Origem do PROLOG

- A Linguagem PROLOG foi criada nos anos 70 por Alain Colmareur, na Universidade de Marselha
- O nome da linguagem vem de PROgramming in LOGic, ou seja, segue o paradigma da Programação em Lógica
- É conjuntamente com a linguagem LISP, criada nos anos 50, uma das linguagens específicas para o desenvolvimento de Sistemas de Inteligência Artificial
- Enquanto que a linguagem LISP teve impacto nos EUA, o PROLOG alcançou notoriedade na Europa e no Japão
- O principal standard de PROLOG foi proposto em Edimburgo, por Clocksin & Mellish

Conceitos Básicos do PROLOG

Em PROLOG temos 3 conceitos básicos:

- **Factos**

  correspondem a axiomas

  é um tipo das cláusulas do PROLOG

  devem ser colocados no ficheiro da Base de Conhecimento

  p.ex.: rio(douro).
  pai(pedro,ana).

- **Regras**

  correspondem a implicações

  é outro tipo das cláusulas do PROLOG

  devem ser colocadas no ficheiro da Base de Conhecimento

  p. ex.: neto(N,A):-filho(N,P),(descendente(P,A,_);descendente(P,_,A)).

- **Questões**

  permitem interrogar a Base de Conhecimento, a partir da consola, o prompt é ?-

  as respostas correspondem a soluções possíveis

  p.ex.: ?-pai(P,ana).

  ?-neto(rui,A).

Factos

Um facto tem 1 **functor** (nome genérico do facto, começado por uma minúscula), geralmente têm um ou mais **argumentos**, englobados por parêntesis e separados por vírgulas, e termina com um ponto (o finalizador do PROLOG).

Poderão não existir argumentos.

Os argumentos são "**átomos**" (valores constantes, números ou strings começadas por minúsculas ou entre ") ou **variáveis** (começadas por uma maiúscula).

Exemplos de Factos

casados(rui,isabel).

Functor: casados
Argumentos: 2 átomos – rui e isabel
Terminador: .

ligado.

Functor: ligado
Argumentos: nenhum
Terminador: .

potência(X,0,1).

Functor: potência
Argumentos: 3 – uma variável e 2 valores numéricos
Terminador: .

Uma Base de Conhecimento com factos

```
fica(porto,portugal).
fica(lisboa,portugal).
fica(coimbra,portugal).
fica(caminha,portugal).
fica(madrid,espanha).
fica(barcelona,espanha).
fica(zamora,espanha).
fica(orense,espanha).
fica(toledo,espanha).

passa(douro,porto).
passa(douro,zamora).
passa(tejo,lisboa).
passa(tejo,toledo).
passa(minho,caminha).
```

Questões sobre a Base de Conhecimento

```
fica(porto,portugal).        falha
fica(lisboa,portugal).       falha
fica(coimbra,portugal).      falha
fica(caminha,portugal).      falha
fica(madrid,espanha).        sucesso
fica(barcelona,espanha).
fica(zamora,espanha).
fica(orense,espanha).
fica(toledo,espanha).
```

```
?-fica(madrid,espanha).
yes
```

```
passa(douro,porto).          falha
passa(douro,zamora).         falha
passa(tejo,lisboa).          falha
passa(tejo,toledo).          falha
passa(minho,caminha).        falha
```

```
?-passa(mondego,coimbra).
no
```

Usando variáveis nas questões sobre a Base de Conhecimento

```
fica(porto,portugal).        falha
fica(lisboa,portugal).       falha
fica(coimbra,portugal).      falha
fica(caminha,portugal).      falha
fica(madrid,espanha).        sucesso
fica(barcelona,espanha).
fica(zamora,espanha).
fica(orense,espanha).
fica(toledo,espanha).


passa(douro,porto).          falha
passa(douro,zamora).         falha
passa(tejo,lisboa).          sucesso
passa(tejo,toledo).
passa(minho,caminha).
```

```
?-fica(X,espanha).
X=madrid <cr>
yes

?- passa(tejo,C).
C=lisboa  <cr>
yes

?-fica(X,Y).
X=porto Y=portugal <cr>
yes
```

Questões sobre a Base de Conhecimento pedindo alternativas com ;

fica(porto,portugal).	falha
fica(lisboa,portugal).	falha
fica(coimbra,portugal).	falha
fica(caminha,portugal).	falha
fica(madrid,espanha).	sucesso
fica(barcelona,espanha).	sucesso
fica(zamora,espanha).	sucesso
fica(orense,espanha).	sucesso
fica(toledo,espanha).	sucesso

passa(douro,porto).
passa(douro,zamora).
passa(tejo,lisboa).
passa(tejo,toledo).
passa(minho,caminha).

```
?-fica(X,espanha).
X=madrid ;
X=barcelona ;
X=zamora ;
X=orense ;
X=toledo
yes
```

Questões sobre a Base de Conhecimento pondo as alternativas numa lista

fica(porto,portugal).	falha
fica(lisboa,portugal).	falha
fica(coimbra,portugal).	falha
fica(caminha,portugal).	falha
fica(madrid,espanha).	~~sucesso~~
fica(barcelona,espanha).	~~sucesso~~
fica(zamora,espanha).	~~sucesso~~
fica(orense,espanha).	~~sucesso~~
fica(toledo,espanha).	sucesso

?-findall(X,fica(X,espanha),L).

L=[madrid,barcelona,zamora,orense,toledo]

yes

Questões sobre a Base de Conhecimento com conjunção (, na questão)

```
fica(porto,portugal).          sucesso
fica(lisboa,portugal).
fica(coimbra,portugal).
fica(caminha,portugal).
fica(madrid,espanha).
fica(barcelona,espanha).
fica(zamora,espanha).
fica(orense,espanha).
fica(toledo,espanha).

passa(douro,porto).            sucesso
passa(douro,zamora).
passa(tejo,lisboa).
passa(tejo,toledo).
passa(minho,caminha).
```

```
?-fica(X,portugal),passa(R,X).
X=porto R=douro <cr>
yes
```

E se fossem pedidas alternativas com o ; ?

```
?-fica(X,portugal),passa(R,X).
X=porto R=douro ;
X=lisboa R=tejo ;
X=caminha R=minho
```

Questões sobre a Base de Conhecimento com disjunção (; na questão)

```
fica(porto,portugal).
fica(lisboa,portugal).
fica(coimbra,portugal).
fica(caminha,portugal).
fica(madrid,espanha).
fica(barcelona,espanha).
fica(zamora,espanha).
fica(orense,espanha).
fica(toledo,espanha).

passa(douro,porto).
passa(douro,zamora).
passa(tejo,lisboa).
passa(tejo,toledo).
passa(minho,caminha).
passa(minho,orense).
```

sucesso

```
?-fica(X,portugal);fica(X,espanha).
X=porto <cr>
yes
```

E se fossem pedidas alternativas com o ; ?

```
?-fica(X,portugal);fica(X,espanha).
X=porto ;
X=lisboa ;
X=coimbra ;
X=caminha ;
X=madrid ;
X=barcelona ;
X=zamora ;
X=orense ;
X=toledo
yes
```

Questões sobre a Base de Conhecimento com negação (\+)

fica(porto,portugal).	sucesso
fica(lisboa,portugal).	
fica(coimbra,portugal).	
fica(caminha,portugal).	
fica(madrid,espanha).	
fica(barcelona,espanha).	
fica(zamora,espanha).	
fica(orense,espanha).	
fica(toledo,espanha).	
passa(douro,porto).	falha
passa(douro,zamora).	falha
passa(tejo,lisboa).	falha
passa(tejo,toledo).	falha
passa(minho,caminha).	falha
passa(minho,orense).	falha

```
?-\+ fica(porto,portugal).
no

?-\+ passa(mondego,coimbra).
yes
```

Regras

Uma regra tem 1 **termo** (**functor** e **argumentos**) do lado esquerdo que corresponde a aquilo que se pretende provar (conclusão), seguido do "**:-**".

À direita do ":-" podem aparecer outros termos afectados por operadores (, ; not) e por primitivas de controlo. No fundo **o lado direito corresponde às condições da regra**.

Tal como nos factos, podemos ter regras com o mesmo nome e mesmo nº de argumentos, ou até com o mesmo nome e nº de argumentos diferentes.

Exemplos de Regras

filho(X,Y):-homem(X),(descendente(X,Y,_);descendente(X,_,Y)).

Assume-se que descendente(A,B,C) indica que A é filho do pai B e da mãe C

A regra deve ser lida do seguinte modo:

SE X é homem E (X é descendente do seu pai Y OU
X é descendente da sua mãe Y)
ENTÃO X é filho de Y

O _ corresponde a uma variável da qual não necessitamos o valor

potência(_,0,1):-!.

potência(X,N,P):- N1 is N-1,potência(X,N1,P1),P is X*P1.

Esta é uma definição recursiva da potência inteira e não negativa de um nº

A recursividade é muito comum nas regras do PROLOG

O ! (cut) é uma primitiva de controlo que será explicada posteriormente

Acrescentando regras à Base de Conhecimento

```
fica(porto,portugal).
fica(lisboa,portugal).
fica(coimbra,portugal).
fica(caminha,portugal).
fica(madrid,espanha).
fica(barcelona,espanha).
fica(zamora,espanha).
fica(orense,espanha).
fica(toledo,espanha).

passa(douro,porto).
passa(douro,zamora).
passa(tejo,lisboa).
passa(tejo,toledo).
passa(minho,caminha).
passa(minho,orense).

rio_português(R):-passa(R,C),fica(C,portugal).
banhadas_mesmo_rio(C1,C2):-passa(R,C1),passa(R,C2),C1\==C2.
```

Questões sobre a Base de Conhecimento com Regras

```
fica(porto,portugal).
fica(lisboa,portugal).
fica(coimbra,portugal).
fica(caminha,portugal).
fica(madrid,espanha).
fica(barcelona,espanha).
fica(zamora,espanha).
fica(orense,espanha).
fica(toledo,espanha).
```

sucesso (2)

```
passa(douro,porto).
passa(douro,zamora).
passa(tejo,lisboa).
passa(tejo,toledo).
passa(minho,caminha).
passa(minho,orense).
```

sucesso (1)

```
?-rio_português(Rio).
Rio=douro
yes
```

Note-se que:

- na chamada à regra, do lado esquerdo, Rio e R passam a ser a mesma variável;
- passa(R,C) tem sucesso com R=douro e C=porto;
- a chamada seguinte já é feita com C já instanciada com porto, na prática essa chamada é feita como sendo fica(porto,portugal)
- Quando se atinge o ponto a regra tem sucesso

```
rio_português(R):-passa(R,C),fica(C,portugal).
banhadas_mesmo_rio(C1,C2):-passa(R,C1),passa(R,C2),C1\==C2.
```

Questões sobre a Base de Conhecimento com Regras

fica(porto,portugal).	**falha**
fica(lisboa,portugal).	sucesso (2)
fica(coimbra,portugal).	
fica(caminha,portugal).	
fica(madrid,espanha).	
fica(barcelona,espanha).	
fica(zamora,espanha).	
fica(orense,espanha).	
fica(toledo,espanha).	
passa(douro,porto).	**falha**
passa(douro,zamora).	**falha**
passa(tejo,lisboa).	sucesso (1)
passa(tejo,toledo).	
passa(minho,caminha).	
passa(minho,orense).	

