Lógica Computacional

PROLOG

Maria João Frade
Departamento de Informática
Universidade do Minho
2006

2° Ano LMCC (2005/06)

Prolog

Lógica Computacional (Práticas)
2005/2006

Lic. Matemática e Ciências da Computação

Maria João Frade (**mjf@di.uminho.pt**)

Departamento de Informática
Universidade do Minho

Bibliografia

Prolog Programming for Artificial Intelligence - (2nd edition).
Ivan Bratko, Addison-Wesley, 1993.

The Art of Prolog : advanced programming techniques - (2nd edition).
L. Sterling & E. Shaphiro, MIT Press, 1994.

Essentials of Logic Programming.
Christopher John Hogger. Oxford University Press, 1990.

SICStus Prolog – *User's Manual*
http://www.sics.se/sicstus/docs/latest/html/sicstus.html/index.html

PROLOG Uma linguagem de PROgramação em LÓGica.

A linguagem Prolog surgiu no início da década de 70.

O Prolog é uma linguagem de declarativa que usa um fragmento da lógica de 1ª ordem (as *Cláusulas de Horn*) para representar o conhecimento sobre um dado problema.

Um programa em Prolog é um "conjunto" de axiomas e de regras de inferência (definindo relações entre objectos) que descrevem um dado problema. A este conjunto chama-se normalmente *base de conhecimento*.

A execução de um programa em Prolog consiste na dedução de consequências lógicas da base de conhecimento.

O utilizador coloca questões e o "motor de inferência" do Prolog pesquisa a base de conhecimento à procura de axiomas e regras que permitam (por dedução lógica) dar uma resposta. O motor de inferência faz a dedução aplicando o algoritmo de resolução de 1ª ordem.

Exemplo de um **programa** Prolog (um conjunto de *Cláusulas de Horn*).

Factos

```
mae(sofia,joao).
mae(ana,maria).
mae(carla,sofia).

pai(paulo,luis).
pai(paulo,sofia).
pai(luis,pedro).
pai(luis,maria).
```

Regras

```
progenitor(A,B) :- pai(A,B).
progenitor(A,B) :- mae(A,B).

avo(X,Y) :- progenitor(X,Z), progenitor(Z,Y).
```

Átomos

Variáveis (lógicas)

$\forall A \, \forall B. \; progenitor(A,B) \Leftarrow pai(A,B)$

$\forall A \, \forall B. \; progenitor(A,B) \Leftarrow mae(A,B)$

$\forall X \, \forall Y. \; avo(X,Y) \Leftarrow \exists Z. \; progenitor(X,Z) \wedge progenitor(Z,Y)$

Cláusulas de Horn são fórmulas da forma $p \Leftarrow q1 \land q2 \land \ldots \land qn$

representadas em Prolog por `p :- q1, q2, ..., qn`

<cabeça da cláusula> `:-` *<corpo da cláusula>*

Notas:

Os **factos** são cláusulas de Horn com o corpo vazio.

As variáveis que aparecem nas cláusulas são quantificadas universalmente e o seu âmbito é toda a cláusula, mas podemos ver as variáveis que ocorrem apenas no corpo da cláusula (mas não na cabeça), como sendo quantificadas existencialmente dentro do corpo da cláusula.

As **questões** são cláusulas de Horn com cabeça vazia.

As questões são um meio de extrair informação de um programa. As variáveis que ocorrem nas questões são quantificadas existencialmente.

Exemplos de **questões** à base de conhecimento.

```
| ?- avo(ana,joao).
no
```

```
| ?- progenitor(luis,maria).
yes
```

```
| ?- pai(X,maria), pai(X,pedro).
X = luis ?
yes
```

```
| ?- avo(paulo,X).
X = pedro ?
yes
```

```
| ?- avo(paulo,X).
X = pedro ? ;
X = maria ? ;
X = joao ? ;
no
```

```
| ?- progenitor(X,Y).
X = paulo,
Y = luis ? ;
X = paulo,
Y = sofia ? ;
X = luis,
Y = pedro ? ;
X = luis,
Y = maria ? ;
X = sofia,
Y = joao ? ;
X = ana,
Y = maria ? ;
X = carla,
Y = sofia ? ;
no
```

Responder a uma questão é determinar se a questão é uma consequência lógica do programa.

Responder a uma questão com variáveis é dar uma instânciação da questão (representada por uma substituição para as variáveis) que é inferível do programa.

O utilizador ao digitar "**;**" força o motor de inferência a fazer *backtracking*. Ou seja, pede ao motor de inferência para construir uma prova outra prova (alternativa) para a questão que é colocada. Essa nova prova pode dar origem a uma outra substituição das variáveis.

A arte de programar em lógica está, em parte, na escolha da axiomatização mais elegante, clara e completa para os problemas.

Prolog (em resumo)

SICStus Prolog – *User's Manual*
http://www.sics.se/sicstus/docs/latest/html/sicstus.html/index.html

O interpretador

```
> sicstus
SICStus 3.11.0 (x86-linux-glibc2.3): Mon Oct 20 15:59:37 CEST 2003
Licensed to di.uminho.pt
| ?-
```

Notas: As extensão usual dos ficheiros (sicstus) prolog é **.pl**

O *ponto fina*l assinala o final das cláusulas.

Carregar ficheiros

```
| ?- consult(file).
| ?- consult('file.pl').
| ?- [file1, 'file2.pl', '/home/abc/ex/file3'].
```

Listar a base de conhecimento que está carregada

```
| ?- listing.
```

Para sair do interpretador

```
| ?- halt.
```

Termos

O **termos** são as entidades sintácticas que representam os objectos (do universo de discurso da lógica de 1ª ordem).
Os termos podem ser *constantes*, *variáveis* ou *termos compostos*.

Constantes

Inteiros

Base 10	`5`	`0`	`27`	`-48`
Outras bases (de 2 a 36)	`2'101`	`8'175`		
Código ASCII	`0'A`	`0'z`	*(65 e 122 respectivamente)*	

Reais

`2.0   -5.71   47.0E3   -0.9e-6`

Átomos

Qualquer sequência de caracteres alfanuméricos começada por letra minúscula

Qualquer sequência de `+ - * / \ ^ >< = ~ : . ? @ # $ &`

Qualquer sequência de caracteres entre `' '`

`!   ;   []   {}`

Variáveis

Qualquer sequência de caracteres alfanuméricos iniciada com letra maiúscula ou _ .

Exemplos: `X   A   _   Valores   _aa   _5   RESP   _VAL`

variável anónima (o `_`)

Termos Compostos

São objectos estruturados da forma *`f(t1,...,tn)`* em que

`f` é o **functor** do termo e `t1, ..., tn` são termos *(subtermos)*.
O functor `f` tem **aridade** `n`.

O nome do functor é um átomo.

Exemplos: `suc(0)   suc(suc(0))   data(15,abril,2006)`

Há functores que podem ser declarados como operadores *(infixos, prefixos, posfixos)*.

Exemplo: `+(X,Y)` ⇒ `X+Y`

Listas

Listas são termos gerados à custa do átomo `[]` *(a lista vazia)*

e do functor `.(H,T)` *(o cons)*

Exemplo: O termo `.(1,.(2,.(3,[])))` representa a lista `[1,2,3]`

O Prolog tem uma notação especial para listas `[` *head* `|` *tail* `]`

Exemplo:

```
.(a,.(b,.(c,[]))) = [a|[b,c]] = [a,b|[c]] = [a,b,c|[]] = [a,b,c]
```

String

Uma string é a lista de códigos ASCII dos seus caracteres.

Exemplo: `"PROLOG" = [80,82,79,76,79,71]`

Note que `"PROLOG"` é diferente de `'PROLOG'`

Programas

Um programa é um conjunto de *cláusulas de Horn*. Ou seja, fórmulas da forma

$$p \Leftarrow q1 \wedge q2 \wedge \ldots \wedge qn$$

representadas em Prolog por `p :- q1, q2, ..., qn`

<cabeça da cláusula> `:-` *<corpo da cláusula>*

Factos são cláusula só com cabeça e de corpo vazio.
Regras são cláusula com cabeça e corpo não vazio.

Exemplo:

comentários

```
% grafo orientado
caminho(a,b).
caminho(b,c).

/*  verifica se existe ligação entre
dois nodos dos grafo */
ligacao(X,Y) :- caminho(X,Y).
ligacao(X,Y) :- caminho(X,Z), ligacao(Z,Y).
```

Os factos e regras com o mesmo nome (à cabeça) definem um ***predicado*** (ou ***procedimento***). Neste exemplo definimos os predicados `caminho/2` e `ligacao/2` *(nome/aridade)*. Note que é possível ter predicados distintos com o mesmo nome mas aridades diferentes.

Questões são cláusulas de Horn com cabeça vazia.

As questões são um meio de extrair informação de um programa. As variáveis que ocorrem nas questões são quantificadas existencialmente.

Responder a uma questão é determinar se ela é uma consequência lógica do programa.

Exemplo:

```
| ?- ligacao(a,K).
K = b ?
yes
```

```
| ?- ligacao(a,K).
K = b ? ;
K = c ?
yes
```

```
| ?- ligacao(a,K).
K = b ? ;
K = c ? ;
no
```

Força o *backtracking*

i.e., pede para serem produzidas novas provas (alternativas)

Justifique esta resposta desenhando a *árvore de procura* de `? ligacao(a,K).`

Estratégia de prova do motor de inferência do Prolog

Assuma que o objectivo a provar (o *goal*) é: `? G1, G2, ..., Gn`

O motor de inferência pesquisa a base de conhecimento (de cima para baixo) até encontrar uma regra cuja cabeça *unifique* com `G1`. Essa unificação produz uma *substituiçao* (o *unificador mais geral*) θ

↳ Se `C :- P1, ..., Pm` é a regra encontrada. θ é tal que `C θ = G1 θ`.

O novo objectivo a provar é agora `? P1θ, ..., Pmθ, G2θ, ..., Gnθ`

↳ Se a regra encontrada é um facto `F`. θ é tal que `F θ = G1 θ` .

O novo objectivo a provar é agora `? G2θ, ..., Gnθ`

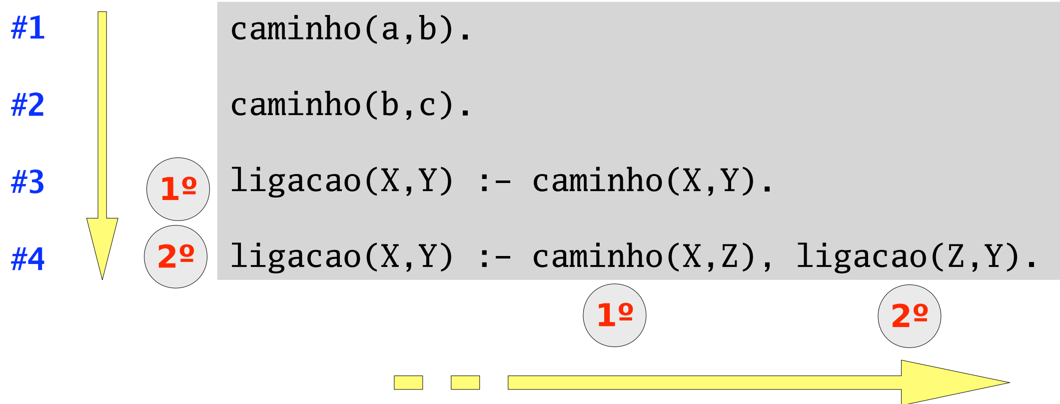
➡ A prova termina quando já não há mais nada a provar (o *goal* é vazio).
O interpretador responde à questão inicial indicando a substituição a que têm que ser sujeitas as variáveis presentes na questão inicial, para produzir a prova.

