com/claudiosa/Prolog/blob/master/mapa_cidade_profundidade.pl

Predicados Extra-Lógicos

Exemplos:

```
if - then - else

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

for
while
```

Predicados *Mão-na-Roda*

Exemplos:

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

statistics \*para estatasticas
do sistema, tempo de cpu, etc.*\

  statistics(cputime,T1),
  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
```

Predicados *Mão-na-Roda*

Dica:

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

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),
          X > 0, !.
cor(X) :- cor(X).
```

Sites Interessantes

- `http://www.cse.unsw.edu.au/~billw/Prologdict.html` (Dicionário de Prolog)
- `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, 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/`

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.