?-rio_português(tejo).

yes

Note-se que:

- na chamada à regra, do lado esquerdo, R fica instanciada com o valor tejo;
- dentro da regra (lado direito) a 1ª chamada já é feita como sendo passa(tejo,C) e tem sucesso com R=tejo e C=lisboa;
- a chamada seguinte já é feita com C já instanciada com lisboa, na prática essa chamada é feita como sendo fica(lisboa,portugal)
- Quando se atinge o ponto a regra tem sucesso

rio_português(R):-passa(R,C),fica(C,portugal).
banhadas_mesmo_rio(C1,C2):-passa(R,C1),passa(R,C2),C1\==C2.

Questões sobre a Base de Conhecimento com Regras

fica(porto,portugal).	falha falha
fica(lisboa,portugal).	falha sucesso (3)
fica(coimbra,portugal).	falha
fica(caminha,portugal).	falha
fica(madrid,espanha).	falha
fica(barcelona,espanha).	falha
fica(zamora,espanha).	falha
fica(orense,espanha).	falha
fica(toledo,espanha).	falha
passa(douro,porto).	~~falha~~
passa(douro,zamora).	falha
passa(tejo,toledo).	sucesso (1)
passa(tejo,lisboa).	sucesso (2)
passa(minho,caminha).	
passa(minho,orense).	

E se trocarmos os factos do rio tejo?

?-rio_português(tejo).

yes

- na chamada à regra, do lado esquerdo, R fica instanciada com o valor tejo;
- dentro da regra (lado direito) a 1ª chamada já é feita como sendo passa(tejo,C) e tem sucesso com R=tejo e C=toledo;
- a chamada seguinte já é feita com C já instanciada com toledo, na prática essa chamada é feita como sendo fica(toledo,portugal) e **falha**
- Volta-se atrás (**backtracking**) e é tentada uma nova solução para passa(tejo,C), ficando C=lisboa
- a chamada seguinte já é feita com C já instanciada com lisboa, na prática essa chamada é feita como sendo fica(lisboa,portugal)
- Quando se atinge o ponto a regra tem sucesso

rio_português(R):-passa(R,C),fica(C,portugal).
banhadas_mesmo_rio(C1,C2):-passa(R,C1),passa(R,C2),C1\==C2.

Questões sobre a Base de Conhecimento com Regras

```
fica(porto,portugal).
fica(lisboa,portugal).
fica(coimbra,portugal).
fica(caminha,portugal).
fica(madrid,espanha).
fica(barcelona,espanha).
fica(zamora,espanha).
fica(orense,espanha).
fica(toledo,espanha).

passa(douro,porto).
passa(douro,zamora).
passa(tejo,lisboa).
passa(tejo,toledo).
passa(minho,caminha).
passa(minho,orense).
```

suc.(1) ~~suc.(2)~~ suc.(3)

?-banhadas_mesmo_rio(C1,C2).

C1=porto C2=zamora

yes

Note-se que:

- na chamada à regra, do lado esquerdo, C1 e C2 continuam como sendo não instanciadas;
- dentro da regra (lado direito) a 1ª chamada já é feita como sendo passa(R,C1) e tem sucesso com R=douro e C1=porto;
- a chamada seguinte já é feita com R já instanciada com douro, na prática essa chamada é feita como sendo passa(douro,C2) e tem sucesso com C2=porto
- O teste seguinte **falha** (porto não é diferente de porto) e faz-se o **backtracking**
- Agora passa(douro,C2) tem sucesso com C2=zamora
- O teste seguinte tem sucesso (porto é diferente de zamora)
- Quando se atinge o ponto a regra tem sucesso

```
rio_português(R) :- passa(R,C),fica(C,portugal).
banhadas_mesmo_rio(C1,C2) :- passa(R,C1),passa(R,C2),C1\==C2.
```

Questões sobre a Base de Conhecimento com Regras

```
fica(porto,portugal).
fica(lisboa,portugal).
fica(coimbra,portugal).
fica(caminha,portugal).
fica(madrid,espanha).
fica(barcelona,espanha).
fica(zamora,espanha).
fica(orense,espanha).
fica(toledo,espanha).

passa(douro,porto).
passa(douro,zamora).
passa(tejo,lisboa).
passa(tejo,toledo).
passa(minho,caminha).
passa(minho,orense).

rio_português(R):-passa(R,C),fica(C,portugal).

banhadas_mesmo_rio(C1,C2):-passa(R,C1),passa(R,C2),C1\==C2.
```

Experimente fazer as seguintes questões e efectue as "traçagens"

?-banhadas_mesmo_rio(orense,C).

?-banhadas_mesmo_rio(C,lisboa).

?-banhadas_mesmo_rio(zamora,porto).

?-banhadas_mesmo_rio(lisboa,porto).

?-banhadas_mesmo_rio(coimbra,C).

Variáveis em PROLOG

- As variáveis em PROLOG têm um desempenho diferente das variáveis de outras linguagens

- Em PROLOG uma variável pode estar apenas em dois estados: não instanciada ou instanciada

- Uma vez instanciada uma variável só pode mudar de valor pelo processo de backtracking, ou seja, voltando a ficar como não instanciada para tomar outro valor

  Exemplos:

  - No acetato 19, C tomou o valor toledo e depois por backtracking passou a tomar o valor lisboa
  - no acetato 20, C2 tomou o valor porto e depois por backtracking passou a tomar o valor zamora

Variáveis em PROLOG

- Em PROLOG o incremento duma variável nunca pode ser feito como N is N+1 (**is** é a atribuição numérica)
  - Se N não estiver instanciado ocorre uma falha ao tentar avaliar N+1
  - Se N estiver instanciado não poderemos obrigar a mudar o seu valor
  - Deve ser usado N1 is N+1

- Se for pedida a impressão de uma variável não instanciada aparecerá um nº antecedido de _ (por exemplo _22456) que tem a ver com a referência da variável e não com o seu valor

- Quando num facto ou regra não interesse o valor de uma variável, esta pode ser substituída por _

- Para saber se uma variável está ou não instanciada devemos usar:
  - var(X) – tem sucesso se X não estiver instanciada;
  - nonvar(X) – tem sucesso se X estiver instanciada

Variáveis em PROLOG

- Vejamos através da interacção com a consola o modo de funcionamento de uma variável

```
?- write('X='),write(X),nl,X=a,write('X='),write(X),nl.
X=_22650
X=a
X = a

?- write('X='),write(X),nl,X=a,write('X='),write(X),nl,X=b.
X=_39436
X=a
no
```

Operadores Lógicos em PROLOG

- Já foi visto que os operadores lógicos em PROLOG eram:
  - , para a conjunção
  - ; para a disjunção
  - \+ para a negação
- Podemos usar os () para tirar ambiguidades ou forçar as expressões pretendidas

Vamos considerar a seguinte base de factos (com factos sem argumentos):

```
a.
b.
c:-fail. /*o fail origina uma falha*/
d.
```

Operadores Lógicos em PROLOG

Base de Factos

```
a.
b.
c:-fail.
d.
```

Questões:

```
?- a.
yes
?- c.
no
?- \+ a.
no
?- \+ c.
yes
?- a,b.
yes
?- a,c.
no
?- a;c.
yes
?- (a,c);(\+ a;b).
yes
```

Operadores Aritméticos em PROLOG

- Os operadores aritméticos do PROLOG são:

+ adição X+Y
- Subtração X-Y
* Multiplicação X*Y
/ divisão X/Y
// divisão inteira X//Y
mod resto da divisão inteira X mod Y
^ Potência X^Y
- Simétrico -X

Funções Aritméticas em PROLOG

- Embora a linguagem PROLOG não seja a mais adequada para cálculo numérico, como em qualquer outra linguagem temos funções aritméticas, alguns exemplos do SWI-PROLOG:

abs(X)	valor absoluto de X
acos(X)	arco-cosseno de X (graus)
exp(X)	e^X
alog(X)	10^X
asin(X)	arco-seno de X (graus)
atan(X)	arco-tangente de X (graus)
cos(X)	cosseno de X (graus)
float_fractional_part(X)	parte não inteira de X (mesmo sinal que X)
floor(X)	inteiro igual ou imediatamente anterior a X

Funções Aritméticas em PROLOG

float_integer_part(X)	parte inteira de X
log(X)	logaritmo natural de X
log10(X)	logaritmo decimal de X
max(X,Y)	máximo entre X e Y
min(X,Y)	mínimo entre X e Y
random(X)	gera um número aleatório entre 0 e X (vírgula flutuante)
sign(X)	sinal de X (-1 se negativo, 0 se zero ou 1 se positivo)
sin(X)	seno de X (graus)
sqrt(X)	raiz quadrada de X
tan(X)	tangente de X (graus)

Operadores Relacionais em Prolog

- Os operadores relacionais do PROLOG são:

  == igualdade X==Y

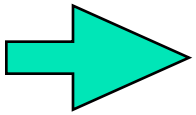
  \== diferença X\==Y

  > maior X>Y

  < menor X<Y

  =< menor ou igual X=<Y

  >= maior ou igual X >= Y

- Convém atender ao facto das variáveis poderem estar instanciadas ou não 

Operadores Relacionais em PROLOG

```
?- X=a,Y=a,X==Y.
X = Y = a

?- X=a,Y=b,X==Y.
no

?- X=a,X==Y.
no

?- X=a,Y=b,X\==Y.
X = a ,
Y = b

?- X=a,X\==Y.
X = a ,
Y = _

?- X==Y.
no

?- X\==Y.
X = _ ,
Y = _

?- X=Y,X==Y.
X = Y = _

?- X=Y,X\==Y.
no
```

Atribuição em PROLOG

- Em PROLOG temos 2 operadores de **atribuição**
  - = para a atribuição **simbólica** X=a
  - is para a atribuição **numérica** X is 5
- A atribuição simbólica é **bidireccional**, para X=Y temos:
  - Se X não está instanciado e Y está então temos X←Y
  - Se X está instanciado e Y não está então temos X→Y
  - Se nenhum está instanciado então passam a ser a mesma variável
  - Se ambos estão instanciados com o mesmo valor então há sucesso
  - Se ambos estão instanciados com valores diferentes então ocorre uma falha

Atribuição em PROLOG

```
?- X=Y,X=a.
X = Y = a

?- Y=a,X=Y.
Y = X = a

?- X=a,X=Y.
X = Y = a

?- X=Y.
X = Y = _

?- X=a,Y=a,X=Y.
X = Y = a

?- X=a,Y=b,X=Y.
no
```