Estratégia de Procura de Soluções

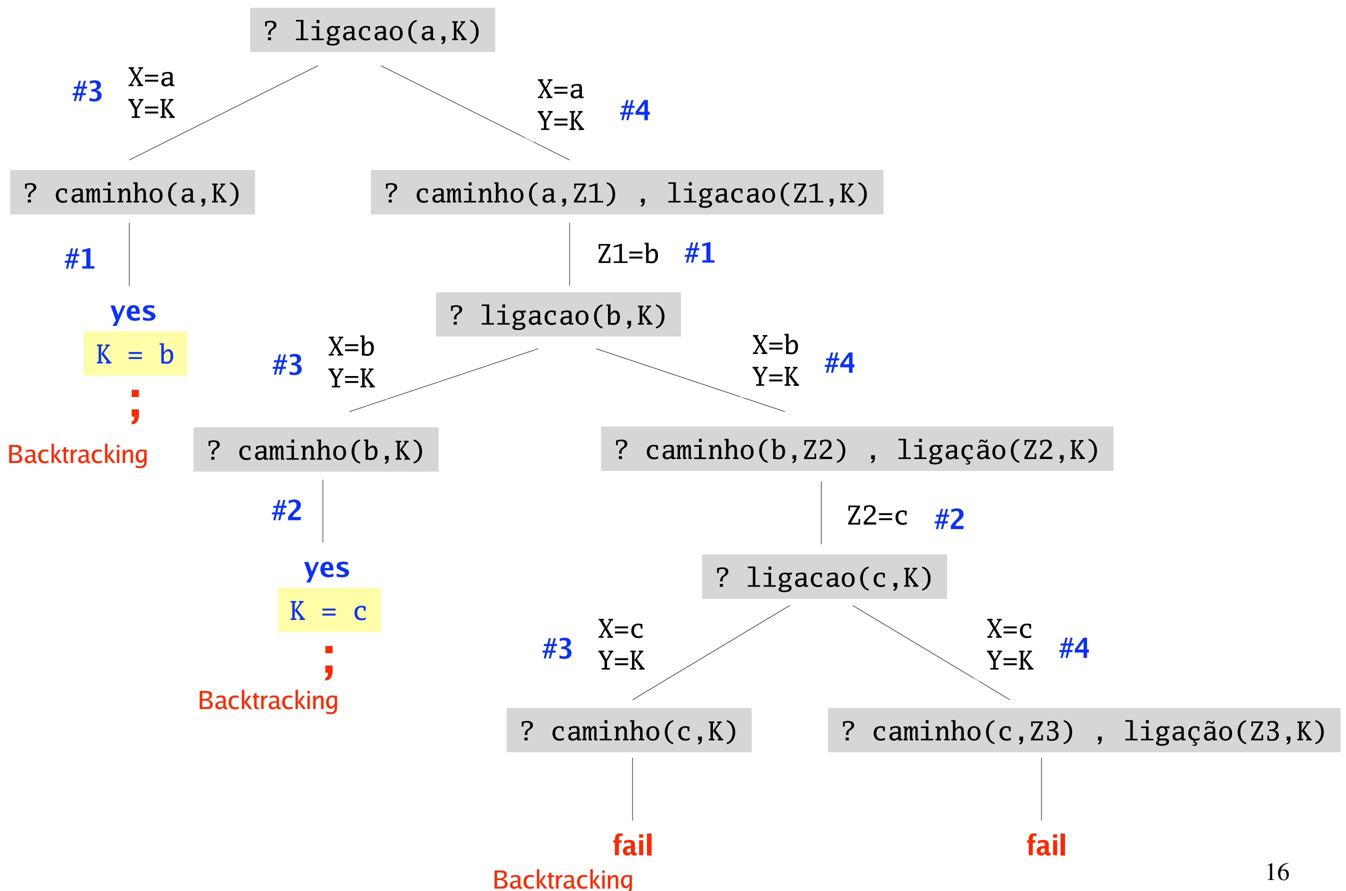
Top Down (de cima para baixo)

Depth-First (profundidade máxima antes de tentar um novo ramo)

Backtracking (volta a tentar encontrar uma prova alternativa)



Árvore de Procura



Observação:

Se o programa fosse

```
cam(a,b).
cam(b,c).

lig(X,Y) :- cam(X,Y).

lig(X,Y) :- lig(Z,Y), cam(X,Z).
```

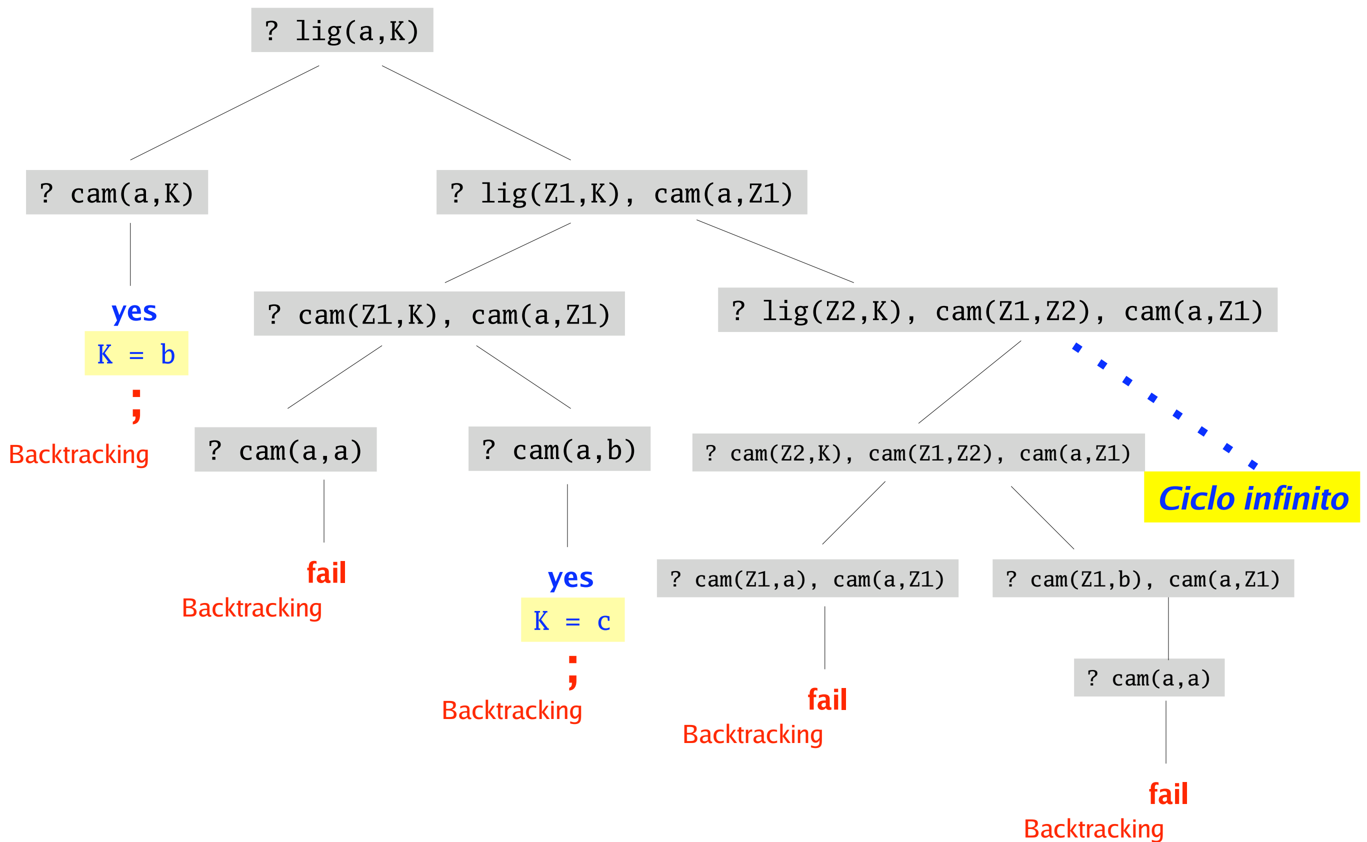
teriamos uma árvore de procura *infinita*.

Note que a única diferença entre o `ligação` e `lig` é a ordem em que aparecem os predicados no corpo da 2ª cláusula.

```
ligacao(X,Y) :- caminho(X,Z), ligacao(Z,Y).

lig(X,Y) :- lig(Z,Y), cam(X,Z).
```


Árvore de Procura



Exemplos com listas

```
% pertence(X,L) indica que X é um elemento da lista L
pertence(X,[X|_]).
pertence(X,[_|T]) :- pertence(X,T).
```

```
% prefixo(L1,L2) indica que a lista L1 é prefixo da lista L2
prefixo([],_).
prefixo([X|Xs],[X|Ys]) :- prefixo(Xs,Ys).
```

```
% sufixo(L1,L2) indica que a lista L1 é sufixo da lista L2
sufixo(L,L).
sufixo(Xs,[Y|Ys]) :- sufixo(Xs,Ys).
```

```
% concatena(L1,L2,L3) indica qua a lista L1 concatenada com a lista L2 é a lista L3
concatena([],L,L).
concatena([H|T],L1,[H|L2]) :- concatena(T,L1,L2).
```

Exercício:

Carregue estas definições no interpretador e interrogue a base de conhecimento. Por exemplo:

```
? pertence(2,[1,2,3]).
? pertence(X,[1,2,3]).
? pertence(3,L).
? pertence(X,L).
```


Exercícios:

1. Considere a representação dos números naturais baseada nos construtores 0 e sucessor: *0, suc(0), suc(suc(0)), suc(suc(suc(0))), ...*

   O predicado `nat` que testa se um termo é um número natural.

   ```
   nat(0).
   nat(suc(X)) :- nat(X).
   ```

   Defina, usando o functor `suc`, predicados que implementem as seguintes relações:

   a) menor ou igual
   b) mínimo
   c) soma
   d) multiplicação
   e) factorial
   f) exponenciação

2. Defina o predicado `last(X,L)` que testa se `X` é o último elemento da lista `L`.

3. Defina a relação `divide(L,L1,L2)` que divide a lista `L` em duas listas `L1` e `L2` com aproximadamente o mesmo tamanho.

Resolução: Uma solução para a soma de números naturais

```
soma(0,N,N).
soma(suc(N),M,suc(Z)) :- soma(N,M,Z).
```

Exemplo de uma árvore de procura

```
? soma(suc(suc(0)),suc(0),X)
        |   N1=suc(0)
        |   M1=suc(0)
        |   X=suc(Z1)
? soma(suc(0),suc(0),Z1)
        |   N2=0
        |   M2=suc(0)
        |   Z1=suc(Z2)
? soma(0,suc(0),Z2)
        |   Z2=suc(0)
      yes
X = suc(suc(suc(0)))

  = suc(Z1) = suc(suc(Z2))
  = suc(suc(suc(0)))
```


Operadores

Para conveniência de notaçao, o Prolog permite declarar functores unários ou binários como operadores *prefixos*, *posfixos* ou *infixos*, associando-lhes ainda uma precedência.

A declaração de operadores faz-se através da directiva

:- op(*precedência*, *tipo*, *nome*)

- *precedência*: Nº de 1 e 1200
- *nome*: Functor ou lista de functores
- *tipo*:

	infixo	prefixo	posfixo
associativo à esquerda	yfx		yf
associativo à direita	xfy	fy	
não associativo	xfx	fx	xf

Exemplos:

```
:- op(500,yfx,[+,-])
:- op(400,yfx,*)
```

```
:- op(300,fy,nao)
```

`a-d+b*c` , `(a-d)+(b*c)` e `+(-(a,d),*(b,c))` são termos equivalentes

`nao nao p` , `nao (nao p)` e `nao(nao(p))` são termos equivalentes

= operador de unificação

O operador infixo = establece uma relação de unificação entre dois termos.

T1 = T2 sucede se *o termo T1 unifica com o termo T2*

Dois termo T1 e T2 **unificam** se existir uma substituição θ que aplicada aos termos T1 e T2 os torne literalmente iguais. Isto é, T1θ == T2θ .

- Dois átomos (ou números) só unificam se forem exactamente iguais.
- Dois termos compostos unicam se os functores principais dos dois termos forem iguais e com a mesma aridade, e os argumentos respectivos unificam.
- Uma variável unifica com qualquer termo (gerando a substituição respectiva).

Exemplo:

```
| ?- data(23,maio,1998) = data(23,maio,1998).
yes
| ?- data(23,maio,1998) = data(23,maio,Ano).
Ano = 1998 ?
yes
| ?- data(23,maio,1998) = data(15,maio,Ano).
no
| ?- data(23,maio,1998) = data(X,maio,X).
no
```


"Occurs-Check"

Exemplo:

```
| ?- hora(10,30) = hora(X,Y).
X = 10,
Y = 30 ?
yes
| ?- hora(H,30) = hora(10,M).
H = 10,
M = 30 ?
yes
| ?- hora(H,30,12) = hora(X,Y,X).
H = 12,
X = 12,
Y = 30 ?
yes
| ?- hora(H,30,12) = hora(X,X,X).
no
| ?- X = f(X).
X = f(f(f(f(f(f(f(f(f(f(...)))))))))) ?
yes
```

O Sicstus Prolog permite unificar uma variável com um termo em que essa variável ocorre, permitindo assim a criação de *termos cíclicos*. Formalmente isto não é desejável mas o teste de ocorrência não é feito por razões de eficiência.

Operadores aritméticos

Alguns operadores aritméticos do Sicstus Prolog (ver *User's Manual*):

+ - / * // mod abs sign gcd min max round truncate

sin cos tan cot asin acos ... sqrt log exp ** ...

Estes **operadores** são **simbólicos**, permitem construir expressões aritméticas, mas não efectuam qualquer cálculo.

