Tutorial Ligeiro de Prolog

Parte I

Ambiente de Desenvolvimento e Interpretador

Inicie o interpretador `swi-prolog` executando em LINUX o comando `swi-prolog` ou seleccionando em Windows o programa *`swi-prolog`*.

```
Last login: Sun Feb  5 19:21:22 on console
/Applications/swipl-5.10.4/bin/i386-darwin10.7.0/swipl ; exit;
MacBook-Pro-de-Paulo-Urbano:~  paulourbano$  /Applications/swipl-5.10.4/bin/i386-
darwin10.7.0/swipl ; exit;
% library(swi_hooks) compiled into pce_swi_hooks 0.01 sec, 3,928 bytes
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 5.10.4)
Copyright (c) 1990-2011 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

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

?-
```

Inicie o editor de texto (o editor emacs tem um modo específico para escrita de programas em PROLOG que pode ser obtido fazendo ESC+x+"prolog-mode"+ENTER)

Os Factos

Em Prolog podemos fazer declarações utilizando factos. Os factos representam relações entre entidades e podem ter 0 ou mais argumentos.

2.1 Factos simples

Os factos podem ser simples, (sem argumentos) e é o caso do facto

```
joao_canta.
```

que pode ser interpretado como a declaração que o João está a cantar. Ou do facto

```
joao_muito_alto.
```

em que declaramos que o João é muito alto.

Os factos devem sempre começar com uma letra minúscula e acabar num ponto final e podem consistir numa combinação de letras e números e do _.

2.2 Factos com Argumentos

Podemos ter factos com argumentos (um número arbitrário) que representam a relação e as coisas implicadas na relação, representadas através de constantes. Os argumentos estão entre parêntesis e são separados por vírgulas.

relacao(<arg1>,<arg2>,…,<argn>).

Podemos declarar que o joão é baixinho através do facto:

```
baixinho(joao).
```

Não estaria errada (sintacticamente) a declaração

```
joao(baixinho).
```

Mas, faz muito mais sentido ter as entidades como argumentos e ter a relação ou propriedade à cabeça do facto.

Por exemplo, podemos declarar que o João viu a Diana através do facto:

```
viu(joao, diana).
```

Mas se soubermos que a viu em Nápoles, teremos por exemplo o seguinte facto:

```
viu(joao, diana, napoles).
```

Reparem que estamos a usar factos com o mesmo nome para a relação. Não há problema porque têm uma aridade ou nº de argumentos diferentes (“overloading”).

Os argumentos podem ser termos Prolog. Os termos Prolog básicos são inteiros, um átomo, uma variável ou uma estrutura. Podemos também ter “floats” e “strings”.

3 Consulta de programas

Considere o programa seguinte, (só com factos) no ficheiro viu.pl. Temos informções sobre pessoas que viram outras pessoas (homens ou mulheres) em diversos lugares que podem ser vilas, cidades ou aldeias. Se quiséssemos utilizar maiúsculas para os nomes poderíamos utilizar ‘Joao’, ‘Napoles’ ou ‘Diana’, para os distinguirmos das variáveis

```
% O João é muito alto

joao_muito_alto.

% O Joao canta

joao_canta.

% as mulheres
mulher(diana).
mulher(ausenda).

% os homens
homem(joao).
homem(pedro).
homem(gonzaga).
homem(gasosa).

% As cidades
cidade(napoles).
cidade(luanda).
cidade(cartagenaDasIndias).

% as vilas
vila(vilaVerde).

% as aldeias
aldeia(pitoesDasJunias).

% O João viu a Diana em Nápoles
viu(joao,diana,napoles).
```

```
% O João viu o Pedro em Piões das Júnias
viu(joao,pedro,pitoesDasJunias).

% O João viu a Ausenda em Nápoles
viu(joao,ausenda,napoles).

% O João viu o Pedro em Vila Verde
viu(joao,pedro,vilaVerde).

% A Diana encontrou o Gonzaga em Cartagena das Índias
viu(diana,gonzaga,cartagenaDasIndias).

% A Diana foi vista pelo Gasosa em Luanda
viu(gasosa,diana,luanda).

% A Diana viu o João em Nápoles
viu(diana,joao,napoles).
```

Depois de escrever o texto no editor (os factos e as regras) é necessário carregar o ficheiro pitoesDasjunias.pl, que está na página da disciplina. Para isso é necessário gravar e consultar o ficheiro (usar o comando **`consult('path/nome_do_ficheiro')`** no interpretador.

```
?- consult('/Users/guest/Dropbox/IIA-2014-
15/Prolog/TutorialLightPrologPlus/pitoesJunias.pl').
% /Users/guest/Dropbox/IIA-2014-
15/Prolog/TutorialLightPrologPlus/pitoesJunias.pl compiled
0.00 sec, 216 bytes
true.
```

Fazendo *listing,* pode-se ver o conteúdo que foi consultado e que está na memória de trabalho.

```
?- listing.

mulher(diana).
mulher(ausenda).

homem(joao).
homem(pedro).
homem(gonzaga).
```

```
homem(gasosa).

cidade(napoles).
cidade(luanda).
cidade(cartagenaDasIndias).

joao_muito_alto.

vila(vilaVerde).

joao_canta.

aldeia(pitoesDasJunias).

viu(joao, diana, napoles).
viu(joao, pedro, pitoesDasJunias).
viu(joao, pedro, vilaVerde).
viu(diana, gonzaga, cartagenaDasIndias).
viu(gasosa, diana, luanda).
viu(diana, joao, napoles).
true.
```

4 Perguntas sem variáveis

Fazer uma pergunta ("query") é a maneira de retirar informação de um programa. Responder a uma pergunta é determinar se esta é uma consequência lógica do programa. De modo informal, uma "query" é uma pergunta ("goal") que é submetida ao Prolog para determinar se é verdadeira ou falsa (*true* ou f*alse*).