Atribuição em PROLOG

- A atribuição numérica é **unidireccional**
- Do lado direito do is, se estiverem envolvidas variáveis, elas devem estar instanciadas
- Do lado esquerdo a variável não deve estar instanciada, senão ocorre uma falha
- Do lado direito as variável que apareçam devem estar instanciadas

Em PROLOG N is N+1 nunca tem sucesso

Escrita em PROLOG

- A escrita no "output stream" (normalmente o monitor) é feita com o write
  - write(hello) – escreve hello
  - write(´Hello World´) – escreve Hello World
  - write(Hello) – escreve conteúdo da variável Hello

- Outra possibilidade é escrever usando o put
  - put(65) – escreve o caracter A (código ASCII 65)

- A mudança de linha é feita com o nl

Leitura em PROLOG

- A leitura do "input stream" (normalmente o teclado) é feita com o read
  - read(X) – lê o valor de X
  - Deve-se terminar com o . seguido de RETURN

- Outra possibilidade é ler usando o get ou o get0
  - get(A) – lê o próximo carácter (não branco, ou seja, ignora CR, TAB, espaço)
  - get0(A) - lê o próximo carácter

```
?- get(X),get(Y),get(Z).
ab  c
X = 97 ,
Y = 98 ,
Z = 99
```

```
?- get0(X),get0(Y),get0(Z).
ab
X = 97 ,
Y = 98 ,
Z = 13
```

Recursividade

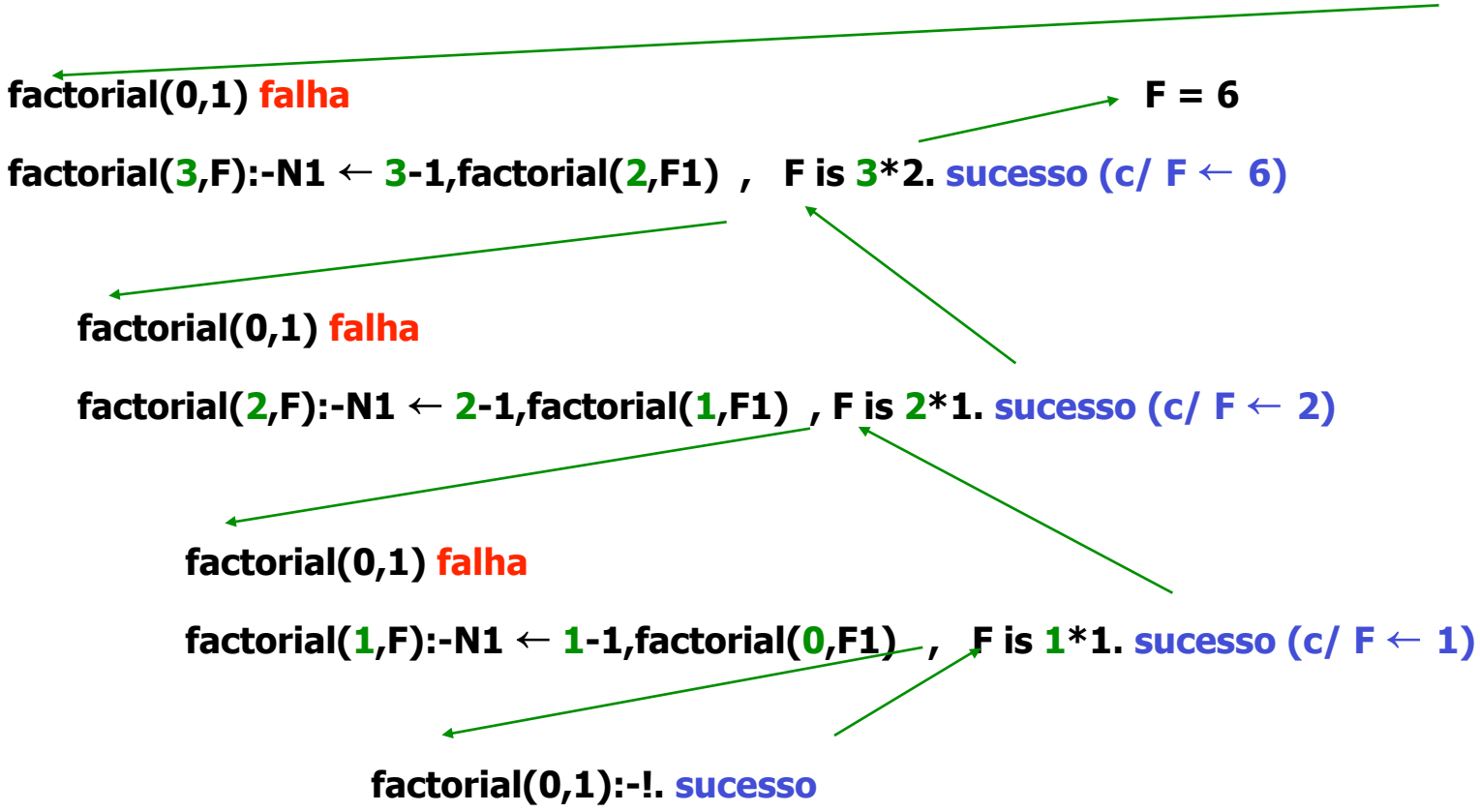
- O uso da recursividade é muito comum na linguagem PROLOG

- Na implementação de um predicado recursivo devemos ter em atenção que deverá haver sempre uma alternativa para finalizar as chamadas recursivas
  - Por uma regra, ou facto, que não efectua essa chamada
  - Por uma das alternativas de uma disjunção

Recursividade

```
factorial(0,1):-!.        /* a função do ! será explicada posteriormente */
factorial(N,F):-N1 is N-1,factorial(N1,F1),F is N*F1.
```

Vejamos o que acontece quando se efectua a chamada ?-factorial(3,F).



Considerando o seguinte programa que resolve o problema das Torres de Hanói:

```
hanoi(N) :- move(N,e,c,d).
move(0,_,_,_) :- !.
move(N,A,B,C) :- M is N-1,
                 move(M,A,C,B),
                 informa(A,B),
                 move(M,C,B,A).
informa(A,B) :- write('MOVER DISCO DE '),
                write(A),write(' PARA '),write(B),nl.
```

Efectue a "traçagem" do que acontece quando se faz a chamada

?-hanoi(3).

Estruturas de Controlo do PROLOG

- Em PROLOG temos 4 estruturas de controlo principais:
  - true – tem sempre sucesso
  - fail – Falha sempre
  - repeat – tem sempre sucesso, quando se volta para trás por backtracking e se passa pelo repeat, este tem sempre sucesso e obriga a ir para a frente
  - ! – lê-se "cut", uma vez atingida uma solução para o que está antes do ! Já não será possível voltar para trás (antes do cut) pelo processo de backtracking

O true

- Se não existisse em PROLOG o true poderia ser implementado pelo seguinte facto:

  true.

- Vejamos um exemplo onde o true faz sentido

```
cidade1(C):-    fica(C,P),
                ((P==portugal,write(C),write(´ é portuguesa'),
                 nl);true).
```

  - Se não tivéssemos o true ocorreria uma falha se C não fosse uma cidade portuguesa, mas o que queremos é que tenha sucesso se C for uma cidade e para o caso particular de ser portuguesa deve aparecer uma mensagem escrita indicando isso

O fail

- O fail obriga à ocorrência de uma falha, é útil no raciocínio pela negativa
- Vejamos um exemplo usando o fail

```
sem_precedentes(X) :- precede(_,X),!,fail.
sem_precedentes(_).

precede(a,b).
precede(a,c).
precede(c,d).
precede(b,e).
precede(f,h).
precede(d,h).
precede(g,i).
precede(e,i).
```

O repeat

- Se não existisse em PROLOG, o repeat poderia ser implementado do seguinte modo:

```
repeat.
repeat:-repeat.
```

- Vejamos um exemplo usando o repeat

```
read_command(C):-menu,
                 repeat,
                 write('Comando --> '),
                 read(C),
          (C==continue;C==abort;C==help;C==load;C==save).

menu:- write('Opções:'),nl,nl,
       write(continue),nl,
       write(abort),nl,
       write(help),nl,
       write(load),nl,
       write(save),nl,nl.
```

O ! (cut)

- Serve para limitar o retrocesso, uma vez passado o ! Não se poderá voltar para trás dele pelo processo de backtracking
- Serve ainda para delimitar a entrada em regras alternativas
- Permite optimizar os programas evitando que se perca tempo com análises não necessárias
- Por vezes é muito redutor, levando a que só seja permitida uma única solução, quando podem existir várias
- O uso adequado do ! É uma das maiores dificuldades dos iniciados no PROLOG

Uma Base de Conhecimento e uma regra com um !

```
a(1).
a(2).
a(3).

b(1,4).
b(3,5).

c(3,6).
c(5,7).
c(5,8).

d(7,9).
d(7,10).
d(8,11).

e(9,12).
e(9,13).
e(10,14).
e(11,15).
e(11,16).

r(X,Y,Z,U,V):-a(X),b(X,Y),c(Y,Z),!,d(Z,U),e(U,V).
```

O que acontecerá se colocarmos a questão

?-r(X,Y,Z,U,V).

e pedirmos várias soluções?

Uma Base de Conhecimento com uma regra com um !

a(1).	~~sucesso~~		
a(2).		~~sucesso~~	
a(3).			sucesso
b(1,4).	~~sucesso~~	falha	falha
b(3,5).	falha	falha	sucesso
c(3,6).	falha		falha
c(5,7).	falha		sucesso
c(5,8).	falha		
d(7,9).			
d(7,10).			
d(8,11).			
e(9,12).			
e(9,13).			
e(10,14).			
e(11,15).			
e(11,16).			

?-r(X,Y,Z,U,V).

- Ao entrar na regra, chama-se a(X) e X toma o valor 1;
- É feita a chamada b(1,Y) e Y toma o valor 4;
- É feita a chamada c(4,Z) e **falha**
- Volta-se atrás por **backtracking**, mas não há mais soluções para b(1,Y) e **falha**
- Volta-se atrás por **backtracking** e a(X) permite que X tome o valor 2
- É feita a chamada de b(2,Y) e **falha**
- Volta-se atrás por **backtracking** e a(X) permite que X tome o valor 3
- É feita a chamada b(3,Y) e Y toma o valor 5;
- É feita a chamada c(5,Z) e Z toma o valor 7

r(X,Y,Z,U,V):-a(X),b(X,Y),c(Y,Z),!,d(Z,U),e(U,V).

Uma Base de Conhecimento com uma regra com um !

```
a(1).
a(2).
a(3).

b(1,4).
b(3,5).

c(3,6).
c(5,7).
c(5,8).

d(7,9).
d(7,10).
d(8,11).

e(9,12).
e(9,13).
e(10,14).
e(11,15).
e(11,16).

r(X,Y,Z,U,V):-a(X),b(X,Y),c(Y,Z),!,d(Z,U),e(U,V).
```

sucesso ~~sucesso~~

sucesso(1ªsol.) falha

falha

sucesso(2ªsol.)