Exemplo:

```
| ?- 5 = 4+1.
no
| ?- 3+X = 5.
no
```

```
| ?- A = 3 mod 2.
A = 3 mod 2 ?
yes
| ?- X-7 = 3*4-7.
X = 3*4 ?
yes
```

O **cálculo aritmético** é efectuado utilizando os seguintes predicados aritméticos pré-definidos:

`=:=    =\=    <    >    =<    >=`

Comparam os valores das expressões numéricas.

Z **is** *expressão*

A expressão é calculada e o seu resultado unifica com Z. Se a expressão não for numérica a cláusula falha.

Exemplos:

```
| ?- round(3.5) =:= round(3.9).
yes
| ?- abs(-5*min(1,7)) < 7//2.
no
```

```
| ?- X is gcd(20,15).
X = 5 ?
yes
| ?- Y is 5**2.
Y = 25.0 ?
yes
```

```
% tamanho(?L,?N)  N é o comprimento da lista L
tamanho([],0).
tamanho([_|T], Z) :- tamanho(T,X), Z is X+1.
```

```
% fib(+N,?Z)  Z é o número de Fibonacci de N
fib(0,0).
fib(1,1).
fib(N,X) :- N > 1, N1 is N-1, fib(N1,A),
            N2 is N-2, fib(N2,B), X is A+B.
```

Nota: Na documentação de um predicado costuma-se usar a seguinte notação:

- **+** *o argumento deve estar instanciado*
- **–** *o argumento deve ser uma variável não instanciada*
- **?** *o argumento pode, ou não, estar instanciado*
- **:** *o argumento tem que ser um predicado*

Comparação de termos

O Sicstus Prolog tem predicados pré-definidos para comparação (sintáctica) de termos (ver *User's Manual*).

termo1 == termo2	*Testa se os termos são literalmente iguais.*
termo1 \== termo2	*Testa se os termos não são literalmente iguais.*

Exemplos:

```
| ?- X == Y.
no
| ?- X*4 \== X*4
no
```

```
| ?- X = Y, X == Y.
Y = X ?
yes
| ?- X == Y, X = Y.
no
```

O Sicstus Prolog establece uma relação de ordem no conjunto de termos (ver o manual).
A comparação de termos de acordo com essa ordem pode ser feita com os operadores:

`@<` `@>` `@=<` `@>=`

Exemplos:

```
| ?- abcd @< xyz.
yes
```

```
| ?- abcd(1,2) @=< xyz.
no
```


Exemplos:

```
| ?- hora(3+5,7-3) = hora(10-2,2*2).
no
| ?- hora(3+5,7-3) == hora(10-2,2*2).
no
| ?- [user].
% consulting user...
| :- op(700,xfx,igual).
| igual(hora(H1,M1),hora(H2,M2)) :- H1 =:= H2, M1 =:= M2.
|
% consulted user in module user, 0 msec 504 bytes
yes
| ?- hora(3+5,7-3) igual hora(10-2,2*2).
yes
| ?-
```

`[user].` Permite acrescentar novas regras na base de conhecimento.
É um editor muito primitivo (linha a linha).
^D para sair do editor e carregar as novas regras.

Note que não se está a alterar nenhum ficheiro!

Exemplos:

```
% apaga todas as ocorrências de um dado elemento de uma lista
apaga([H|T],H,L) :- apaga(T,H,L).
apaga([H|T],X,[H|L]) :- H \== X, apaga(T,X,L).
apaga([],_,[]).
```

```
% ordenação de uma lista com o algoritmo insertion sort
isort([],[]).
isort([H|T],L) :- isort(T,T1), ins(H,T1,L).

ins(X,[],[X]).
ins(X,[Y|Ys],[Y|Zs]) :- X > Y, ins(X,Ys,Zs).
ins(X,[Y|Ys],[X,Y|Ys]) :- X =< Y.
```

Exercícios:

1. Defina os seguintes predicados sobre listas:
   a) `minimo/2` que produz o menor elemento presente numa lista.
   b) `somatorio/2` que cálcula o somatório de uma lista.
   c) `nesimo/3` que dá o elemento da lista na n-ésima posição
2. Defina um procedimento que ordene de uma lista segundo o algoritmo *quicksort.*

Uso de Acumuladores

Exemplo: Inversão de uma lista com e sem acumuladores.

```
inverte([],[]).
inverte([H|T],L) :- inverte(T,T1), concatena(T1,[H],L).
```

acumulador

```
inv(Xs,Ys) :- inv(Xs,[],Ys).

inv([X|Xs],Ac,Ys) :- inv(Xs,[X|Ac],Ys).
inv([],Ys,Ys).
```

Exercício:

1. Considere o seguinte procedimento para o cálculo do factorial

   ```
   fact(0,1).
   fact(N,F) :- N>0, N1 is N-1, fact(N1,F1), F is N*F1.
   ```

   Defina uma outra versão de factorial que utilize um parâmetro de acumulação.

2. Defina uma versão do predicado somatório que utilize um acumulador.

Predicados de tipo *(meta-lógica)*

O Sicstus Prolog tem um conjunto de predicados pré-definidos que permitem fazer uma análise dos termos (ver *User's Manual*).

var nonvar atom number atomic simple compound ground

integer float ...

```
functor(+Term,?Name,?Arity)
functor(?Term,+Name,+Arity)

arg(+ArgNo,+Term,?Arg)
```

```
+Term =.. ?List
?Term =.. +List

name(+Const,?CharList)
name(?Const,+CharList)
```

Exemplo:

```
| ?- functor(data(X,abril,2006),N,A).
A = 3,
N = data ?
yes
| ?- functor(X,hora,2).
X = hora(_A,_B) ?
yes
| ?- name('ABC',X), name(abc,Y), name(123,Z).
X = [65,66,67],
Y = [97,98,99],
Z = [49,50,51] ?
yes
```

```
| ?- arg(2,data(X,abril,2006),A).
A = abril ?
yes
| ?- data(X,abril,2006) =.. L.
L = [data,X,abril,2006] ?
yes
| ?- Z =.. [hora,12,30].
Z = hora(12,30) ?
yes
```


Exemplos: Duas implementações de predicados que testam a relação de subtermo.

```
% subtermo(T1,T2) testa se T1 é subtermo de T2
subtermo(T1,T2) :- T1 == T2.
subtermo(S,T) :- compound(T), functor(T,F,N), subtermo(N,S,T).

subtermo(N,S,T) :- N>1, N1 is N-1, subtermo(N1,S,T).
subtermo(N,S,T) :- arg(N,T,A), subtermo(S,A).
```

```
% subterm(T1,T2) testa se T1 é subtermo de T2
subterm(T,T).
subterm(S,T) :- compound(T), T =.. [F|As], subtermList(S,As).

subtermList(S,[A|R]) :- subterm(S,A).
subtermList(S,[A|R]) :- subtermList(S,R).
```

Note as diferenças entre as duas versões.

```
| ?- subterm(f(X),g(h(t,f(X)),a)).
true ?
yes
```

```
| ?- subtermo(f(t),g(h(t,f(X)),a)).
no
| ?- subterm(f(t),g(h(t,f(X)),a)).
X = t ?
yes
```

Exercícios:

1. Defina a relação `flatten/2` (que lineariza uma lista) de forma a que, por exemplo:

```
| ?- flatten([a,b,[c,d],[],[[[e,f]],g],h],X).
X = [a,b,c,d,e,f,g,h] ?
yes
```

2. Escreva um programa para reconhecer se uma fórmula da lógica proposicional está na forma normal conjuntiva, ou seja, é uma conjunção de disjunções de literais. Um literal é um símbolo proposicional ou a sua negação.

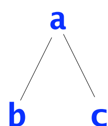
   Considere a declaração das seguintes conectivas lógicas:

```
:- op(500,yfx,/\).
:- op(500,yfx,\/).
:- op(300,fy,~).
```

3. Defina o predicado `conta_ocorr/3` para contar quantas vezes uma constante ocorre numa lista. (Sugestão: usar `atomic/1`).

4. Suponha que tem factos da forma *quadrado(Lado)*.
   Defina o predicado `zoom(+X,?Y,+F)` tal que `Y` é o quadrado que resulta de multiplicar pelo factor `F` os lados do quadrado `X`. (Sugestão: usar `=..`).

Um exemplo com árvores binárias



Esta árvore é representada pelo termo

nodo(a, nodo(b,vazia,vazia), nodo(c,vazia,vazia))

Exemplo:

```
arv_bin(vazia).
arv_bin(nodo(X,Esq,Dir)) :- arv_bin(Esq), arv_bin(Dir).

na_arv(X,nodo(X,_,_)).
na_arv(X,nodo(Y,Esq,_)) :- na_arv(X,Esq).
na_arv(X,nodo(Y,_,Dir)) :- na_arv(X,Dir).
```

Exercícios:

1. Defina predicados que permitam fazer as travessias *preorder*, *inorder* e *postorder*.
2. Defina um predicado `search_tree/1` que teste se uma dada árvore é uma árvore binária de procura.
3. Defina a relação `insert_tree(+X,+T1,?T2)` que sucede se `T2` é uma árvore binária de procura resultado da inserção de `X` na árvore binária de procura `T1`.
4. Defina a relação `path(+X,+Tree,?Path)` que sucede se `Path` é o caminho da raiz da árvore binária de procura `Tree` até `X`.

O *cut* !

O ***cut*** **!** não deve ser visto como um predicado lógico. Apenas interfere na semântica procedimental do programa. A sua acção é a seguinte:

> *Durante o processo de prova, a 1ª passagem pelo cut é sempre verdadeira (com sucesso). Se por backtracking se voltar ao cut, então o cut faz falhar o predicado que está na cabeça da regra.*

O cut "corta" ramos da árvore de procura. Os predicados antes do *cut* são apenas instanciados uma vez.

Exemplo:

```
x :- p, !, q.
x :- r.
```

`x` tem um comportamento semelhante a

`if p then q else r`

Green Cut

Chamam-se ***green cuts*** aos *cuts* que só alteram a semântica procedimental do programa, mas não alteram o significado do predicado. Este *cuts* são usados apenas para melhorar a eficiência. Se forem retirados serão produzidos os mesmos resultados.

Red Cut

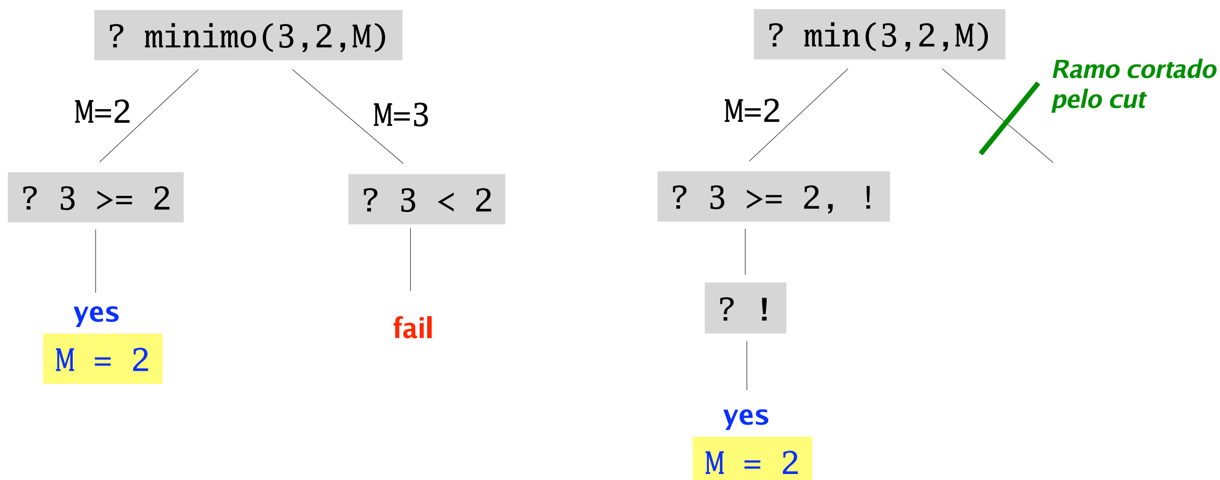
Chamam-se ***red cuts*** aos *cuts* que alteram não só a semântica procedimental do programa, como também o seu significado declarativo. Se forem retirados serão produzidos resultados diferentes. A utilização destes *cuts* deve ser evitada.

Exemplos:

`minimo` e `min` são predicados equivalentes.
O cut só está a cortar ramos que falham.

