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

- **As variáveis podem residir num de dois estados**: não instanciadas ou instanciadas. Quando é encontrada uma solução podem ser exibidas as instanciações possíveis

- Quando algo não está explicitamente definido como um axioma é assumido como sendo falso (**Assumpção do Mundo Fechado**). Há várias extensões ao PROLOG que assumem uma lógica tri-valor (verdadeiro, falso ou desconhecido)

- Em PROLOG é possível **criar/remover dinamicamente axiomas**

Conceitos Básicos do PROLOG

Em PROLOG temos 3 conceitos básicos:

Cláusulas

[a ser colocadas no ficheiro da Base de Conhecimento]

- **Factos**

correspondem a axiomas

```
rio(douro).
pai(pedro,ana).
```

- **Regras**

correspondem a implicações

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

Questões (à Base de Conhecimento)

[a partir da consola, a seguir ao prompt ?-]
[as respostas correspondem a soluções possíveis]

```
?- pai(P,ana).
?- neto(rui,A).
```

Factos

Um facto é composto por

- Um **functor** (nome genérico, começado por uma minúscula),
- Zero ou mais **argumentos**, englobados por parêntesis e separados por vírgulas,

e termina com um **ponto** (o **finalizador** do PROLOG).

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).
passa(minho,orense).
```

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). falha
passa(tejo,toledo). falha
passa(minho,caminha). falha
passa(minho,orense). falha

?-fica(madrid,espanha).
yes

?-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).
passa(minho,orense).
```

```
?-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).
passa(minho,orense).

```
?-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).
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

sucesso

```
?-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

Termo

Sequência de termos

passa(Aluno, Media, Faltas) :- Media >= 9.5, Faltas < 5.

Conclusão a provar

Condições

Implicação

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 precisamos conhecer o valor

Exemplos de Regras

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

- Definição recursiva da potência inteira e não negativa de um número
- A recursividade é muito comum nas regras do PROLOG
- O ! (cut) é uma primitiva de controlo a ser 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

- 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 instanciada com **porto**, ou seja, **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

- na chamada à regra, do lado esquerdo, R fica instanciada com o valor **tejo**;
- No lado direito da regra, a 1ª chamada já é feita como **passa(tejo,C)** e tem sucesso com R=tejo e C=lisboa;
- a chamada seguinte já é feita com C instanciada com **lisboa**, ou seja, **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;
- No lado direito da regra, a 1ª chamada já é feita como **passa(tejo,C)** e tem sucesso com R=tejo e C=toledo;
- a chamada seguinte já é feita com C instanciada com **toledo**, ou seja, **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 instanciada com **lisboa**, ou seja, **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).      suc.(1).  suc.(2)
passa(douro,zamora).     suc.(3)
passa(tejo,lisboa).
passa(tejo,toledo).
passa(minho,caminha).
passa(minho,orense).
```

?-banhadas_mesmo_rio(C1,C2).

C1=porto C2=zamora

yes

- na chamada à regra, do lado esquerdo, C1 e C2 continuam não instanciadas;
- No lado direito da regra, 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 instanciada com **douro**, ou seja, **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 comportamento diferente das variáveis em 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 ao estado de não instanciada.

Incremento duma variável em PROLOG

Errado - N is N + 1

- 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

Correcto - N1 is N + 1

- Uso de uma variável auxiliar

Nota: **is** é o operador de atribuição numérica

Variáveis em PROLOG

- Se for pedida a impressão de uma variável não instanciada aparecerá um nº antecedido de _ (por ex. _22456) que representa a referência da variável e não 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

Consideremos 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).
```

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 LPA-PROLOG:

abs(X)	valor absoluto de X
acos(X)	arco-cosseno de X (graus)
aln(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)
fp(X)	parte não inteira de X (mesmo sinal que X)
int(X)	inteiro igual ou imediatamente anterior a X

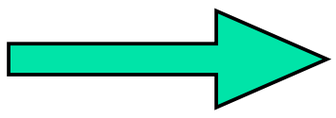
Funções Aritméticas em PROLOG

ip(X)	parte inteira de X
ln(X)	logaritmo natural de X
log(X)	logaritmo decimal de X
max(X,Y)	máximo entre X e Y
min(X,Y)	mínimo entre X e Y
rand(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 ou não instanciadas 

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, deverão estar instanciadas
- Do lado esquerdo a variável não deve estar instanciada, senão ocorre uma falha

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 usar o **put**
  - **put(65)** – escreve o caracter A (código ASCII 65)

- A mudança de linha é feita com o **nl**
- **tab(Espaços)**

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 usar 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 c
X = 97 ,
Y = 98 ,
Z = 13
```

Recursividade

- O uso da **recursividade** é muito comum em PROLOG
- Na implementação de um predicado recursivo 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.