- Neste momento, X=3,Y=5 e Z=7, e passamos pelo !, logo não será possível encontrar nenhuma solução com outros valores de X,Y e Z, visto que foram instanciados antes do !;
- É feita a chamada d(7,U) e U toma o valor 9;
- É feita a chamada e(9,V) e V toma o valor 12;
- Chegamos ao . e encontramos a 1ª solução:
  - X=3, Y=5, Z=7, U=9, V=12
- E se pedirmos mais uma solução com o ";" ;
- É tentada uma nova solução para e(9,V) e V toma o valor 13
- Chegamos ao . e encontramos a 2ª solução:
  - X=3, Y=5, Z=7, U=9, V=13
- E se pedirmos mais uma solução com o ";" ;

Uma Base de Conhecimento com uma regra com um !

```
a(1).
a(2).
a(3).

b(1,4).
b(3,5).

c(3,6).
c(5,7).
c(5,8).

d(7,9).
d(7,10).
d(8,11).

e(9,12).
e(9,13).
e(10,14).
e(11,15).
e(11,16).
```

sucesso
sucesso
falha

sucesso(1ªsol.) falha
sucesso(2ªsol.) falha
falha sucesso(3ªsol.)
falha falha
falha falha

```
r(X,Y,Z,U,V):-a(X),b(X,Y),c(Y,Z),!,d(Z,U),e(U,V).
```

- É tentada uma nova solução para e(9,V) e **falha**
- Volta-se atrás por **backtracking**, e d(7,V) origina uma nova solução com V=10;
- É feita a chamada e(10,V) e V toma o valor 14;
- Chegamos ao . e encontramos a 3ª solução:
  - X=3, Y=5, Z=7, U=10, V=14
- E se pedirmos mais uma solução com o ";" ;
- É tentada uma nova solução para e(10,V) e **falha**
- Volta-se atrás por **backtracking**, e d(7,V) **falha** ;
- Ao tentar voltar para trás por backtracking, encontra-se o ! e portanto não há mais soluções e **falha**

Uma Base de Conhecimento com uma regra com um !

```
a(1).
a(2).
a(3).

b(1,4).
b(3,5).

c(3,6).
c(5,7).
c(5,8).

d(7,9).
d(7,10).
d(8,11).

e(9,12).
e(9,13).
e(10,14).
e(11,15).
e(11,16).

r(X,Y,Z,U,V):-a(X),b(X,Y),c(Y,Z),!,d(Z,U),e(U,V).
```

Concluindo:

- O ! não impediu o backtracking antes dele
- O ! não impediu o backtracking depois dele
- Aquilo que o ! impede é que o processo de backtracking se extenda de depois do ! para antes do !

- Experimente retirar o ! e pedir todas as soluções

?-r(X,Y,Z,U,V).

X=3, Y=5, Z=7, U=9, V=12 ;
X=3, Y=5, Z=7, U=9, V=13 ;
X=3, Y=5, Z=7, U=10, V=14 ;
X=3, Y=5, Z=8, U=11, V=15 ;
X=3, Y=5, Z=8, U=11, V=16

O ! para evitar a entrada em regras alternativas

- O ! é frequentemente usado para que depois de se atingir o sucesso por uma das alternativas das regras se impeça que por backtracking se atinja o sucesso por outra alternativa

  Um exemplo:

  ```
  potência(_,0,1).
  potência(X,N,P):-N1 is N-1,potência(X,N1,P1),P is X*P1.
  ```

- Vejamos o que acontece se forem pedidas várias soluções quando é posta a seguinte questão:

  ```
  ?- potência(5,3,P).
  P = 125 ;
  Error 2, Local Stack Full, Trying potência/3
  Aborted
  ```

O ! para evitar a entrada em regras alternativas

- No exemplo, a falha ocorreu porque depois de se atingir o sucesso pela 1ª alternativa (quando o 2º argumento tomou o valor 0), se continuar a tentar uma nova solução pela segunda e o segundo argumento passa a tomar valores negativos (-1,-2,-3,…) nos níveis de recursividade que se seguem, até que a Stack fica cheia
- Não adianta aumentar a dimensão da Stack
- A solução consiste em por um !:

```
potência(_,0,1):-!.
potência(X,N,P):-N1 is N-1,potência(X,N1,P1),P is X*P1.
```

- E agora passa a haver apenas uma solução:

```
?- potência(5,3,P).
P = 125
```

Listas

- Em PROLOG as listas podem ser:
  - Não vazias, tendo
    - uma cabeça (1º elemento da lista)
    - e uma cauda (lista com os restantes elementos)
  - Vazias, quando não têm nenhum elemento (equivalente ao NULL ou NIL de outras linguagens), uma lista vazia não tem cabeça nem cauda
- As listas podem ser representadas
  - pela enumeração dos seus elementos separados por vírgulas e envolvidos por [e]
    - por exemplo [a,b,c,d]
  - pela notação cabeça-cauda separadas pelo | e envolvidas por [e]
    - por exemplo [H|T]
- Em PROLOG os elementos das listas não têm de ser do mesmo tipo (por exemplo, [a,2,abc,[x,1,zzz]])

Listas

- [] é a lista vazia
- [a] é uma lista com 1 elemento (a)
- [X] é uma lista com 1 elemento (a variável X)
- [b,Y] é uma lista com 2 elementos (b e a variável Y)
- [X,Y,Z] é uma lista com 3 elementos (as variáveis X,Y e Z)
- [H|T] é uma lista com cabeça H e cauda T

Vejamos algumas questões PROLOG:

```
?- [H|T]=[a,b,c,d].
H = a ,
T = [b,c,d]

?- [H|T]=[a,b,X].
H = a ,
T = [b,X] ,
X = _

?- [H|T]=[a].
H = a ,
T = []

?- [H|T]=[[a,b],3,[d,e]].
H = [a,b] ,
T = [3,[d,e]]

?- [H|T]=[].
no
```

O Predicado membro

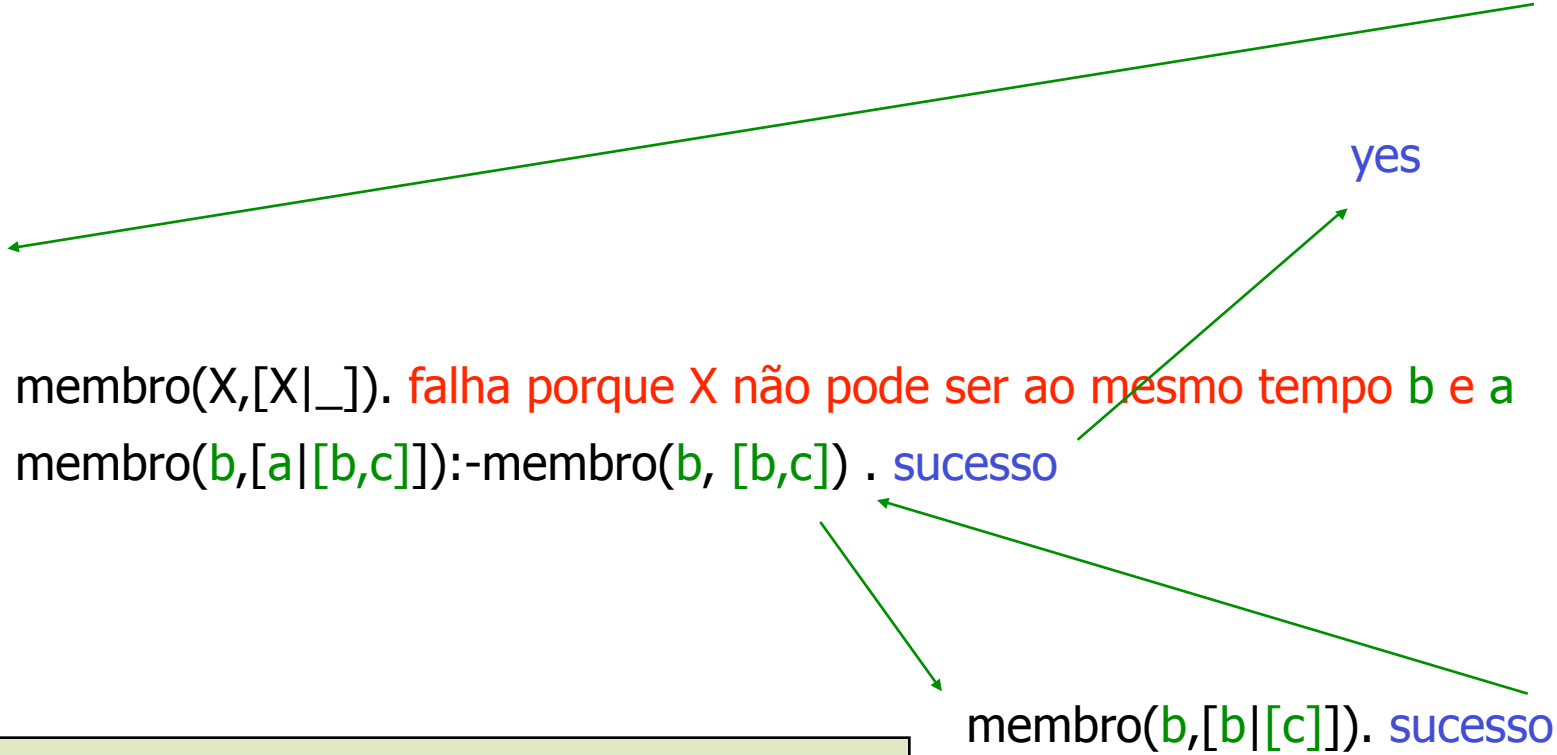
- Hoje em dia as implementações de PROLOG já trazem o predicado member/2 (dois argumentos, aridade 2) que verifica se o primeiro argumento é membro da lista do segundo argumento.