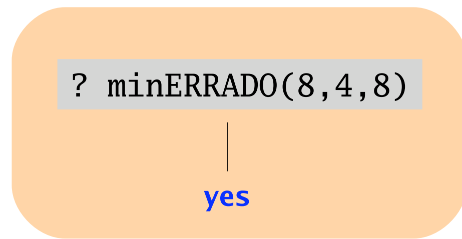
```
minimo(X,Y,Y) :- X >= Y.
minimo(X,Y,X) :- X < Y.
```

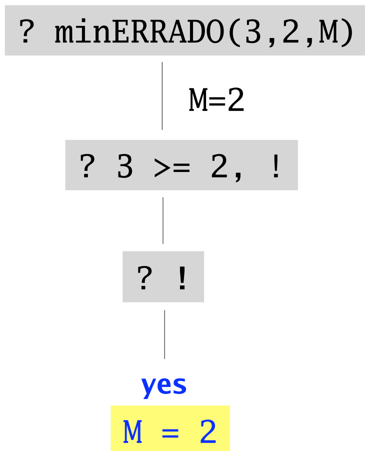
```
min(X,Y,Y) :- X >= Y, !.
min(X,Y,X) :- X < Y.
```

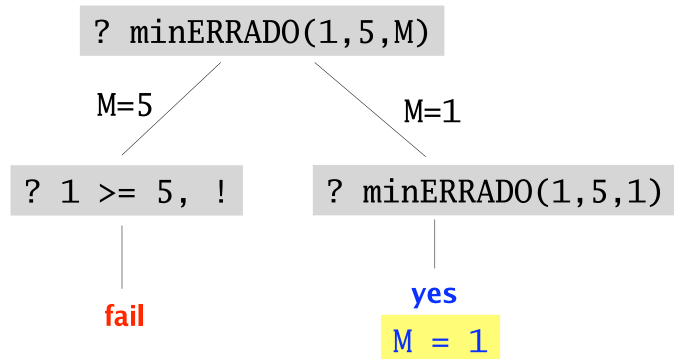


Exemplo de utilização *errada* do cut.

```
minERRADO(X,Y,Y) :- X >= Y, !.
minERRADO(X,Y,X).
```







Exercícios:

1. Defina um predicado `no_dupl/2` que remova os duplicados de uma lista.

2. Defina um predicado `enumerar(+N,+M,+P,?L)` que gera a lista `L` de números entre `N` e `M`, a passo `P`. Por exemplo:

```
| ?- enumerar(3,10,2,L).
L = [3,5,7,9] ?
yes
```

3. Defina um programa que faça a ordenação de uma lista pelo algoritmo *merge sort*. Use o *cut* para implementar o predicado `merge` de forma mais eficiente.

Negação por falha

O Prolog tem pré-definidos os predicados **true** e **fail** :

true sucede sempre,
fail falha sempre

Utilizando o *cut* e o *fail* é possível implementar a negação.

Exemplo:

```
:- op(700,fy,nao).

nao X :- X, !, fail.
nao X.
```

nao X falha se o predicado X tiver solução, i.e., for verdadeiro.

No *Sicstus Prolog* o predicado de negação (por falha) é o **\+** .

Exemplos:

```
| ?- fib(1,1).
yes
| ?- nao fib(1,1).
no
| ?- \+ fib(1,1).
no
```

```
| ?- nao nao fib(1,1).
yes

| ?- \+ \+ fib(1,1).
yes
```

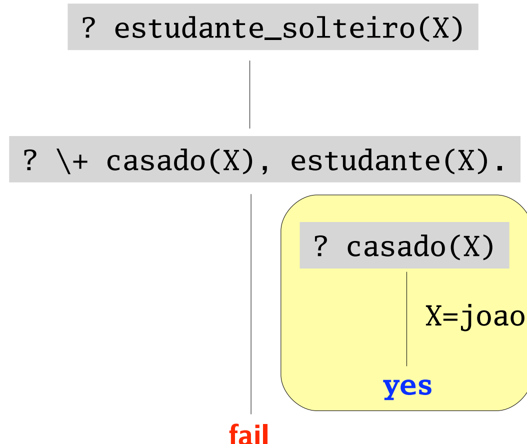

Problemas com a negação por falha

Exemplo:

```
estudante(paulo).
casado(joao).

estudante_solteiro(X) :- \+ casado(X), estudante(X).
```

```
| ?- estudante_solteiro(paulo).
yes
| ?- estudante_solteiro(X).
no
```



Deveria ser

```
estSolt(X) :-  estudante(X), \+ casado(X).
```

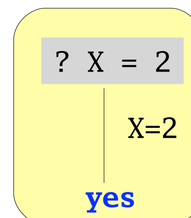
```
| ?- estSolt(X).
X = paulo ?
yes
```

Nas chamadas ao not, \+, não devem ocorrer variáveis.

Exemplo:

```
| ?- \+ X=2, X=1.
no
```

? \+ X = 2, X = 1.



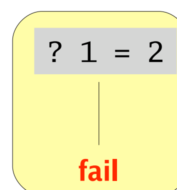
fail

```
| ?- X=1, \+ X=2.
X = 1 ?
yes
```

? X = 1, \+ X = 2.

X=1

? \+ 1 = 2.



yes

Outros predicados de controlo

Para além da *conjunção* de predicados (representada por **,**), também é possível combinar predicados pela *disjunção* (representada por **;**).

Exemplo:

```
progenitor(A,B) :- pai(A,B).
progenitor(A,B) :- mae(A,B).

avo(X,Y) :- progenitor(X,Z), progenitor(Z,Y).
```

```
progenitor(A,B) :- pai(A,B) ; mae(A,B).
```

```
tio(X,Y) :- (pai(A,Y) ; mae(A,Y)), irmao(X,A).
```

O Sicstus Prolog disponibiliza mais alguns predicados de control *(ver User's Manual)*.
Por exemplo:

+P **->** *+Q***;** *+R*	equivalente a	(P -> Q; R) :- P, !, Q. (P -> Q; R) :- R.
+P **->** *+Q*	equivalente a	(P -> Q; fail)
once(*+P*)	equivalente a	(P -> true; fail)

Programação de 2ª ordem

Existem meta-predicados que permitem coleccionar todas as soluções para um dado objectivo de prova (ver *User's Manual*).

findall(*?Template*,*:Goal*,*?Bag*)

Bag é a lista de instâncias de *Template* encontradas nas provas de *Goal*.
A ordem da lista corresponde à ordem em que são encontradas as respostas.
Se não existirem instanciações para *Template*, *Bag* unifica com a lista vazia.

bagof(*?Template*,*:Goal*,*?Bag*)

Semelhante a findall, mas se *Goal* falhar, bagof falha.

setof(*?Template*,*:Goal*,*?Set*)

Semelhante a bagof, mas a lista é ordenada e sem repetições.

Exemplo: Considere o seguinte programa

```
amigo(ana,rui).
amigo(pedro,rui).
amigo(maria,helena).
amigo(pedro,ana).
amigo(maria,rui).

gosta(ana,cinema).
gosta(ana,pintura).
gosta(ana,ler).
gosta(rui,ler).
gosta(rui,musica).
gosta(maria,ler).
gosta(pedro,pintura).
gosta(pedro,ler).

compativeis(A,B) :- amigo(A,B), gosta(A,X), gosta(B,X).
compativeis(A,B) :- amigo(B,A), gosta(A,X), gosta(B,X).
```

```
| ?- compativeis(ana,X).
X = rui ? ;
X = pedro ? ;
X = pedro ? ;
no
```

```
| ?- findall(X,compativeis(ana,X),L).
L = [rui,pedro,pedro] ?
yes
| ?- bagof(X,compativeis(ana,X),L).
L = [rui,pedro,pedro] ?
yes
| ?- setof(X,compativeis(ana,X),L).
L = [pedro,rui] ?
yes
```

```
| ?- compativeis(helena,X).
no
```

```
| ?- findall(X,compativeis(helena,X),L).
L = [] ?
yes
| ?- bagof(X,compativeis(helena,X),L).
no
| ?- setof(X,compativeis(helena,X),L).
no
```

```
| ?- setof(comp(X,Y),compativeis(X,Y),L).
L = [comp(ana,pedro),comp(ana,rui),comp(maria,rui),comp(pedro,ana),
comp(pedro,rui),comp(rui,ana),comp(rui,maria),comp(rui,pedro)] ?
yes
```


Exercícios:

1. Defina o predicado `subconj(-S,+C)` onde `S` e `C` são duas listas que representam dois conjuntos. Este predicado deve gerar, por backtracking, todos os subconjuntos possíveis de `C`.

2. Defina o predicado `partes(+C,-P)`, que dado um conjunto `C` (implementado como lista) dá em `P` o conjunto de todos os subconjuntos de `C`.

3. Considere a linguagem proposicional gerada pelos simbolos proposicionais (átomos) e as conectivas: `falso, verdade, ~, /\, \/, =>.` Relembre que um modelo é um conjunto de simbolos proposicionais.

```
:- op(600,xfy,=>).
:- op(500,yfx,/\).
:- op(500,yfx,\/).
:- op(300,fy,~).
```

Defina o predicado `atrib(+M,+P,?V)` que sucede se `V` é o valor de verdade da proposição `P` no modelo `M`. (Relembre a função μ da aulas teóricas).
Pode utilizar o predicado `simp(+E,?V)` que faz o cálculo do valor de uma expressão na álgebra booleana $\mathbf{Z}_2$ (ver slide seguinte).

```
simp(X+X,0).
simp(X*X,X).
simp(X+0,X).
simp(0+X,X).
simp(X*0,0).
simp(0*X,0).

simp(X+Y,Z) :- simp(X,X1), simp(Y,Y1), simp(X1+Y1,Z).
simp(X*Y,Z) :- simp(X,X1), simp(Y,Y1), simp(X1*Y1,Z).
```

4. Defina os predicados `formula_valida(+M,+P)` e `teoria_valida(+M,+T)` que sucedem se a proposição `P` e teoria `T` é válida no modelo `M`.

```
| ?- formula_valida([p,q], p/\q => q\/r => ~r /\ ~ ~p).
yes
```

5. Defina o predicado `consequencia(+T,+P)` sucede se a proposição `P` é uma consequência (semântica) da teoria `T`. (Sugestão: gere primeiro todos os modelos possíveis com os simbolos proposicionais das fórmulas envolvidas.)

6. Defina `tautologia/1` que testa se uma fórmula é uma tautologia.

7. Defina `inconsistente/1` que testa se uma teoria é inconsistente.

8. Defina os predicados `soma/3` e `produto/3` que implementam as operações de soma e produto de conjuntos de conjuntos de literais (ver apontamento da aulas teóricas).

9. Defina o predicado `fnn(+P,-FNN)` que dada uma proposição (da linguagem do exercício 3.) gera a forma normal negativa que lhe é semanticamente equivalente.

10. Defina o predicado `fnc(+FNN,-L)` que dada uma forma normal negativa, produz em `L` a forma normal conjuntiva equivalente, representada por conjuntos de conjuntos de literais.

11. Defina o predicado `constroi_fnc(+L,-P)` que sucede se `P` é a proposição que o conjunto de conjuntos de literarais `L` representa.

12. Defina o predicado `gera_fnc(+P,-FNC)` que dada uma proposição `P` produz `FNC` uma proposição na forma normal conjuntiva, semanticamente equivalente a `P`.

Módulos

Para além dos predicados pré-definidos, o Sicstus Prolog possui um vasto conjunto de **módulos** onde estão implementados muitos predicados que poderão ser úteis em diversas aplicações. Para poder utilizar estes predicados é necessário carregar os módulos onde eles estão definidos.

Os módulos estão organizados por temas. Alguns exemplos de módulos:

- **lists** – fornece os predicados de manipulação de listas
- **terms** – fornece os predicados de manipulação de termos
- **queues** – fornece uma implementação de filas de espera
- **random** – fornece um gerador de números aleatóreos
- **system** – fornece primitivas para aceder ao sistema operativo
- **ugraphs** – fornece uma implementação de grafos sem informação nos arcos
- **wgraphs** – fornece uma implementação de grafos com informação nos arcos
- **tcltk** – fornece um interface para o Tcl/Tk
- **vbsp** – fornece um interface para o Visual Basic
- **jasper** – fornece um interface para o Java
- ...

Para mais detalhes sobre módulos, consulte o *Sicstus User's Manual*.

Declaração de Módulos

É possível declarar novos módulos, colocando no início do ficheiro uma directiva da forma:

:- module(*ModuleName*, *ExportList*).