?-factorial(3,F).

factorial(0,1) falha

factorial(3,F):-N1 ← 3-1, factorial(2,F1), F is 3*2. sucesso (c/ F ← 6)

F = 6

factorial(0,1) falha

factorial(2,F):-N1 ← 2-1, factorial(1,F1), F is 2*1. sucesso (c/ F ← 2)

factorial(0,1) falha

factorial(1,F):-N1 ← 1-1, factorial(0,F1), F is 1*1. sucesso (c/ F ← 1)

factorial(0,1):-!. sucesso

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 chega ao *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

- Vejamos um exemplo onde o **true** faz sentido

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

- Sem a alternativa true ocorreria uma falha caso C não fosse portuguesa, mas o que se quer é que tenha sucesso se C for uma cidade e para o caso particular de ser portuguesa que apareça uma mensagem escrita.

repeat

Exemplo do uso do **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.
```

! (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 desnecessárias
- Por vezes é redutor, permitindo 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).

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 ";" ;

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).
```

sucesso

sucesso

falha

sucesso(1ªsol.) falha

sucesso(2ªsol.) falha

falha sucesso(3ªsol.)

falha

falha

- 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

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).
```

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 da direita do ! para a sua esquerda

- 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 após se ter atingido o sucesso por uma das alternativas das regras se impeça que por backtracking se chegue lá 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

- A falha ocorreu porque, se 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, o segundo argumento tomará valores negativos (-1,-2,-3,...) nos níveis de recursividade que se seguem, até que a Stack fique cheia.

- Não adianta aumentar a dimensão da Stack
- A solução consiste em usar 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
```

fail

- O **fail** obriga à ocorrência de uma falha, é útil no *raciocínio pela negativa*
- Exemplo:

```
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).
```

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.

Listas

- As listas podem ser representadas
  - pela **enumeração** dos seus elementos separados por vírgulas e envolvidos por parênteses rectos
    por exemplo [a,b,c,d]
  - pela **notação cabeça-cauda** separadas pelo | e envolvidas por **[ ]**
    por exemplo [H|T]

- Os elementos das listas podem ser de tipos diferentes:

  [a,2,abc,N,[x,1,zzz]]

Listas

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

```
?- [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**

- Já existe o predicado **member/2** (dois argumentos, aridade 2) que verifica se o 1º argumento é membro da lista do 2º argumento.
- Se tal predicado não existisse poderia ser implementado do seguinte modo:

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

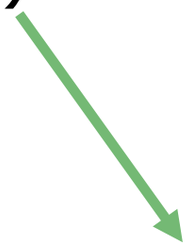
O Predicado **membro**

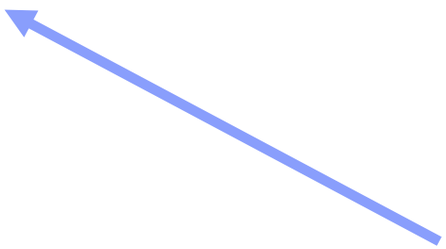
?-membro(b,[a,b,c]).

yes

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

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





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

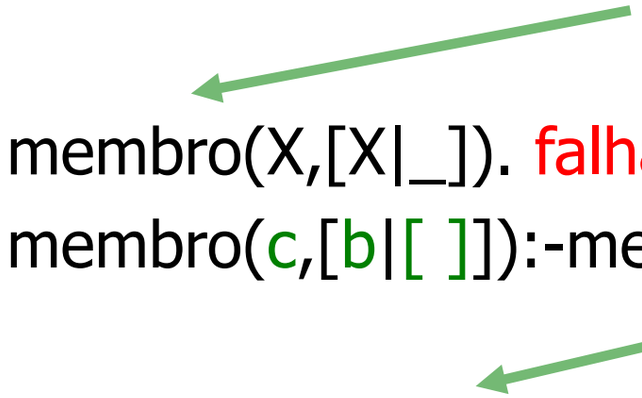
```
membro(X,[X|_]).
membro(X,[_|L]) :- membro(X,L).
```

O Predicado membro

?-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]).

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

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

no

falha porque não há mais cláusulas possíveis

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

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

falha porque não há mais cláusulas possíveis

membro(X,[X|_]). falhou, [X|_] não foi instanciável com [ ]

membro(c,[_|L]):- falhou, [_|L] não foi instanciável com [ ]

O Predicado membro

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

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

X=a

sucesso, como X tomou o valor a no 2º argumento, o 1º argumento também fica com o valor a

Se pedirmos uma alternativa com ; ocorrerá uma falha e será tentada a 2ª cláusula

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

O Predicado membro

Continuação após pedir alternativa

à 1ª solução X=a ;

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

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

X=b

membro(b,[b|[c]]). sucesso, como X tomou o valor b no 2º argumento, o 1º argumento também fica com o valor b

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

O Predicado membro

Vejamos 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 interacção 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

E se trocássemos a ordem das cláusulas?

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

Dá-se prioridade à visita da cauda da lista, antes de verificar o conteúdo da cabeça

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. nesse processo, verificará quais os elementos que estão à cabeça, falhando sempre até chegar ao 1º nível, quando ocorre o sucesso

O Predicado concatena

- Existe já 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:

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

O predicado 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

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

O Predicado concatena

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

concatena([],L,L). falha

concatena([a|[b]],[c,d,e],[a|L3]):- concatena([b],[c,d,e],L3)

concatena([],L,L). falha

concatena([b|[ ]],[c,d,e],[b|L3]):- concatena([ ],[c,d,e],L3)

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

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

O Predicado concatena

?-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

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

O predicado inverte

- Já existe 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).
```

Acumulador

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 copiada 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 as ocorrências do 1º argumento na lista do 2º argumento originando a lista do 3º argumento.

- 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).
```

apaga(X,L,L1)

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
```

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],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

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

```
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).
```