- Mas se esse predicado não existisse poderia ser implementado do seguinte modo:

```
membro(X,[X|_]).
membro(X,[_|L]) :- membro(X,L).
```

O Predicado membro

Vejamos o que acontece quando se põe a questão ?-membro(b,[a,b,c]).




```
membro(X,[X|_]).
membro(X,[_|L]) :- membro(X,L).
```

O Predicado membro

Vejamos o que acontece quando se põe a questão ?-membro(c,[a,b]).

membro(X,[X|_]). falha porque X não pode ser ao mesmo tempo c e a

membro(c,[a|[b]]):-membro(c, [b])

membro(X,[X|_]). falha porque X não pode ser c e b

membro(c,[b|[ ]]):-membro(c, [ ])

membro(X,[X|_]). falha, [X|_] não é instanciável com [ ]

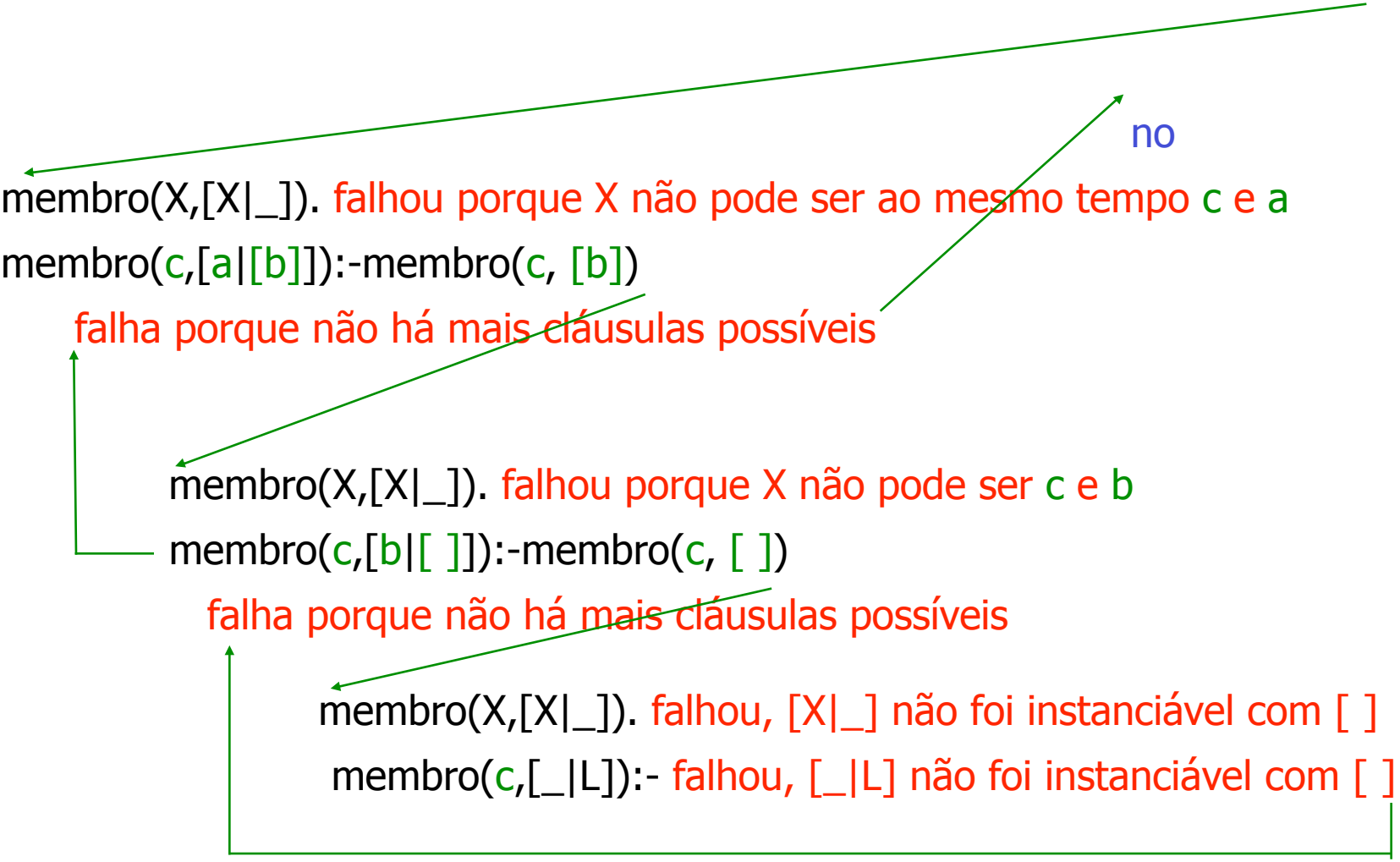
membro(c,[_|L]):- falha, [_|L] não é instanciável com [ ]

```
membro(X,[X|_]).
membro(X,[_|L]) :- membro(X,L).
```

O Predicado membro

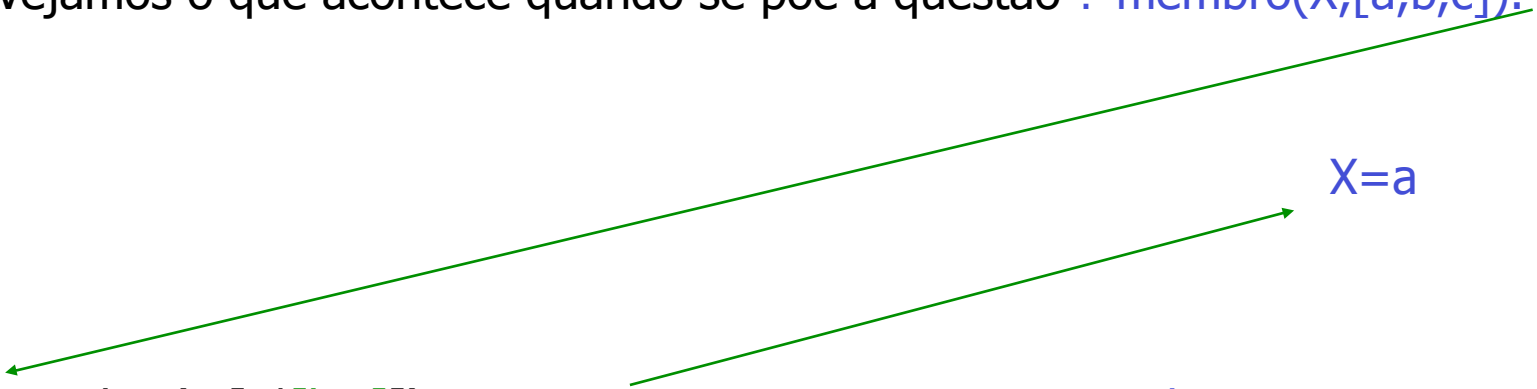
Continuando a ver o que acontece

?-membro(c,[a,b]).



O Predicado membro

Vejamos o que acontece quando se põe a questão ?-membro(X,[a,b,c]).



membro(a,[a|[b,c]]).

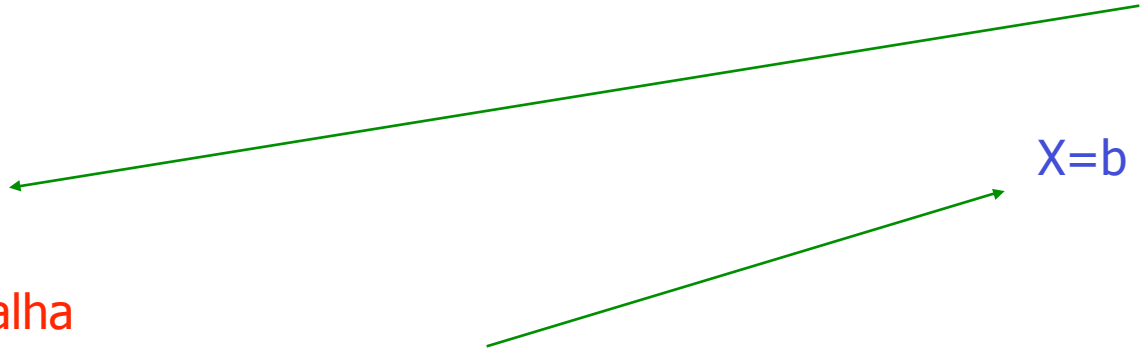
sucesso, como X tomou o valor a no 2º argumento, o primeiro argumento também fica com o valor a

E se carregarmos em ; tudo se passará como se tivesse ocorrido uma falha e o PROLOG irá tentar a segunda alternativa da cláusula

```
membro(X,[X|_]).
membro(X,[_|L]) :- membro(X,L).
```

O Predicado membro

Continuação após pedir nova solução após ter dado a primeira solução X=a ;



X=b

membro(a,[a|[b,c]]). falha

membro(X,[a|[b,c]]):-membro(X, [b,c]) . sucesso, X tomou o valor b no retorno

membro(b,[b|[c]]). sucesso, como X tomou o valor b no 2º argumento, o primeiro argumento também fica com o valor b

```
membro(X,[X|_]).
membro(X,[_|L]) :- membro(X,L).
```

O Predicado membro

Vejamos então os possíveis resultados das questões:

```
?- membro(b,[a,b,c]).
yes

?- membro(c,[a,b]).
no

?- membro(X,[a,b,c]).
X = a ;
X = b ;
X = c ;
no

?- membro(a,L).
L = [a|_899] ;
L = [_49162,a|_49171] ;
L = [_49162,_45220,a|_45229]
```

O Predicado membro

A análise do predicado **membro** permitiu:

- Ver se um elemento conhecido pertencia ou não a uma lista de elementos conhecidos (relação de pertença)
- Seleccionar um elemento de uma lista
- Gerar uma potencial lista que contenha um elemento

E se trocássemos a ordem das cláusulas?

```
membro1(X,[_|L]):-membro1(X,L).
membro1(X,[X|_]).
```

Observe-se que agora dá-se prioridade à visita da cauda da lista, antes de ver qual é o elemento que está à cabeça

O Predicado membro

A interacção agora seria a seguinte:

?- membro1(b,[a,b,c]).
yes

?- membro1(c,[a,b]).
no

?- membro1(X,[a,b,c]).
X = c ;
X = b ;
X = a

?- membro1(a,L).
Error 1, Backtrack Stack Full, Trying membro1/2
Aborted

O Predicado membro

- A inversão nas cláusulas não afectou os dois primeiros exemplos (membro1(b,[a,b,c]) deu yes e membro1(c,[a,b]) deu no).
- A formulação de membro1 é menos eficiente, sobretudo se a lista for longa. Exemplo:
  1. **membro1(a,[a,b,c,d,e])** terá sucesso, mas deverá 1º visitar todos os elementos da lista até ocorrer uma falha, voltando para trás por b/t.
  2. nessa processo, verificará quais os elementos que estão à cabeça, falhando sempre até chegar ao 1º nível, quando ocorre o sucesso

O Predicado membro

- No terceiro exemplo (membro1(X,[a,b,c])) é visível que a prioridade é dada à visita da cauda da lista, as soluções são obtidas do final para o princípio da lista
- O último funcionamento conduz, na prática, a chamadas recursivas infinitas (só não o são porque a stack "estoura")
- Concluindo
  - apesar de membro1 resolver os casos mais normais, o seu funcionamento não é nada eficiente
  - Para o último caso testado, cujo uso não é vulgar, o membro1 está mesmo mal concebido

O Predicado membro

Vamos agora considerar o seguinte programa:

```
teste:-  write('A -> '),read(A),
         write('lista L -> '),read(L),
         membro2(A,L),A<5.

membro2(X,[X|_]):-write('passando por X= '),write(X),nl.
membro2(X,[Y|L]):-write('passando por Y= '),
                  write(Y),nl,membro2(X,L).
```