Nome a dar ao módulo (ModuleName)

Lista dos predicados a exportar pelo módulo (ExportList)

```
:- module(familia, [mae/2,pai/2,avo/2]).

mae(sofia,ana).
mae(ana,maria).

pai(rui,luis).
pai(luis,pedro).
pai(luis,maria).

progenitor(A,B) :- pai(A,B).
progenitor(A,B) :- mae(A,B).

avo(X,Y) :- progenitor(X,Z), progenitor(Z,Y).
```

Exemplos de utilização de módulos

O carregamento / importação de módulos pode ser feita através dos predicados:

```
use_module(library(Package)).
use_module(ModuleName).
use_module(library(Package), ImportList).
```

Exemplo:

```
| ?- use_module(library(random)).
```

```
| ?- random(X).
X = 0.2163110752346893 ?
yes
| ?- random(X).
X = 0.6344657121210742 ?
yes
| ?- random(20,50,X).
X = 31 ?
yes
| ?- randseq(3, 100, L).
L = [24,34,85] ?
yes
```


Exemplo:

```
| ?- use_module(library(lists),[sublist/2,remove_duplicates/2]).
```

```
| ?- remove_duplicates([2,3,4,3,5,6,5,3,2],L).
L = [2,3,4,5,6] ?
yes
| ?- append([1,2,3],[6,7],L).
! Existence error in user: : /2
! procedure library(_95):append/3 does not exist
! goal:  library(_95):append([1,2,3],[6,7],_93)
| ?- sublist(S,[1,2,3]).
S = [1,2,3] ? ;
S = [2,3] ? ;
S = [3] ? ;
S = [] ? ;
S = [2] ? ;
S = [1,3] ? ;
S = [1] ? ;
S = [1,2] ? ;
no
```

Exemplo: Considere que o seguinte programs está gravado no ficheiro *exemplo.pl*

```
:- use_module(familia).
:- use_module(library(lists),[append/3, member/2,
                              remove_duplicates/2]).

avos([],[]).
avos([H|T],L) :- findall(A,avo(A,H),L1), avos(T,L2),
                 append(L1,L2,L3), remove_duplicates(L3,L).

ocorre(_,[],[]).
ocorre(X,[L|T],[L|L1]) :- member(X,L), !, ocorre(X,T,L1).
ocorre(X,[L|T],L1) :- \+ member(X,L), ocorre(X,T,L1).
```

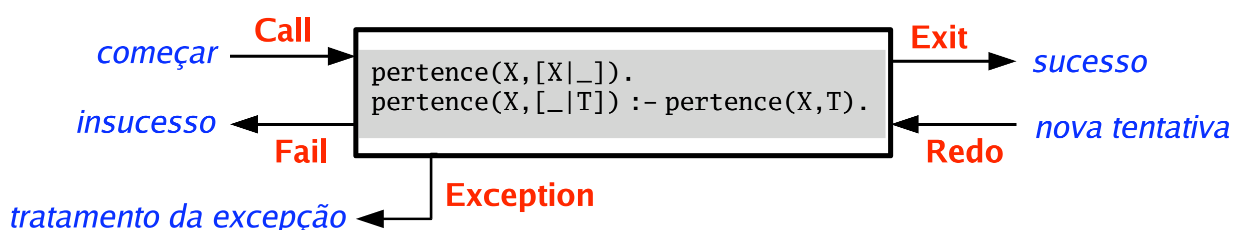
```
| ?- [exemplo].
| ?- avos([pedro,maria],X).
X = [rui,sofia] ?
yes
| ?- progenitor(Z,pedro).
! Existence error in user: : /2
! procedure library(_84):progenitor/2 does not exist
! goal:  library(_84):progenitor(_81,pedro)
| ?- ocorre(2,[[3,2,4],[4,5],[a,d,2,e],[4,3,3,a]],L).
L = [[3,2,4],[a,d,2,e]] ?
yes
```


Debugging

O interpretador Prolog possui um mecanismo de *debug*, que permite ter informação sobre os vários passos de execução (de prova) de um objectivo de prova.
A utilização deste mecanismo pode ser uma ferramenta preciosa na detecção e correcção de erros.

Uma forma de visualizar o fluxo de control de uma execução (incluindo o *backtracking*), é ver cada predicado/procedimento como uma caixa com as seguintes *"portas"*:

Call – lança um predicado para ser provado;
Exit – termina a prova do predicado com sucesso;
Fail – não consegue fazer a prova do predicado;
Redo – tenta construir uma nova prova para o predicado, forçada por backtracking;
Exception – ocorreu uma excepção.



Tracing

A base do mecanismo de *debug* é a traçagem. A traçagem de um objectivo de prova, vai dando informação sobre os sucessivos passos da construção da prova (quais os predicados que vão sendo invocados e os argumentos da invocação).

Para activar o modo de traçagem faz-se, no interpretador, a invocação do predicado **trace**. Quando o trace está activo o interpretador pára sempre uma das *"portas"* (do slide anterior) é activada.

```
| ?- trace.
% The debugger will first creep -- showing everything (trace)
yes
| ?- pertence(2,[1,2,3]).
        1      1 Call: pertence(2,[1,2,3]) ?
        2      2 Call: pertence(2,[2,3]) ?
?       2      2 Exit: pertence(2,[2,3]) ?
?       1      1 Exit: pertence(2,[1,2,3]) ?
yes
```


Alguns comandos úteis do modo *trace*:

h	help	lista todos os comandos disponíveis.
c	creep	avança mais um passo na prova (basta fazer *enter*).
l	leap	avança sempre até encontrar um *spypoint.*
s	skip	válido só para a porta Call ou Redo, faz com que não se desça mais no detalhe da prova desse *subgoal.*
f	fail	força a falha do *subgoal.* Passa o controle para a porta Fail.
r	retry	passa de novo o controle para a porta Call.
a	abort	Aborta a prova do predicado (e da traçagem).

...

Para mais informações sobre debugging consulte o *Sicstus User's Manual*.

Para sair do modo de traçagem invoque, no interpretador, o prediado **notrace**.

```
| ?- notrace.
% The debugger is switched off
yes
```

Exemplos:

```
pertence(X,[X|_]).
pertence(X,[_|T]) :- pertence(X,T).
```

```
| ?- pertence(X,[1,2,3]).
        1      1 Call: pertence(_430,[1,2,3]) ? c
?       1      1 Exit: pertence(1,[1,2,3]) ?
X = 1 ? ;
        1      1 Redo: pertence(1,[1,2,3]) ? c
        2      2 Call: pertence(_430,[2,3]) ?
?       2      2 Exit: pertence(2,[2,3]) ?
?       1      1 Exit: pertence(2,[1,2,3]) ?
X = 2 ? ;
        1      1 Redo: pertence(2,[1,2,3]) ? s
?       1      1 Exit: pertence(3,[1,2,3]) ?
X = 3 ? ;
        1      1 Redo: pertence(3,[1,2,3]) ?
        2      2 Redo: pertence(3,[2,3]) ?
        3      3 Call: pertence(_430,[]) ?
        3      3 Fail: pertence(_430,[]) ?
        2      2 Fail: pertence(_430,[2,3]) ?
        1      1 Fail: pertence(_430,[1,2,3]) ?
no
```



```
| ?- pertence(X,[1,2,3]).
        1      1 Call: pertence(_430,[1,2,3]) ?
?       1      1 Exit: pertence(1,[1,2,3]) ?
X = 1 ? ;
        1      1 Redo: pertence(1,[1,2,3]) ? s
?       1      1 Exit: pertence(2,[1,2,3]) ?
X = 2 ? ;
        1      1 Redo: pertence(2,[1,2,3]) ? f
        1      1 Fail: pertence(_430,[1,2,3]) ?
no
| ?- pertence(X,[1,2,3]).
        1      1 Call: pertence(_430,[1,2,3]) ? l
X = 1 ? ;
X = 2 ? ;
X = 3 ? ;
no
```

Exemplo:

```
isort([],[]).
isort([H|T],L) :- isort(T,T1), ins(H,T1,L).

ins(X,[],[X]).
ins(X,[Y|Ys],[Y|Zs]) :- X > Y, ins(X,Ys,Zs).
ins(X,[Y|Ys],[X,Y|Ys]) :- X =< Y.
```

```
| ?- isort([3,2,8],L).
        1      1 Call: isort([3,2,8],_476) ?
        2      2 Call: isort([2,8],_1010) ? s
?       2      2 Exit: isort([2,8],[2,8]) ?
        3      2 Call: ins(3,[2,8],_476) ?
        4      3 Call: 3>2 ?
        4      3 Exit: 3>2 ?
        5      3 Call: ins(3,[8],_2316) ?
        6      4 Call: 3>8 ?
        6      4 Fail: 3>8 ?
        7      4 Call: 3=<8 ?
        7      4 Exit: 3=<8 ?
        5      3 Exit: ins(3,[8],[3,8]) ?
?       3      2 Exit: ins(3,[2,8],[2,3,8]) ?
?       1      1 Exit: isort([3,2,8],[2,3,8]) ?
L = [2,3,8] ? ;
        1      1 Redo: isort([3,2,8],[2,3,8]) ?
        3      2 Redo: ins(3,[2,8],[2,3,8]) ?
        8      3 Call: 3=<2 ?
        8      3 Fail: 3=<2 ?
        3      2 Fail: ins(3,[2,8],_476) ?
        2      2 Redo: isort([2,8],[2,8]) ?
        2      2 Fail: isort([2,8],_1010) ?
        1      1 Fail: isort([3,2,8],_476) ?
no
```


Exercícios:

1. Use a traçagem para confirmar a construção das árvores de procura que foram apresentas ao longo dos slides anteriores.

2. A seguinte definição pretende contar o número de ocorrências de um elemento numa lista, usando um parâmetro de acumulação.

```
conta_errado(X,L,N) :- contaAC(X,L,0,N).

contaAC(_,[],Ac,Ac).
contaAC(X,[H|T],Ac,N) :- X==H, Ac1 is Ac+1, contaAC(X,T,Ac1,N).
contaAC(X,[_|T],Ac,N) :- contaAC(X,T,Ac,N).
```

Mas este predicado não está correctamente definido. Por exemplo:

```
| ?- conta_errado(3,[3,2,3,4],N).
N = 2 ? ;
N = 1 ? ;
N = 1 ? ;
N = 0 ? ;
no
```

Faça debugging deste predicado para detectar o erro, e corrija-o.

Manipulação da Base de Conhecimento

Um programa Prolog pode ser visto como uma base de dados que especifica um conjunto de relações (de forma explícita através dos factos e implícita através das regras).

O Prolog tem predicados pré-definidos que permitem fazer a manipulação da base de conhecimento. Ou seja, predicados que permitem acrescentar e / ou retirar factos e regras da base de conhecimento, durante a execução de um programa.

Os predicados que estão definidos nos ficheiros que são carregados (por `consult`) são, por omissão, ***estáticos***. Ou seja, é impossível o programa alterar dimâmicamente tais predicados.

Para que seja possível alterar dinâminamente predicados carregados de ficheiro, esses predicados deverão ser declarados como ***dinâmicos*** no ficheiro em que está definido. Isso é feito incluindo no ficheiro a directiva:

```
:- dynamic PredicateName/Arity.
```

Os predicados que não estão, à partida, definidos são considerados dinâmicos

Adicionar factos e regras

Para adicionar factos e regras à base de conhecimento podem-se usar os predicados:

assert/1	acrescenta o facto/regra como *último* facto/regra do predicado
asserta/1	acrescenta o facto/regra como *primeiro* facto/regra do predicado
assertz/1	igual a `assert/1`

Exemplos:

amigos.pl

```
:- dynamic amigos/2.
:- dynamic mas/1.

amigos(ana,rui).
amigos(pedro,rui).
amigos(maria,helena).
amigos(pedro,ana).

mas(rui).
mas(pedro).

fem(ana).
fem(maria).
fem(helena).
```

```
| ?- ['amigos.pl'].
% consulting ...
yes
| ?- asserta(amigos(joao,helena)).
yes
| ?- assertz(mas(joao)).
yes
| ?- assertz(fem(isabel)).
! Permission error: cannot assert static user:fem/1
! goal:  assertz(user:fem(isabel))

| ?- assertz((amiga(X,Y) :- amigos(X,Y), fem(X))).
true ?
yes
| ?- assert((amiga(X,Y) :- amigos(Y,X), fem(X))).
true ?
yes
```

```
| ?- listing.
amiga(A, B) :- amigos(A, B), fem(A).
amiga(A, B) :- amigos(B, A), fem(A).

amigos(joao, helena).
amigos(ana, rui).
amigos(pedro, rui).
amigos(maria, helena).
amigos(pedro, ana).

fem(ana).
fem(maria).
fem(helena).

mas(rui).
mas(pedro).
mas(joao).

yes
| ?-
```

Se quiser ver apenas alguns predicados pode usar o `listing/1`.

Exemplos:

```
| ?- listing(fem).
```

```
| ?- listing(amigos/2).
```

```
| ?- listing([amiga,mas/1]).
```

Note que *este programa é volátil*. Quando sair do interpretador os novos factos e regras perdem-se.

Remover factos e regras

Para remover factos e regras da base de conhecimento podem-se usar os predicados:

`retract/1` remove da base de conhecimento a *primeira* cláusula (facto ou regra) que unifica com o termo que é passado como parâmetro.