É óbvio que o teste A<5 deveria estar após a leitura de A, mas este é um exemplo académico e vamos ver o que acontece

O Predicado membro

```
?- teste.
A -> |: 3.
lista L -> |: [1,2,3,6,3,6,7].
passando por Y= 1
passando por Y= 2
passando por X= 3
yes
?- teste.
A -> |: 6.
lista L -> |: [1,2,3,6,3,6,7].
passando por Y= 1
passando por Y= 2
passando por Y= 3
passando por X= 6   sucesso pela 1ª cláusula de membro2, mas falha A<5
passando por Y= 6   entra pela 2ª cláusula de membro2
passando por Y= 3
passando por X= 6   sucesso pela 1ª cláusula de membro2, mas falha A<5
passando por Y= 6   entra pela 2ª cláusula de membro2
passando por Y= 7
no                  falha pois a lista acaba
```

```
teste:-     write('A -> '),read(A),
            write('lista L -> '),read(L),
            membro2(A,L),A<5.

membro2(X,[X|_]):-write('passando por X= '),write(X),nl.
membro2(X,[Y|L]):-write('passando por Y= '),
                  write(Y),nl,membro2(X,L).
```

O Predicado membro

O problema poderia ser resolvido colocando um ! na primeira alternativa de membro 2:

```
teste:-  write('A -> '),read(A),
         write('lista L -> '),read(L),
         membro2(A,L),A<5.


membro2(X,[X|_]):-!,write('passando por X= '),write(X),nl.
membro2(X,[Y|L]):-write('passando por Y= '),
                  write(Y),nl,membro2(X,L).
```

Isso impede que se houver sucesso pela primeira cláusula de membro2 e se depois noutro predicado ocorrer uma falha (neste caso A<5) ao voltarmos para trás não iremos tentar a outra cláusula de membro2

O Predicado membro

Agora temos:

```
?- teste.
A -> |: 3.
lista L -> |: [1,2,3,6,3,6,7].
passando por Y= 1
passando por Y= 2
passando por X= 3
yes

?- teste.
A -> |: 6.
lista L -> |: [1,2,3,6,3,6,7].
passando por Y= 1
passando por Y= 2
passando por Y= 3
passando por X= 6
no
```

O Predicado membro

Mas vejamos o funcionamento de membro2 como selector de elementos de uma lista:

?- membro2(X,[a,b,c,d]).

passando por X= a

X = a

Só obtivemos uma solução, compare-se com o que se passa com membro:

?- membro(X,[a,b,c,d]).

X = a ;

X = b ;

X = c ;

X = d ;

no

O Predicado membro

Concluímos que:

- O uso do ! em membro2 permitiu reduzir a ineficiência, pois impediu que tentássemos de novo encontrar o 6 na lista, quando isso não iria alterar nada (continua a não ser menor que 5)

- Mas o mesmo ! limitou em demasia o processo de retrocesso (como já vimos anteriormente) e levou a que membro2 não funcionasse bem como selector de elementos de uma lista

O Predicado concatena

- Hoje em dia as implementações de PROLOG já trazem o predicado append/3 que junta a lista do 1º argumento com a lista do 2º, gerando a lista do 3º argumento.
- Mas se esse predicado não existisse poderia ser implementado do seguinte modo:

```
concatena([ ],L,L).
concatena([A|B],C,[A|D]):-concatena(B,C,D).
```

```
conc( [ ], L, L ).
conc( [X|L1], L2, [X|L3] :- conc( L1, L2, L3 ).
```

O predicado concatena

Vamos ver como seria a resposta a questões sobre o **concatena**

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

?- concatena(L1,L2,[a,b,c]).
L1 = [] ,
L2 = [a,b,c] ;

L1 = [a] ,
L2 = [b,c] ;

L1 = [a,b] ,
L2 = [c] ;

L1 = [a,b,c] ,
L2 = [] ;

no
```

```
concatena([ ],L,L).
concatena([A|B],C,[A|D]):-concatena(B,C,D).
```

O Predicado concatena

Vamos ver o que acontece quando se põe a questão:

?-concatena([a,b],[c,d,e],L).

concatena([],L,L). falha

concatena([a|[b]],[c,d,e],[a|D]):- concatena([b],[c,d,e],D)

concatena([],L,L). falha

concatena([b|[ ]],[c,d,e],[b|D]):- concatena([ ],[c,d,e],D)

concatena([ ], [c,d,e], [c,d,e]). sucesso

```
concatena([ ],L,L).
concatena([A|B],C,[A|D]):-concatena(B,C,D).
```

O Predicado concatena

Continuando…:

?-concatena([a,b],[c,d,e], [a,b,c,d,e]). sucesso

L=[a,b,c,d,e]

concatena([],L,L). falha

concatena([a|[b]],[c,d,e],[a|[b,c,d,e]]):-concatena([b],[c,d,e],[b,c,d,e]).

sucesso

concatena([],L,L). falha

concatena([b|[ ]],[c,d,e],[b|[c,d,e]]):- concatena([ ],[c,d,e], [c,d,e]).

sucesso

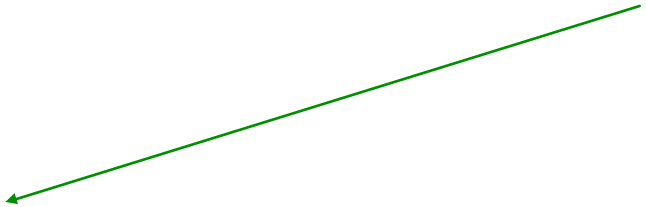
concatena([ ], [c,d,e], [c,d,e]). sucesso

```
concatena([ ],L,L).
concatena([A|B],C,[A|D]):-concatena(B,C,D).
```

O Predicado concatena

Vejamos agora o que se passa com as duas primeiras listas não instanciadas e a 3ª instanciada

?-concatena(L1,L2, [a,b,c]).



concatena([], [a,b,c], [a,b,c]). sucesso

O Predicado concatena

Continuando...vendo a 1º solução...:

?-concatena([ ], [a,b,c], [a,b,c]). sucesso

L1=[ ] L2=[a,b,c];

concatena([], [a,b,c], [a,b,c]). sucesso

... e se carregarmos em ;

O Predicado concatena

... vai ser gerada uma segunda solução

?-concatena(L1,L2,[a,b,c]).

L1=[] L2=[a,b,c] ; sucesso

L1=[a] L2=[b,c];

concatena([], L, L). sucesso

concatena([a|B], C,[a|[b,c]]):-concatena(B, C, [b,c]).

concatena([], [b,c], [b,c]). sucesso

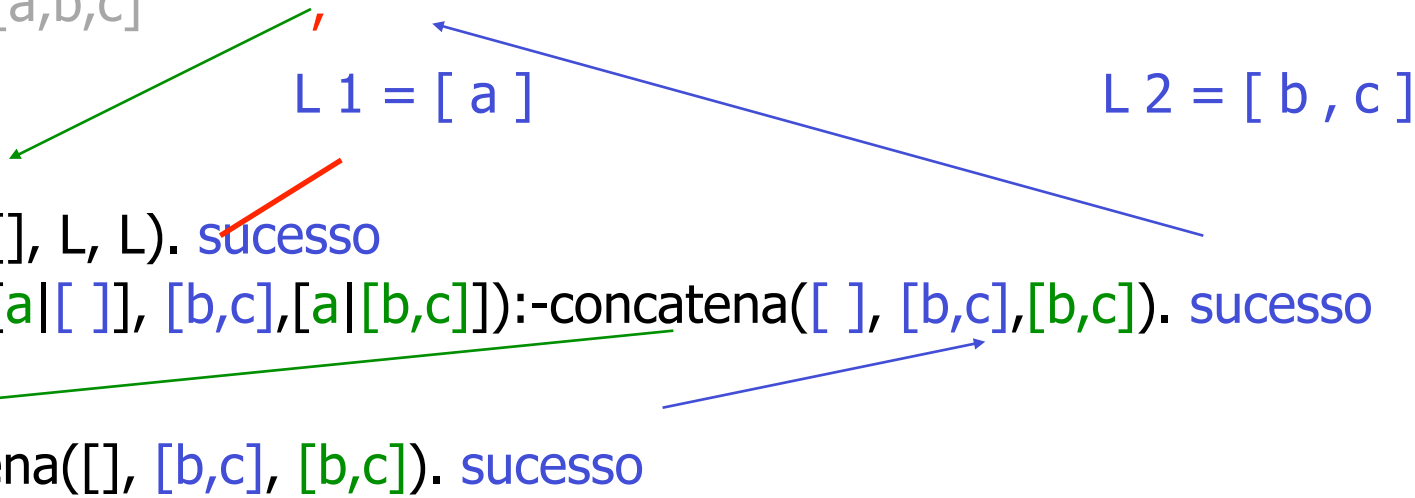
```
concatena([ ],L,L).
concatena([A|B],C,[A|D]):-concatena(B,C,D).
```

O Predicado concatena

... continuando...

?-concatena([a],[b,c],[a,b,c]).

L1=[] L2=[a,b,c] ;



E se forem pedidas novas soluções com o ; apareceriam as seguintes soluções

L1=[a,b] L2=[c] ;

L1=[a,b,c] L2=[ ]

```
concatena([ ],L,L).
concatena([A|B],C,[A|D]):-concatena(B,C,D).
```

O Predicado concatena

O predicado concatena permitiu:

- Juntar duas listas instanciadas, gerando uma terceira lista com a concatenação das duas primeiras
- Gerar todas as listas que concatenadas possam dar origem a uma lista instanciada

E se trocássemos a ordem das cláusulas?

```
concatena1([A|B],C,[A|D]):-concatena1(B,C,D).
concatena1([ ],L,L).
```

O Predicado concatena

A interacção agora seria a seguinte:

?- concatena1([a,b],[c,d,e],L).
L = [a,b,c,d,e]

?- concatena1(L1,L2,[a,b,c]).
L1 = [a,b,c] ,
L2 = [] ;

L1 = [a,b] ,
L2 = [c] ;

L1 = [a] ,
L2 = [b,c] ;

L1 = [] ,
L2 = [a,b,c]
……as soluções são as mesmas, mas geradas noutra ordem

O predicado inverte

- Hoje em dia as implementações de PROLOG já trazem o predicado **reverse**/2 que inverte a lista do 1º argumento originando a lista do 2º argumento.

- Mas se esse predicado não existisse poderia ser implementado do seguinte modo:

```
inverte(L,LI):-inverte1(L,[ ],LI).

inverte1([ ],L,L).
inverte1([X|L],L2,L3):- inverte1(L,[X|L2],L3).
```

O predicado inverte

- Vamos mudar ligeiramente o predicado para perceber o que se passa:

```
inverte(L,LI):-inverte1(L,[ ],LI).

inverte1([ ],L,L).
inverte1([X|L],L2,L3):-
   write('[X|L2]='), write([X|L2]),nl,
   inverte1(L,[X|L2],L3).
```

O predicado inverte

```
?- inverte([a,b,c],L).
[X|L2]=[a]
[X|L2]=[b,a]
[X|L2]=[c,b,a]
L = [c,b,a]
```

- Observe-se que na lista do 2º argumento de inverte1 foi sendo construída a lista invertida, a qual é passada para o 3º argumento de inverte1 quando a lista do 1º argumento fica [ ]

O predicado apaga

- Hoje em dia as implementações de PROLOG já trazem o predicado **delete**/3 que apaga todas as ocorrências do elemento do 1º argumento na lista do 2º argumento originando a lista do 3º argumento.

apaga(X,L,L1)

- Mas se esse predicado não existisse poderia ser implementado do seguinte modo:

```
apaga(_,[ ],[ ]).
apaga(X,[X|L],M):-!,apaga(X,L,M).
apaga(X,[Y|L],[Y|M]):-apaga(X,L,M).
```

O Predicado apaga

```
apaga(_, [], []).
apaga(X,[X|L],M):-!,apaga(X,L,M).
apaga(X,[X|L], [Y|M])):-apaga(X,L,M).
```

?-apaga(1,[2,1,3],L).

apaga(_, [], []). falha

apaga(1,[2|[1,3]],M):- !, apaga(X,L,M). falha

apaga(1,[2|[1,3]],[2|M]):- apaga(1, [1,3],M).

apaga(_, [], []). falha

apaga(1,[1|[3]],M):- !, apaga(1, [3],M).

apaga(_, [], []). falha

apaga(1,[3|[]],M):- !, apaga(X,L,M). falha

apaga(1,[3|[]],[3|M]):- apaga(1, [],M).

apaga(_, [], []). sucesso

O Predicado apaga

```
apaga(_, [], []).
apaga(X,[X|L],M):-!,apaga(X,L,M).
apaga(X,[Y|L], [Y|M])):-apaga(X,L,M).
```

?-apaga(1,[2,1,3], [2,3]). sucesso

apaga(_, [], []). falha

apaga(1,[2|[1,3]],M):- !, apaga(X,L,M). falha

apaga(1,[2|[1,3]], [2|[3]]):- apaga(1, [1,3], [3]). sucesso

apaga(_, [], []). falha

apaga(1,[1|[3]], [3]):- !, apaga(1, [3], [3]). sucesso

apaga(_, [], []). falha

apaga(1,[3|[]],M):- !, apaga(X,L,M). falha

apaga(1,[3|[]],[3|[]]):- apaga(1, [], []). sucesso

apaga(_, [], []). sucesso

O predicado apaga

```
apaga(_,[ ],[ ]).
apaga(X,[X|L],M):-!,apaga(X,L,M).
apaga(X,[Y|L],[Y|M]):-apaga(X,L,M).
```

- Vejamos uma interacção

```
?- apaga(1,[1,2,1,3,1,4],L).
L = [2,3,4] ;
no
```

- Como deveria ser o predicado se fosse pretendido que se apagasse apenas a primeira ocorrência do elemento na lista?

- O que aconteceria se fosse removido o "!" e pedidas várias soluções?

O predicado apaga

```
apaga(_,[ ],[ ]).
apaga(X,[X|L],M):-!,apaga(X,L,M).
apaga(X,[Y|L],[Y|M]):-apaga(X,L,M).
```

Apaga apenas a 1ª ocorrência:

```
apaga1(_,[ ],[ ]).
apaga1(X,[X|L],M):-!, apaga2(X,L,M).
apaga1(X,[Y|L],[Y|M]):- apaga1(X,L,M).

apaga2(_,[ ],[ ]).
apaga2(X,[Y|L],[Y|M]):- apaga2(X,L,M).
```

O predicado apaga

```
?- apaga(1,[1,2,1,3,1,4],L).
L = [2,3,4] ;
L = [2,3,1,4] ;
L = [2,1,3,4] ;
L = [2,1,3,1,4] ;
L = [1,2,3,4] ;
L = [1,2,3,1,4] ;
L = [1,2,1,3,4] ;
L = [1,2,1,3,1,4] ;
no
```

O predicado união

- O predicado união é diferente da concatenação na medida que as listas representam conjuntos, logo não podem ter elementos repetidos.

- Esse predicado pode ser implementado do seguinte modo:

```
união([ ],L,L).
união([X|L1],L2,LU):-membro(X,L2),!,união(L1,L2,LU).
união([X|L1],L2,[X|LU]):-união(L1,L2,LU).
```

O Predicado união

```
união([], L, L).
união([X|L1],L2,LU):-
                    membro(X,L2),!,união(L1,L2,LU).
união([X|L1],L2,[X|LU]):-união(L1,L2,LU).
```

?-união([1,2],[2,3],L).

união([], L, L). falha

união([1|[2]],[2,3],LU):- membro(1, [2,3]),!, ... falha

união([1|[2]],[2,3],[1|LU]):- união([2], [2,3], LU).

união([], L, L). falha

união([2|[]], [2,3], LU):- membro(2, [2,3]),!,união([], [2,3],LU).

união([], [2,3], [2,3]). sucesso

O Predicado união

```
união([], L, L).
união([X|L1],L2,LU):-
                membro(X,L2),!,união(L1,L2,LU).
união([X|L1],L2,[X|LU]):-união(L1,L2,LU).
```

?-união([1,2],[2,3],[1,2,3]).

união([], L, L). falha

união([1|[2]],[2,3],LU):- membro(1, [2,3]),!, ... falha

união([1|[2]],[2,3],[1|[2,3]]):- união([2], [2,3], [2,3]).

união([], L, L). falha

união([2|[]], [2,3], [2,3]):- membro(2, [2,3]),!,união([], [2,3], [2,3]).

sucesso

união([], [2,3], [2,3]). sucesso

O predicado união

```
união([ ],L,L).
união([X|L1],L2,LU):-membro(X,L2),!,união(L1,L2,LU).
união([X|L1],L2,[X|LU]):-união(L1,L2,LU).
```

- Vejamos uma interacção

  ?- união([1,2,3,4],[1,3,5,7],L).

  L = [2,4,1,3,5,7]

- Justifique por que razão a solução obtida não é [1,2,3,4,5,7]

- O que aconteceria se fosse removido o "!" e pedidas várias soluções?

O predicado união

```
união([ ],L,L).
união([X|L1],L2,LU):-membro(X,L2),união(L1,L2,LU).
união([X|L1],L2,[X|LU]):-união(L1,L2,LU).

?- união([1,2,3,4],[1,3,5,7],L).
L = [2,4,1,3,5,7] ;
L = [2,3,4,1,3,5,7] ;
L = [1,2,4,1,3,5,7] ;
L = [1,2,3,4,1,3,5,7]
```

- Justifique o número de soluções encontradas e a ordem pela qual as soluções aparecem

O predicado intersecção

- O predicado intersecção é análogo ao união, desde que estejamos a pensar em termos de conjuntos.

- Esse predicado pode ser implementado do seguinte modo:

```
intersecção([ ],_,[ ]).
intersecção([X|L1],L2,[X|LI]):-
                    membro(X,L2),!,intersecção(L1,L2,LI).
intersecção([_|L1],L2, LI):- intersecção(L1,L2,LI).
```

O predicado intersecção

```
intersecção([ ],_,[ ]).
intersecção([X|L1],L2,[X|LI]):-membro(X,L2),!,intersecção(L1,L2,LI).
intersecção([_|L1],L2, LI):- intersecção(L1,L2,LI).
```

- Vejamos uma interacção

  ?- intersecção([1,2,3,4],[1,3,5,7],L).

  L = [1,3]

- O que aconteceria se fosse removido o "!" e pedidas várias soluções?

O predicado intersecção

```
intersecção([ ],_,[ ]).
intersecção([X|L1],L2,[X|LU]):-membro(X,L2),intersecção(L1,L2,LU).
intersecção([_|L1],L2, LU):- intersecção(L1,L2,LU).
```

```
?- intersecção([1,2,3,4],[1,3,5,7],L).
L = [1,3] ;
L = [1] ;
L = [3] ;
L = []
```

- Justifique o número de soluções encontradas e a ordem pela qual as soluções aparecem

Alguns aspectos complementares da linguagem PROLOG

Vamos agora abordar um conjunto de aspectos complementares, incluindo:

- A conversão de strings em listas de códigos
- A carga de programas e Bases de Conhecimento
- A alteração dinâmica do conhecimento
- A entrada/saída usando ficheiros
- Functores, Argumentos e =..
- Soluções Múltiplas
- Definição dinâmica de operadores

Conversão de strings em listas de códigos

O predicado name converte o átomo do 1º argumento numa lista de códigos de caracteres que aparecem no 2º argumento e vice-versa

name/2 é usado para construir/decompor átomos

?- name(abc,L).

L = [97,98,99]

?- name(A,[97,97,98,99,98,99]).

A = aabcbc

Conversão entre átomos e strings

O predicado **string_to_atom** permite efectuar conversões entre átomos e strings, ou vice-versa, dependendo de qual a variável que está instanciada.

string_to_atom(Atomo, String)

Exemplo:

string_to_atom(start, S)

(**S** ficará instanciada com a string `start`)

string_to_atom(A, `start`)

(**A** ficará instanciada com o átomo start)

Conversão entre números e strings

O predicado atom_number permite efectuar conversões entre inteiros e átomos, ou vice-versa, dependendo de qual a variável que está instanciada.

atom_number(Atomo,Número)

Exemplo:

atom_number(A, 123)

(**A** ficará instanciada com o átomo '123')

atom_number(123, N)

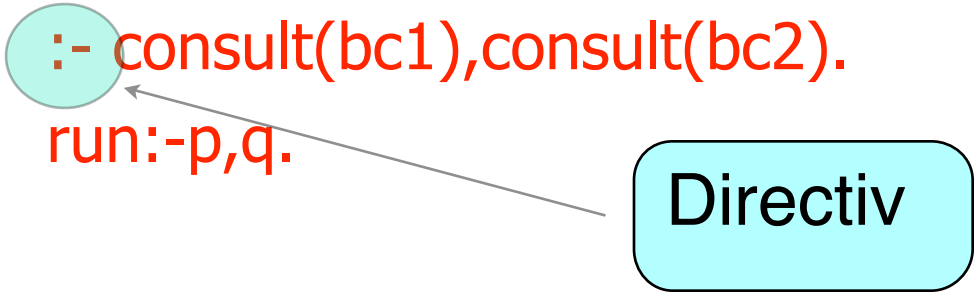
(**N** ficará instanciada com o inteiro 123)

Carga de programas e Bases de Conhecimento

- A maioria das implementações de PROLOG permitem que a carga de uma Base de Conhecimento seja feita através de um Menu
- Essa carga pode ser feita programaticamente a partir de um ficheiro em PROLOG, usando:

  [filename] ou consult(filename)
- É comum que um programa PROLOG comece pela carga de outros ficheiros auxiliares:

```
:- consult(bc1),consult(bc2).
run:-p,q.
```



Carga de programas e Bases de Conhecimento

- Vejamos o conteúdo dos ficheiros bc1 e bc2:
  - bc1 contém:

    p:-write(p).

  - bc2 contém:

    q:-write(q), nl.

Vejamos o que acontece quando se faz a consulta de bc:

```
# 0.000 seconds to consult c:\carlos\prolog\bc1.pl
# 0.000 seconds to consult c:\carlos\prolog\bc2.pl
# 0.000 seconds to consult c:\carlos\prolog\bc.pl
?- run.
pq
yes
```

Carga de programas e Bases de Conhecimento

- Quando num programa aparece o :- sem nada à esquerda isso significa que nesse momento o que está do lado direito é executado automaticamente, mesmo antes de ser interpretado o resto do programa

  Exemplos de directivas:

  :- initialization

  :- dynamic

- Em qualquer momento podemos reconsultar um ficheiro escrito em PROLOG, usando:

  [-filename] ou reconsult(filename)

Alteração dinâmica do conhecimento

- Podemos adicionar novos factos, e mesmo regras na Base de Conhecimento com o predicados assert, asserta ou assertz
- É possível retirar factos e regras com o predicado retract
- Contudo, tais factos ou regras têm que ser declarados como sendo dinâmicos através de :-dynamic predicado/aridade

Alteração dinâmica do conhecimento

Um exemplo:

```
:-dynamic figura/2.

figura(hexágono,6).

cria_figuras:-
    assertz(figura(triângulo,3)),
    asserta(figura(quadrado,4)),
    assertz(figura(pentagono,5)).
```

```
?- figura(F,NL).
F = hexágono ,
NL = 6

?- cria_figuras.
yes

?- figura(F,NL).
F = quadrado ,
NL = 4 ;

F = hexágono ,
NL = 6 ;

F = triângulo ,
NL = 3 ;

F = pentagono ,
NL = 5
```

Alteração dinâmica do conhecimento

Um exemplo:

```
:-dynamic figura/2.

figura(hexágono,6).

cria_figuras:-
    assertz(figura(triângulo,3)),
    asserta(figura(quadrado,4)),
    assertz(figura(pentagono,5)).
```

```
?- retract(figura(triângulo,X)).
X = 3

?- retract(figura(X,Y)).
X = quadrado
Y = 4
```

Alteração dinâmica do conhecimento

Um exemplo criando uma regra:

```
:-dynamic r/2.

cria_regra:-
    asserta((r(X,Y):-r1(X,Z),r2(Z,Y))).

r1(a,b).
r1(a,c).
r1(a,d).

r2(b,e).
r2(b,f).
r2(d,g).
```

```
?- r(X,Y).
no

?- cria_regra.
yes

?- r(X,Y).
X = a ,
Y = e ;

X = a ,
Y = f ;

X = a ,
Y = g
```

Factos dinâmicos

No Win-Prolog, um facto "asserted" por um predicado ou directamente na consola é, por defeito, **dinâmico**. Como tal, pode ser "retracted".

Um facto pré-existente ou consultado de um ficheiro é, por defeito, **estático**. Como tal, não pode ser "retracted" por estar **protegido**.

A declaração de um facto como **dynamic** permite que possa ser retracted mesmo após consulta.

Após consulta deste ficheiro será possível fazer o retract de foo/1 mas não de bar/1

```
:- dynamic foo/1.
foo( hello ).
bar( world ).
```

listing

listing fornece a lista de todos os predicados dinâmicos existentes no momento

```
?- assert( foo(hello,there) ), assert( bar(world) ).
<enter>
yes
?- listing. <enter>
% foo/2
foo( hello, there ).
% bar/1
bar( world ).
yes
```

Entrada/Saída usando ficheiros

(1) No início direccionados para a a consola

Entrada de Dados:

see(filename) – as operações de leitura passarão a ser feitas usando o ficheiro filename (é o novo input stream)

seen – fecha o input stream

seeing(F) – F toma o valor do input stream

Saída de Dados:

tell(filename) – as operações de escrita passarão a ser feitas usando o ficheiro filename (é o novo output stream)

told – fecha o output stream

telling(F) – F toma o valor do output stream

```
...
tell(file1)
write2file(Info)
tell(user)
...
```

```
tell(prolog(foo),
write(`hello.`), nl, told.
```

```
?- telling(C), tell( foo), write(`abc`),
tell(C). <enter>
C = user
```

Functores, Argumentos

Termo – estrutura com 1 functor e argumentos (tal como acontece nos factos)

functor(T,F,N) – o termo T tem o functor F e N argumentos

```
?- functor(func(a,b,c),F,N).
F = func ,
N = 3
```

```
?- functor(D,data,3),
arg(1, D, 23),
arg(2, D, maio),
arg(3, D, 1956).
D = data(23, maio, 1956)
```

arg(N,T,A) – A é o n^{mo} argumento de T

```
?- arg(2,func(a,b,c),A).
A = b
```

=..

O =.. Converte o termo que está a sua esquerda numa lista, cuja cabeça é o functor do termo e os outros elementos são os diversos argumentos do termo

```
?- pred(a,b,c)=..L.
L = [pred,a,b,c]

?- T=..[pred,a,b,c].
T=pred(a,b,c)
```

Soluções Múltiplas

Quando uma questão é colocada o PROLOG dá as soluções uma a uma, carregando-se em ;

Mas tal é pouco prático, existindo predicados que colocam todas as soluções numa lista

bagof(X,Q,L) – L é a lista dos X que satisfazem Q, se não houver solução bagof falha

setof(X,Q,L) – L é a lista dos X que satisfazem Q, L vem ordenada, elementos repetidos são eliminados, se não houver solução setof falha

findall(X,Q,L) – L é a lista dos X que atendem a Q, se não houver solução findall tem sucesso com L=[]

Soluções Múltiplas - Exemplos

```
idade(pedro,9).
idade(maria,8).
idade(clara,12).
idade(rosa,8).
```

```
?- bagof(Criança, idade(Criança,8), Lista).
Lista = [maria, rosa]
```

```
?- setof(Cria/Id, idade(Cria,Id), Lista).
Lista = [clara/12, maria/8, pedro/9, rosa/8]
```

```
?- findall(Idade, idade(Criança,11), Lista).
Lista = [ ]
```

Soluções Múltiplas

```
f(a,1).
f(a,2).
f(a,2).
f(z,6).
f(z,5).
f(x,4).
f(x,3).
```

```
?- findall(f(L,N),f(L,N),Lista).
L = _ ,
N = _ ,
Lista = [f(a,1),f(a,2),f(a,2),f(z,6),f(z,5),f(x,4),f(x,3)]

?- setof(f(L,N),f(L,N),Lista).
L = _ ,
N = _ ,
Lista = [f(a,1),f(a,2),f(x,3),f(x,4),f(z,5),f(z,6)]

?- bagof(f(L,N),f(L,N),Lista).
L = _ ,
N = _ ,
Lista = [f(a,1),f(a,2),f(a,2),f(z,6),f(z,5),f(x,4),f(x,3)]

?- findall(f(L,N),(f(L,N),N>7),Lista).
L = _ ,
N = _ ,
Lista = []

?- setof(f(L,N),(f(L,N),N>7),Lista).
no

?- bagof(f(L,N),(f(L,N),N>7),Lista).
no
```

Soluções Múltiplas

O **findall** é muito usual:

```
findall(_,cruzamento,_)

findall( L, linha(_,L), LE)

findall( Linha, ( linha(Linha,LEL), member(E,LEL) ), LL)

findall( _,
      ( estações(LE), member(E,LE), todas_linhas(E,LL),
       assertz(estação_linhas(E,LL))),
      _)

findall( h(L,C,Q,LN), member( h(L,C,Q,LN), LLH ), LLHQ)
```

forall

forall(Goal1, Goal2)

Testa se um dado predicado é verdadeiro para todos os casos de um outro.

Terá sucesso se
para todas as soluções de **Goal1**, **Goal2** for verdadeiro

forall

Posição desta linha!

```
nameof( 1, one ).
nameof( 2, two ).
nameof( 3, three ).
nameof( 4, four ).
nameof( 5, five ).
nameof( 6, six ).
nameof( 7, seven ).
nameof( 8, eight ).
nameof( 9, nine ).
nameof( 10, ten ).
evens :-
    forall(   ( nameof( Number, Name ),
                0 is Number mod 2
              ),
              ( write( Name ),
                write( ` is even` ),
                nl
              )
            ).
```

Um exemplo integrador

Vamos considerar a seguinte Base de conhecimento:

```
capital(portugal,lisboa).
capital(espanha,madrid).
capital(frança,paris).
capital(itália,roma).
capital(inglaterra,londres).
```

Vamos escrever um programa que crie na Base de Conhecimento factos com o seguinte formato

```
nome_país(nome_capital)
```

Um exemplo integrador

```
converte:-    findall((X,Y),capital(X,Y),L),
              assert_país_capital(L).

assert_país_capital([(X,Y)|L]):-T=..[X,Y],
                                assertz(T),
                                assert_país_capital(L).
assert_país_capital([]).
```

Note-se que os novos factos só ficam residentes em memória (e não no ficheiro com o programa)

Um exemplo integrador

Vamos resolver o mesmo problema enviando os novos factos para um ficheiro "pais_cap" que seja posteriormente consultado

```
converte1:-     findall((X,Y),capital(X,Y),L),
                tell(pais_cap),
                write_file(L),
                told,
                consult(pais_cap).

write_file([(X,Y)|L]):-T=..[X,Y],
                        write(T),write('.'),nl,
                        write_file(L).
write_file([ ]).
```

Operadores

Directiva **op**(Precedência,Notação,Nome)

A precedência é um valor numérico

A Notação pode ser:

- fx: prefixa
- xf: pósfixa
- xfx,xfy,yfx: infixa

A um mesmo nível são aplicados os operadores que têm o valor numérico de precedência inferior. Para ultrapassar ambiguidades podemos usar os (), cuja precedência é nula.

Operadores

- Não está normalmente associada a um operador qualquer operação sobre dados
- Operadores são usados (como os functors) para associar objectos em estruturas.

em **xfy**
x representa um argumento com precedência *menor* que a do functor ou operador
y representa um argumento com precedência *maior* que a do functor ou operador

Operadores

```
tem(pedro, informação)

:- op(600, xfx, tem)

pedro tem informação
```

Operadores

```
:-op(700,xfy,ou).
:-op(600,xfy,e).
:-op(500,fy,nao).

X ou Y :- X;Y.
X e Y :- X,Y.
nao X :- not X.

a.
b.
c:-fail.
```

```
?- a e b.
yes
?- a ou b.
yes
?- a e c.
no
?- nao a.
no
?- nao c.
yes
```