`retractall/1` remove da base de conhecimento *todos* os factos ou regras cuja **cabeça** unifique com o termo que é passado como parâmetro.

`abolish/1` remove da base de conhecimento *todos* os factos e regras com o functor/aridade que é passada como parâmetro.

`abolish/2` semelhante a `abolish/1`, mas passando o nome do functor e a sua aridade separadamente.

Exemplo:

Considere o novo ficheiro

AMIGOS.pl

```
:- dynamic amigos/2, mas/1.

amigos(ana,rui).
amigos(pedro,rui).
amigos(maria,helena).
amigos(pedro,ana).
amigos(X,Y) :- amigos(Y,X).

mas(rui).
mas(pedro).

fem(ana).
fem(maria).
fem(helena).
```

```
| ?- ['AMIGOS'].
% consulting ...
| ?- retract(amigos(_,rui)).
yes
| ?- retract(fem(_)).
! Permission error: cannot retract static user:fem/1
! goal:  retract(user:fem(_79))
| ?- abolish(fem/1).
yes
| ?- listing.
amigos(pedro, rui).
amigos(maria, helena).
amigos(pedro, ana).
amigos(A, B) :- amigos(B, A).

mas(rui).
mas(pedro).
yes
| ?- retractall(amigos(pedro,_)).
yes
| ?- listing.
amigos(maria, helena).

mas(rui).
mas(pedro).
yes
| ?- abolish(mas,1).
yes
```

Nota: abolish remove mesmo predicados estáticos !

Exercícios:

1. A informação referente aos horários das salas de aula pode estar guardada na base de conhecimento em factos da forma sala(*num,dia,inicio,fim,discipl,tipo*)

```
:- dynamic sala/6.

sala(cp1103, seg, 10, 13, aaa, p).
sala(cp2301, ter, 10, 11, aaa, t).
sala(di011, sab, 12, 10, xxx, p).  % com erro
sala(cp3204, dom, 8, 10, zzz, p).
sala(di011, sex, 14, 16, xxx, p).
sala(cp204, sab, 15, 17, zzz, tp).
sala(di011, qui, 14, 13, bbb, tp). % com erro
sala(di104, qui, 9, 10, aaa, tp).
sala(dia1, dom, 14, 16, bbb, t).
sala(cp1220, sab, 14, 18, sss, p).
```

a) O seguinte predicado, permite retirar da base de dados todas marcações de sala em que, erradamente, a hora de início da aula é superior à hora de fim.

```
apaga_erros :- sala(N,D,Hi,Hf,C,T), Hf =< Hi,
               retract(sala(N,D,Hi,Hf,C,T)), fail.
apaga_erros.
```

Qual é o efeito da segunda clausula apaga_erros ?

b) Execute um programa que retire todas as marcações de salas para os domingos.

c) Execute um programa que retire todas as marcações de salas para os sábados depois das 13 horas.

d) Defina o predicado `ocupada(+NSala,+Dia,+Hora)` que sucede se a sala número `Nsala` está ocupada no dia `Dia` à hora `Hora`.

e) Defina o predicado `livre(?NSala,+Dia,+Hora)` que sucede se a sala número `NSala` está livre no dia `Dia` à hora `Hora`.

f) Defina o predicado `livres(-Lista,+Dia,+Hora)` que sucede se `Lista` é a lista dos números das salas que estão livres no dia `Dia` à hora `Hora`.

g) Defina o predicado `total_ocupacao(+NSala,-Total)` que sucede se `Total` é o número total de horas que a sala `NSala` está ocupada.

h) Defina o predicado `cria_total(+NSala)` que acrescenta à base de dados um facto, associando à sala `NSala` o número total horas de ocupação dessa sala por semana.

i) Defina o predicado `intervalo_livre(?NSala,+Dia,+Inicio,+Fim)` que sucede se `NSala` está livre no dia `Dia` durante o periodo de tempo entre `Inicio` e `Fim`.

2. Assuma que a informação referente às notas dos alunos à disciplina de *Lógica Computacional 2005/06* está guardada na base de conhecimento em factos da forma:

modalidadeA(*numero, nome, fichas, exame*)
modalidadeB(*numero, nome, fichas, trabalho, exame*)

Por exemplo:

```
modalidadeA(1111, 'Maria Campos' , 14, 12).
modalidadeA(3333, 'Rui Silva', 13, 15).
modalidadeA(4444, 'Paulo Pontes', 17, 12).
modalidadeA(8888, 'Antonio Sousa', 14, 8).

modalidadeB(2222, 'Ana Miranda', 14, 15, 12).
modalidadeB(5555, 'Joao Ferreira', 15, 16, 11).
```

a) Escreva o predicado `gera_notas/0` que cria factos `nota/3`, que definem a relação entre o aluno (número e nome) e a sua nota final (calculada de acordo com o estabelecido para esta disciplina).

b) Defina o predicado `aprovados(-Lista)` que sucede se `Lista` contem o número dos alunos aprovados à disciplina.

3. Assuma que para implementar uma agenda electrónica temos na base de conhecimento factos com a seguinte informação agenda(*data, horaInicio, horaFim, tarefa, tipo*). Por exemplo:

```
agenda(data(5,abril,2006), 9, 13, join, palestras).
agenda(data(6,abril,2006), 11, 13, join, palestras).
agenda(data(6,abril,2006), 16, 17, logcomp, aulas).
agenda(data(6,abril,2006), 17, 20, atendimento, aulas).
agenda(data(4,abril,2006), 15, 17, di, reuniao).
agenda(data(7,abril,2006), 8, 13, logcomp, aulas).
agenda(data(7,abril,2006), 15, 17, ccc, reuniao).
agenda(data(4,maio,2006), 11, 13, pure, palestras).
```

a) Defina o predicado `cria_tipo(+Tipo)` que consulta a agenda e gera factos do nome do tipo com a lista de pares *(data,tarefa)* associados a esse tipo.

b) Defina o predicado `apaga_mes(+Mes)` que apaga todas as marcações de um dado mês.

c) Defina o predicado `marca(+Tarefa,+Tipo,+LDatas)` faz a marcação de uma dada tarefa, de um dado tipo, para uma lista de datas.

Input / Output

Para além dos canais de comunicação usuais com o utilizador (teclado/écran) um programa Prolog pode fazer input/output de informação através de outros canais, por exemplo: ficheiros.

Os canais de entrada de dados chamam-se ***input streams*** e os de saída ***output streams***.

O Sicstus Prolog tem diversos predicados pré-definidos para input/output de caracteres e de termos (ver *User's Manual*). A maioria destes predicados tem duas versões: *com* e *sem* a indicação explícita do *stream*. Quando o *stream* não é indicado, o *input* é do canal de entrada actual *(current input stream)* e o *output* é para o canal de saida actual *(current output stream)*.

No início da execução de um programa, os canais actuais de entrada e de saida são o teclado e o écran, e o nome destes canais é **`user`**. Mas o input e o output pode ser redireccionado através da invocação de predicados apropriados (como veremos).

Input / Output de Caracteres

O Sicstus Prolog tem diversos predicados pré-definidos para input/output de caracteres (ver *User's Manual*):

get_char get_code put_char put_code nl read_line skip_line ...

Exemplos:

```
| ?- get_char(X).
|: b

X = b ? yes
| ?- get_char(X).
|: K

X = 'K' ? yes
| ?- get_char(user,X).
|: 2

X = '2' ? yes
```

```
| ?- get_code(X).
|: A

X = 65 ? yes
| ?- get_code(X).
|: 1

X = 49 ? yes
| ?-
get_code(user,X).
|: z

X = 122 ? yes
```

```
| ?- get_char(a).
|: a

yes
| ?- get_char(a).
|: b

no
| ?- get_code(65).
|: A

yes
```


Exemplos:

```
| ?- put_char(X).
! Instantiation error in argument 1 of put_char/1
! goal:  put_char(_73)
| ?- put_char('X').
X
yes
| ?- put_char(a).
a
yes
| ?- put_char(1).
! Type error in argument 1 of put_char/1
! character expected, but 1 found
! goal:  put_char(1)
| ?- put_char('1').
1
yes
| ?- put_char(user,'a').
a
yes
```

```
| ?- read_line(X).
|: A a 1 Bb
X = [65,32,97,32,49,32,66,98] ?
yes
```

```
| ?- skip_line.
|: isto serve de exemplo
yes
| ?- nl.

yes
```

Exemplos:

```
| ?- put_code(65).
A
yes
| ?- put_code(a).
! Type error in argument 1 of put_code/1
! integer expected, but a found
! goal:  put_code(a)
| ?- put_code(X).
! Instantiation error in argument 1 of put_code/1
! goal:  put_code(_73)
| ?- put_code(32).

yes
| ?- put_code(user,120).
x
yes
```

Existem outros predicados pré-definidos para input/output de caractes. Em particurar, os tradicionalmente usados (embora tendam a cair em desuso):

get, **get0** e **put** são semelhantes a **get_char**, **get_code** e **put_code**, respectivamente.

skip e **ttynl** são semelhantes a **skip_line** e **nl**.
tab(*+N*) escreve *N* espaços; etc ...

Input / Output de Termos

O Sicstus Prolog tem diversos predicados pré-definidos para input/output de termos (ver *User's Manual*):

read(*?Term*) lê o próximo termo da *stream* de entrada e unifica-o com *Term*. Se a *stream* já tiver chegado ao fim *Term* unifica com o átomo **end_of_file**. Note que cada termo tem que ser seguido de ponto final e *enter* ou espaço.

write(*?Term*) escreve o termo *Term* na *stream* de saída.

writeq(*?Term*) escreve o termo *Term* na *stream* de saída, e coloca pelicas sempre que isso seja necessário (para que esse termo possa, posteriormente, ser lido pelo predicado read).

Exemplo:

```
| ?- write('Escreva um número: '), read(N), nl, N1 is N*N,
     write('O quadrado de '), write(N),  write(' é '), write(N1), nl, nl.
Escreva um número: 3.

O quadrado de 3 é 9

N = 3,
N1 = 9 ?
yes
```

`read(T)` faz com que o próximo termo da *input stream* unifique com `T`. Se a unificação não for possível, o predicado falha e não há backtracking para ler outro termo.

Exemplos:

```
| ?- read(X).
|: um exemplo.
! Syntax error in read/1
! ...
| ?- read(X).
|: 'um exemplo'.
X = 'um exemplo' ?
yes
| ?- read(user,X).
|: amanha.
X = amanha ?
yes
| ?- read(X).
|: 'Amanha'.
X = 'Amanha' ?
yes
| ?- read(X).
|: 'um exemplo
    com mais de uma linha'.
X = 'um exemplo\ncom mais de uma linha' ?
yes
```

```
| ?- read(X).
|: [1,2,3,4].
X = [1,2,3,4] ?
yes
| ?- read(user,Z).
|: 1+3*5.
Z = 1+3*5 ?
yes
| ?- read(a).
|: a.
yes
| ?- read(a).
|: b.
no
| ?- read(1+X).
|: 1+3*5.
X = 3*5 ?
yes
| ?- read(X*5).
|: 1+3*5.
no
```


`writeq(T)` coloca pelicas em `T` sempre que necessário. Um termo escito com `writeq` pode depois ser sempre lido com `read`.

Exemplos:

```
| ?- read(X), write(X), nl, writeq(X).
|: 'bom dia'.
bom dia
'bom dia'
X = 'bom dia' ?
yes
| ?- read(X), write(X), nl, writeq(X).
|: olá.
olá
olá
X = olá ?
yes
| ?- read(X), write(X), nl, writeq(X).
|: Olá.
_430
_430
true ?
yes
| ?- read(X), write(X), nl, writeq(X).
|: 'Olá'.
Olá
'Olá'
X = 'Olá' ?
yes
```

Exemplos:

```
| ?- read(data(D,M,A)), write(D), write(' de '), write(M),
     write(' de '), write(A).
|: data(15,abril,2006).
15 de abril de 2006
A = 2006, D = 15, M = abril ?
yes
| ?- read(data(D,M,A)), write(D), write(' de '), name(M,[H|T]),
     H1 is H-32, name(M1,[H1|T]), write(M1),  write(' de '), write(A).
|: data(15,abril,2006).
15 de Abril de 2006
A = 2006, D = 15, H = 97, M = abril, T = [98,114,105,108], H1 = 65, M1 = 'Abril' ?
yes
| ?- read(X), X =.. [horas,H,M], write(H), write(:), write(M).
|: horas(10,25).
10:25
H = 10, M = 25, X = horas(10,25) ?
yes
```

O Sicstus Prolog tem muitos outros predicados de IO, como por exemplo o predicado `format` que tem semellhanças com o *`printf` do C* (ver *User's Manual*).

`format`(*`+Format`*, *`:Arguments`*) escreve no output stream de acordo com *`Format`* e a lista de argumentos *`Arguments`*.

Exemplos:

```
| ?- read(horas(H,M)), format('São ~d horas e ~d minutos!',[H,M]).
|: horas(9,45).
São 9 horas e 45 minutos!
H = 9,  M = 45 ?
yes
| ?- read(X), X =.. L, format('São ~i~d horas e ~d minutos!',L).
|: time(15,30).
São 15 horas e 30 minutos!
L = [time,15,30],  X = time(15,30) ?
yes
| ?- format('Escreva um número: ',[]), read(N), Q is N*N,
     format('O quadrado de ~2f é ~2f \n\n',[N,Q]).
Escreva um número: 2.3.
O quadrado de 2.30 é 5.29

N = 2.3,  Q = 5.289999999999999 ?
yes
| ?- format('Olá, ~a ~s !', [muito,"bom dia"]).
Olá, muito bom dia !
yes
| ?- format('Aqui ~a um ~20s!!!', ['está mais',"bom exemplo"]).
Aqui está mais um bom exemplo         !!!
yes
```

```
conversoes :- write('Escreva uma lista de números (ou fim para terminar):'),
              nl, read(L), processa(L).

processa(fim) :- !.
processa(L) :- format('\nCelcius \tFarenheit \tKelvin\n',[]),
               format('~40c\n',[0'-]), converte(L).

converte([]) :- nl, conversoes.
converte([C|T]) :- F is C * 1.8 + 32, K is C + 273,
                   format('~2f \t\t~2f \t\t~2f\n',[C,F,K]), converte(T).
```

Exemplo:

Este programa lê uma lista de temperaturas em graus Celcius e gera uma tabela com o resultado das conversões.

```
| ?- conversoes.
Escreva uma lista de números (ou fim para terminar):
|: [27.4,32.75,100].

Celcius        Farenheit      Kelvin
----------------------------------------
27.40          81.32          300.40
32.75          90.95          305.75
100.00         212.00         373.00

Escreva uma lista de números (ou fim para terminar):
|: [0, -15.4, -28.2, 12.7].

Celcius        Farenheit      Kelvin
----------------------------------------
0.00           32.00          273.00
-15.40         4.28           257.60
-28.20         -18.76         244.80
12.70          54.86          285.70

Escreva uma lista de números (ou fim para terminar):
|: fim.
yes
```


Exercícios:

1. Defina o predicado `tabuada(+N)` que dado um número inteiro N, apresenta no écran a tabuada do N.

2. Escreva um programa `escreve_tabuadas` que lê um inteiro do teclado, escreve no écran a sua tabuada, e continua pronto para escrever tabuadas até que seja mandado terminar.

3. Escreva um programa que lê uma lista de pares *(átomo, nº inteiro)* e apresenta um gráfico de barras dessa lista de pares. (Cada unidade deve ser representada pelo caracter #, e as barras podem ser horizontais.)

4. Escreva um programa que lê os coeficientes de um polinómio de 2º grau $a x^2 + b x + c$ e calcula as raizes reais do polinómio, apresentando-as no écran. Se o polinómio não tiver raizes reais, o programa deve informar o utilizador desse facto.

Ficheiros

Os ficheiros são vistos como *streams*. A *stream* corresponde ao descritor do ficheiro (ao nível do sistema operativo.)

Um ficheiro pode ser aberto para leitura ou escrita através do predicado **open**. Este predicado devolve o descritor do ficheiro que pode depois ser passado como argumento dos predicados de I/O. Para fechar o ficheiro usa-se o predicado **close**.

Existem predicados para saber informação sobre as streams existentes.

O Sicstus Prolog tem um vasto conjunto e predicados para manilupação de streams (ver *User's Manual*). Ficam aqui alguns exemplos:

open(*+FileName*,*+Mode*,*-Stream*) abre o ficheiro *FileName* em modo *Mode* (pode ser: read, write ou append). *Stream* fica como descritor desse ficheiro.

set_input(*+Stream*) torna *Stream* o canal actual de entrada.
set_output(*+Stream*) torna *Stream* o canal actual de saída.
current_input(*?Stream*) *Stream* é o canal actual de entrada.
current_output(*?Stream*) *Stream* é o canal actual de saída.
current_stream(*?FileName*,*?Mode*,*?Stream*) serve para saber informação sobre as *Streams*.

flush_output(*+Stream*) descarrega o *buffer* do canal *Stream*.
close(*+Stream*) fecha o canal *Stream*.

Exemplo:

```
iniciaConv :- open('CelFar.txt',append,CF),
              open('CelKel.txt',write,CK), convFich(CF,CK).

convFich(CF,CK) :- write('Escreva uma lista de números (ou fim.):'),
                   nl, read(user,L), trata(L,CF,CK).

trata(fim,CF,CK) :- close(CF), close(CK), !.
trata(L,CF,CK) :- format(CF,'\nCelcius \tFarenheit\n~25c\n',[0'-]),
                  format(CK,'\nCelcius \tKelvin\n~22c\n',[0'-]),
                  converteF(L,CF,CK).

converteF([],CF,CK) :- nl(user), convFich(CF,CK).
converteF([C|T],CF,CK) :- F is C*1.8+32, format(CF,'~2f \t\t~2f\n',[C,F]),
                          K is C+273, format(CK,'~2f \t\t~2f\n',[C,K]),
                          converteF(T,CF,CK).
```

Note que o ficheiro *CelFar.txt* é aberto em modo append, enquanto o ficheiro *CelKel.txt* é aberto em modo write.

```
| ?- iniciaConv.
Escreva uma lista de números (ou fim.):
|: [27.4,32.75,100].

Escreva uma lista de números (ou fim.):
|: fim.
yes

| ?- iniciaConv.
Escreva uma lista de números (ou fim.):
|: [55, 23.6].

Escreva uma lista de números (ou fim.):
|: [-10.8, 0, 20.2].

Escreva uma lista de números (ou fim.):
|: fim.
yes
```

CelFar.txt

```
Celcius                 Farenheit
-------------------------------
27.40                   81.32
32.75                   90.95
100.00                  212.00

Celcius                 Farenheit
-------------------------------
55.00                   131.00
23.60                   74.48

Celcius                 Farenheit
-------------------------------
-10.80                  12.56
0.00                    32.00
20.20                   68.36
```

CelKel.txt

```
Celcius                 Kelvin
--------------------------
55.00                   328.00
23.60                   296.60

Celcius                 Kelvin
--------------------------
-10.80                  262.20
0.00                    273.00
20.20                   293.20
```


tell, telling, told ... see, seeing, seen

São os predicados Prolog tradicionalmente usados na manipulação de ficheiros.

tell(*+File*) — abre *File* para escrita e torna-o canal de saída actual. Se *File* já tiver aberta apenas o torna canal de saída actual.

telling(*?FileName*) — unifica *FileName* com o nome do ficheiro de saída actual se este foi aberto com tell, senão unifica com user.

told — fecha o canal actual de saída (e este volta a ser user).

see(*+File*) — abre *File* para leitura e torna-o canal de entrada actual. Se *File* já tiver aberta apenas o torna canal de entrada actual.

seeing(*?FileName*) — unifica *FileName* com o nome do ficheiro de entrada actual se este foi aberto com see, senão unifica com user.

seen — fecha o canal de entrada actual (e este volta a ser user).

Exemplos:

```
| ?- see('teste.txt'), read(T), seeing(X), seen.
! Existence error in argument 1 of see/1
! file 'teste.txt' does not exist
! goal:  see('teste.txt')
| ?- tell('teste.txt'),
     writeq('Isto é uma experiencia!'), write('.'), nl,
     telling(X), told.
X = 'teste.txt' ?
yes
| ?- see('teste.txt'), read(T1), read(T2), seeing(X), seen.
X = 'teste.txt',
T1 = 'Isto é uma experiencia!',
T2 = end_of_file ?
yes
| ?- seeing(X), telling(Y).
X = user, Y = user ?
yes
```


Exemplos:

```
| ?- tell(aaa), format('Uma ~a experiencia!',[nova]), telling(X), told.
X = aaa ?
yes
| ?- see(aaa), get_char(C1), get_char(C2).
C1 = 'U',
C2 = m ?
yes
| ?- get_char(C3), seeing(X).
X = aaa,
C3 = a ?
yes
| ?- seen, get_char(C4).
|: z

C4 = z ?
yes
```

```
| ?-  open(aaa,read,A), seeing(X), get_char(C), get_char(A,K), close(A).
|: z

A = '$stream'(4085968),
C = z,
K = 'U',
X = user ?
yes
```

Os predicados `telling` e `seeing` podem ser usados para recolher informação sobre os canais de saída e de entrada (num dado momento) de forma a mais tarde se conseguir repôr o mesmo contexto de comunicação. Usam-se as combinações:

```
telling(F), tell(file), ... , told, tell(F)
```

```
seeing(F), see(file), ... , seen, see(F)
```

Exemplo:

```
| ?- tell(fam), write('pai(rui,carlos).'), nl.
yes
| ?- write(pai(pedro,hugo)), write('.'), nl.
yes
| ?- telling(F), tell(ami),
     write('amigos(ana,rui).'), nl, write('amigos(hugo,ana).'), nl,
     told, tell(F).
F = fam ?
yes
| ?- write('pai(paulo,ricardo).'), nl, write('pai(manuel,helena).'), nl.
yes
| ?- telling(X), told.
X = fam ?
yes
```

Produz os ficheiros

fam

```
pai(rui,carlos).
pai(pedro,hugo).
pai(paulo,ricardo).
pai(manuel,helena).
```

ami

```
amigos(ana,rui).
amigos(hugo,ana).
```


"Repeat loops"

O Sicstus Prolog tem pré-definido o predicado **repeat**, que é um predicado de controlo que permite fazer o controlo do backtracking, e que deve ser usado em combinação com o **cut** para gerar ciclos.

O esquema geral de programação com *repeat loops* é o seguinte:

```
Head :- ...,
        repeat,
          generate(Datum),
          action(Datum),
          test(Datum),
        !,
        ...
```

O seu propósito é repetir uma determinada *acção* sobre os elementos que vão sendo *gerados* (por exemplo, que vão sendo lidos de uma *stream*) até que uma determinada condição de *teste* seja verdadeira. Note que estes ciclos só têm interesse se envolverem efeitos laterais.

A utilização típica de *repeat loops* é no processamento de informação lida de ficheiros e na interacção com o utilizador (na implementação de menus e na validação dos dados de entrada).

Processamento de ficheiros

O processamento de ficheiros baseia-se, normalmente, na repetição da leitura até que se encontre o fim de ficheiro. Obedece normalmente a um dos seguintes esquemas:

Usando apenas a recursão

```
processfile(F) :- seeing(S), see(F),
                  read(Term), process(Term),
                  seen, see(S).

process(end_of_file) :- !.
process(Term) :- treat(Term), read(T),
                 process(T).
```

Usando *repeat loops* (mais eficiente)

```
processfile(F) :- seeing(S), see(F),
                  repeat,
                    read(T),
                    process(T),
                    T == end_of_file,
                  !,
                  seen, see(S).

process(end_of_file).
process(Term) :- treat(Term).
```

Nota: `treat` deverá fazer o processamneto do termo actal.

Interacção com o utilizador

Exemplo: Implementação de um menu.

```
menu :- repeat,
          write('==== MENU ===='), nl,
          write('1. - Opção A'), nl,
          write('2. - Opção B'), nl,
          write('0. - Sair'), nl,
          read(X),
          opcao(X),
          X==0,
        !.

opcao(0) :- !.
opcao(1) :- write('Escolheu a opção A ...'), nl, !.
opcao(2) :- write('Escolheu a opção B ...'), nl, !.

opcao(_) :- write('Opção inválida!'), nl, !.
```

Exemplo: Validar a entrada de dados.

```
le_int(X) :- repeat,
               read(X),
               integer(X),
             !.
```

```
| ?- le_int(X).
|: abc.
|: 1.4.
|: 3.
X = 3 ?
yes
```

Exercícios:

1. Relembre o problema apresentado anteriormente, em que a informação referente aos horários das salas de aula está guardada na base de conhecimento em factos da forma: `sala`(*num,dia,inicio,fim,discipl,tipo*).
   Defina um predicado `salva(+Ficheiro)` que guarda no `Ficheiro` os factos com a informação sobre as salas que são válidas (i.e., em que a hora de início é inferior à hora de fim).

2. Defina o predicado `findterm(+Term)` que escreve o écran o primeiro termo lido que unifica com `Term`.

3. Defina o predicado `findalltermsfile(+Term,+FileName)` que escreve no écran todos os termos do ficheiro que unificam com `Term` (garanta que `Term` não é instanciado).

4. Defina o predicado `to_upper(+FileIn,+FileOut)` que recebe um ficheiro de texto `FileIn` e gera o ficheiro `FileOut` com o mesmo texto de entrada mas convertido para letras maiúsculas. (Note que apenas as letras minúsculas são alteradas, o resto deverá ser mantido.)

5. Defina o predicado `numera_linhas(+FileIn,+FileOut)` que recebe o ficheiro `FileIn` e produz o ficheiro `FileOut`, que contém as mesmas linhas de `FileIn`, mas com as linhas numeradas.

6. Relembre o problema do cálculo da nota final à disciplina de *Lógica Computacional*, apresentado anteriormente, em que as notas do alunos estão guardadas na base de conhecimento em factos da forma:

   `modalidadeA`(*numero, nome, fichas, exame*)
   `modalidadeB`(*numero, nome, fichas, trabalho, exame*)

   Pretende-se agora poduzir a pauta final, tendo a informação sobre os alunos inscritos num ficheiro com factos da forma: `aluno`*(numero, nome, tipo)*

   Defina o predicado `gera_pauta(+Inscritos,+Pauta)` que dado o ficheiro `Inscritos`, vai lendo a informação sobre os alunos inscritos à disciplina e, sem alterar a base de conhecimento, produz no ficheiro `Pauta` o texto com a pauta devidamente preenchida. Note que:

   -- se o aluno está inscrito mas não se submeteu a avaliação, a sua nota será Faltou;

   -- se o aluno tem alguma das componentes de avaliação inferior a 9, ou a média final arredondada inferior a 10, a sua nota será Reprovado;

   -- nos restantes casos, será o arredondamento da média pesada (se quiser, pode escrever também a nota por extenso).

Exemplos de Programação em Prolog

Gerar e Testar é uma técnica muito usada na progamação em Prolog.

Na procura de soluções para um dado problema, um predicado **gera** uma possível solução e outro predicado **testa** se a solução cadidata verifica os requesitos impostos pelo problema (ou seja, é efectivamente uma solução).

```
procura(X) :- gera(X), testa(X).
```

Se o teste falhar novas soluções são geradas pelo mecanismo de backtracking.

Exemplo: Testar se duas listas se intersectam (i.e. se têm um elemento comum).

```
intersecta(Xs,YS) :- member(X,Xs), member(x,Ys).
```

Aqui o 1º `member` gera um elemento da lista `Xs` e o 2ª `member` testa se esse elemento está em `Ys`.

O problema das N rainhas

Colocar N rainhas num tabuleiro de xadrez NxN de nodo a que nenhuma rainha possa ser capturada.

Podemos modelar este problema de várias maneiras.

Solução 1: Representar cada rainha por um par com as suas cpordenadas (X,Y). A solução final é dada por uma lista de N rainhas que não se atacam.

Dado que, para não se atacarem, as rainhas terão de estar colocadas em colunas distintas, podemos fixar as coordenadas X. A solução terá a seguinte forma

```
[(1,Y1), (2,Y2), ..., (N,YN)]
```

O problema reduz-se agora a em encontrar instâncias deste padrão que não se ataquem.

```
rainhas1(N,L) :- template(1,N,L), solucao1(N,L).

template(N,N,[(N,_)]).
template(M,N,[(M,_)|L]) :- M<N, M1 is M+1, template(M1,N,L).
```

```
rainhas1(N,L) :- template(1,N,L), solucao1(N,L).

template(N,N,[(N,_)]).
template(M,N,[(M,_)|L]) :- M<N, M1 is M+1, template(M1,N,L).

solucao1(_,[]).
solucao1(N,[(X,Y)|Resto]) :- solucao1(N,Resto),
                             entre(1,N,L),
                             member(Y,L),              % gera
                             naoataca((X,Y),Resto).   % testa

entre(M,N,[M|L]) :- M<N, M1 is M+1, entre(M1,N,L).
entre(N,N,[N]).

naoataca(_,[]).
naoataca((X,Y),[(X1,Y1)|Resto]) :- X =\= X1, Y =\= Y1,
                                   X-X1 =\= Y-Y1, X-X1 =\= Y1-Y,
                                   naoataca((X,Y),Resto).
```

Exemplo:

```
| ?- rainhas1(4,S).
S = [(1,3),(2,1),(3,4),(4,2)] ? ;
S = [(1,2),(2,4),(3,1),(4,3)] ? ;
no
```


Solução 2: Coordenada X de cada rainha dada pela posição na lista.
A solução final é uma permutação da lista [1,2,...,N] sem “ataques”.

Dado que uma solução para este problema tem necessariamente que colocar cada rainha numa coluna diferente (e isso era dado à partida na resolução 1), podemos omitir a informação sobre as coordenadas X: ela será dada pela posição na lista.

Uma representação mais económica é representar o tabuleiro como a lista das coordenadas Y das rainhas: [Y1,Y2,...,YN]

```
  1ª coluna   2ª coluna          Nª coluna
      ↓           ↓                  ↓
[   linha  ,   linha  ,   ...  ,   linha   ]
```

Para evitar ataques horizontais, as rainhas não podem estar numa mesma linhas. Isto impõem restrições às coordenadas Y: as N rainhas têm que ocupar N linhas diferentes.

O problema reduz-se então ao problema de *encontrar uma permutação da lista [1,2,...,N] em que não haja ataques diagonais*.

```
rainhas2(N,S) :- entre(1,N,L),
                 permutation(L,S),      % gera
                 segura(S).             % testa

segura([Y|Ys]) :- segura(Ys), \+ ataca(Y,Ys).
segura([]).

% uma rainha ataca outra a uma distância de N colunas, se esta segunda tiver
% uma coordenada-Y que é maior ou menor N unidades (casas) do que a primeira
% rainha

ataca(R,L) :- ataca(R,1,L).

ataca(R,N,[Y|_]) :- R is Y+N ; R is Y-N.
ataca(R,N,[_|Ys]) :- N1 is N+1, ataca(R,N1,Ys).
```

Exemplo:

```
| ?- rainhas2(8,S).
S = [1,5,8,6,3,7,2,4] ? ;
S = [1,6,8,3,7,4,2,5] ? ;
S = [1,7,4,6,8,2,5,3] ? ;
S = [1,7,5,8,2,4,6,3] ?
...
```


Exercícios:

1. Construa uma 3ª solução para o problema das N rainhas, que em vez de testar a permutação completa (i.e. colocar todas as rainhas e depois testar) teste cada rainha à medida que a coloca no tabuleiro. Note que a solução final é cronstruida utilizando um acumulador.
   A solução a apresentada continua a usar a técnica de gerar e testar ?

2. Defina um predicado `gnat(+N,?X)` para gerar, por backtracking, números naturais sucessivos até N.

3. Usando o predicado anterior, escreva predicado `raiz(+N,?I)` para calcular a raiz quadrada inteira de um número natural N, definido como sendo o número I tal que $I^2 \leq N$ e $(I+1)^2 > N$.

O problema de colorir um mapa

Colorir uma mapa de modo a que regiões vizinhas não tenham a mesma cor (sabe-se que 4 cores são suficientes para colorir qualquer mapa).

Instanciar uma *estrutura de dados desenhada especialmente para um problema* é um meio eficaz de implementar soluções de gerar e testar. O controlo da construção da solução final é feito pela unificação.

Solução: Implementar o seguinte algoritmo (suportado por uma estrutura de dados adquada)

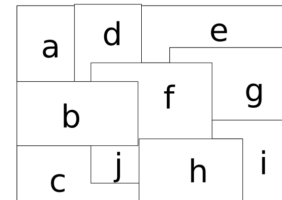
Para cada região do mapa:
    escolher uma cor;
    escolher (ou verificar) as cores para as regiões vizinhas, das cores que sobram.

Estrutura de dados:

```
[ regiao(nome,cor,lista de cores dos vizinhos), ...]
```

Exemplo:

```
[ regiao(a,A,[D,B]), regiao(b,B,[A,D,F,H,J,C]),
  regiao(c,C,[B,J,H]), regiao(d,D,[A,E,F,B]),
  regiao(e,E,[D,F,G]), ... ]
```



O seguinte programa usa a técnica de geração e teste.

```
color_map([],_).
color_map([R|Rs],Cores) :- color_region(R,Cores),
                           color_map(Rs,Cores).

color_region(regiao(Nome,Cor,Vizinhos),Cores) :- select(Cor,Cores,Cores1),
                                                  members(Vizinhos,Cores1).

members([X|Xs],Ys) :- member(X,Ys), members(Xs,Ys).
members([],_).
```

A partilha de variáveis, na estrutura de dados de suporte, assegura que a mesma região não possa ser colorida com cores diferentes, nas diversas iterações do algoritmo.

Exercício: Pretende-se construir um programa que permita testar esta solução de coloração de mapas.

1. Defina na base de conhecimento exemplos de mapas (associe a cada mapa um nome).
2. Defina o predicado `colorir(?Nome,+Cores,?Pintura)` que dado o nome do mapa e a lista de cores a usar, produz em `Pintura` a associação entre dada região do mapa e a respectiva cor.
3. Analise o funcionamento do programa fazendo a traçagem da sua execução.

Puzzles lógicos

A resolução de puzzles lógicos pode ser feita muito facilmente utilizando a técnica de gerar e testar.

O puzzle descreve uma determinada situação. Essa descrição consiste num conjunto de factos acerca de um conjunto de entidades. Essas entidades têm um certo número de atributos. Com as informações dadas deve ser possivel atribuir (de forma unívoca) os atributos às entidades envolvidas no puzzle.

O desafio do puzzle é responder a algumas questões sobre a situação descrita que não é explítica na apresentação do problema.

Este tipo de puzzles resolve-se de forma elegante em Prolog, por instanciação de uma *estrutura de dados desenhada para um problema* (que o descreva).
A resposta (solução) ao puzzle é depois feita extraindo a informação util da estrutura de dados já instanciada.

Exemplo: ***Puzzle dos 3 amigos***

Três amigos ficaram em 1º, 2º e 3º lugar num concurso.
Cada um tem um nome diferente, nacionalidade diferente, e pratica um desporto diferente.

Miguel gosta de basket e foi melhor classificado do que o americano.
Simão, o Israelita, foi melhor do que o jogador de tenis.
O jogador de futebol ficou em 1º lugar.

Quem é o Australiano ?
Qual o desporto favorito do Ricardo ?

Observações:

Cada amigo pode ser representado pelos seus atributos

```
amigo(Nome,Pais,Desporto)
```

Cada amigo tem uma determinada posição (1º, 2º, ou 3º lugar) num concurso.
Isto sugere que a estrutura para modelar o problema seja uma lista de 3 amigos

```
[amigo(N1,P1,D1), amigo(N2,P2,D2), amigo(N3,P3,D3)]
```

As restrições do puzzle podem ser facilmente implementadas pelos seguintes predicados:

```
melhor(A,B,[A,B,C]).
melhor(A,C,[A,B,C]).
melhor(B,C,[A,B,C]).

primeiro([H|_],H).

:- op(100,xfy,on).
                              % on é equivalente a member
X on [X|R].
X on [_|R] :- X on R.

jogaAmigos(NomeAustr,DespRic) :-
     Amigos = [amigo(N1,P1,D1), amigo(N2,P2,D2), amigo(N3,P3,D3)],

     melhor(amigo(miguel,_,basket), amigo(_,americano,_), Amigos),  % info 1
     melhor(amigo(simao,israelita,_), amigo(_,_,tenis),Amigos),     % info 2
     primeiro(Amigos, amigo(_,_,futebol)),                          % info 3

     amigo(NomeAustr,australiano,_) on Amigos,                   % questão 1
     amigo(ricardo,_,DespRic) on Amigos.                         % questão 2
```

```
| ?- jogaAmigos(NomeAustraliano,DesportoRicardo).
DesportoRicardo = tenis,
NomeAustraliano = miguel ?
yes
```


Exercício: Escreva um programa em Prolog para resolver o seguinte puzzle.

Numa rua existem 5 casas, de 5 cores diferentes, habitadas por 5 pessoas diferentes. A nacionalidade de cada habitante, a sua bebida perferida, o seu carro e o seu animal de estimação, são todos diferentes.

1) O ingles mora na casa vermelha
2) O espanhol tem um cão.
3) Bebe-se café na casa verde.
4) O ucraniano bebe chá.
5) A casa verde é à direita da casa beje.
6) O dono do Audi tem um caracol.
7) O Opel está numa casa amarela.
8) Na casa do meio bebe-se leite.
9) O norueguês vive na 1ª casa a contar da esquerda.
10) O dono do Citroën é vizinho do dono da raposa.
11) O Opel está na casa vizinha à casa que tem um cavalo.
12) O dono do Mercedes bebe sumo de laranja.
13) O japonês tem um Renault.
14) O norueguês é vizinho de uma casa azul.

Quem é o dono da zebra ?
Quem bebe água ?