Vamos fazer primeiro perguntas relativas a factos sem argumentos.

```
?- joao_canta.
true.

?- joao_nao_canta.
ERROR: toplevel: Undefined procedure: joao_nao_canta/0
(DWIM could not correct goal)
```

Poderíamos esperar que uma pergunta como "joao_nao_canta" devesse devolver false. No entanto só são avaliados como *true* ou *false* os predicados definidos e, neste caso, joao_nao_canta não foi definido. Sendo assim, o *false* nunca pode aparecer como resposta a uma pergunta com 0 argumentos.

Por exemplo, se quisermos saber se o João viu a Diana em Napoles, fazemos uma "query" em que submetemos o "goal" viu(joana, diana, napoles). Após a resposta *true* colocamos um *;* para pedir mais soluções.

```
?- viu(joao,diana,napoles).
true ;
false.
```

Como não há mais factos que unificam com a pergunta então é devolvido f*alse*, que deve ser lido como "não há mais soluções".

Mas, se perguntarmos se o João viu a Tecla em Londres obteremos um rotundo *false*. Se a pergunta se referir a um facto cuja relação não existe… reaparecerá de novo a queixa de predicado não definido.

```
?- viu(joao,tecla,londres).
false.

?- falou(joao,diana,napoles).
ERROR: toplevel: Undefined procedure: falou/3 (DWIM could
not correct goal)
```

Podemos fazer "queries" compostas, com conjunções de "goals". Os "goals" são separados por vírgulas e a satisfação das "queries" dá-se da esquerda para a direita. O interpretador de Prolog logo que satisfaz uma "query", avança para a seguinte. Só devolve *true* se todas as "queries" atómicas forem satisfeitas (verdadeiras).

```
?- homem(joao),viu(joao,diana,napoles).
true

?- mulher(joao),viu(joao,diana,napoles).
false.

?- viu(joao,diana,napoles),homem(diana).
false.
```

Nas últimas duas "queries" apenas um dos elementos da conjunção é satisfeito e elas falham, devolvendo *false*.

5 Variáveis Lógicas e Unificação

Uma variável lógica refere-se a qualquer entidade não especificada e começa com uma letra maiúscula, podendo conter letras, dígitos ou _. `X`, `Pessoa`, `PessoaVerde` são exemplos de variáveis.

As variáveis lógicas diferem muito das variáveis habituais das linguagens imperativas. O termo "lógica" está a distinguir as variáveis da programação em lógica

das variáveis das linguagens imperativas em que se referem a uma localização da memória podendo conter diferentes valores em diferentes momentos de execução de um programa.

Uma variável Prolog pode ser instanciada com qualquer termo: constante, variável ou estrutura: um exemplo de estrutura é par(2,3). Durante a execução de um "goal" uma variável lógica, depois de instanciada, nunca pode mudar de valor excepto se acontecer uma falha e um retrocesso. Qualquer tentativa de instanciar novamente uma variável, i.e. que já esteja instanciada, com um valor diferente do actual, dará origem a uma falha na unificação e dará início ao retrocesso até ao ponto de escolha mais recente. Se duas variáveis forem unificadas, elas serão vistas como idênticas no processamento subsequente.

O alcance ("scope") de uma variável é uma cláusula; dentro desta cláusula, as variáveis com o mesmo nome são consideradas idênticas. As variáveis com o mesmo nome mas que ocorrem em diferentes cláusulas são variáveis diferentes. O alcance de uma variável não abrange mais do que uma "query".

A unificação **(=)** quer dizer "matching" múltiplo e não corresponde a uma relação de igualdade. Será que dois objectos se podem transformar através de substituições, de modo a que correspondam?

Duas constantes ou dois números unificam apenas se forem iguais.

```
?- papa = zaza.
false.

?- papa = papa.
true.

?- 4 = 3.
false.

?- 4 = 4.
true.
```

Vamos agora ver como é o processo de unificação usando variáveis. Comecemos por confirmar que uma variável unifica com uma constante ou com um número ou com qualquer coisa e que a mesma variável não mantém os valores de "query" para "query", começando sem valor quando aparece pela primeira vez na "query".

```
?- X = pedro.
X = pedro.

?- X = manuel.
X = manuel.
```

```
?- X = 4.
X = 4.

?- X = tata(5,6).
X = tata(5, 6).
```

Um erro típico consiste em tentar fazer com que uma variável mude de valor, mas isso é impossível em Prolog, a não ser devido a uma falha que force o retrocesso. Perceberemos o retrocesso mais à frente. Na "query" vamos tentar fazer com que uma variável mude de valor.

```
?- X = 1, X = 2.
false.
```

O que acontece é que após a primeira componente da conjunção, X unifica com 1 e a segunda componente da conjunção falha porque 1 não unifica com 2.

Mais umas unificações:

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

?- gosta(X,carlos) = gosta(diana,Y).
X = diana,
Y = carlos.

?- X = Y, Y = Z, Z = 3.
X = Y, Y = Z, Z = 3.
```

Reparem que na última pergunta ou "query", X unifica com Y, não tendo ambos valor ainda, o que implica que posteriormente, se X unificar com outra expressão, o mesmo acontecerá com Y. Como X unifica com Y e Y com Z então X unifica com Z e se Z unifica com 3 posteriormente, o mesmo acontecerá com X e Y.

Para verificar que dois termos não são unificáveis utilizamos o operador \=.

```
?- 2 \= tata(10).
true.

?- tata(X) \= tata(Y).
false.
```

```
?- carlos \= manuel.
true.

?- X = 2, X \= 4.
X = 2.
```

Uma constante não unifica com outra constante que seja diferente e por isso carlos não unifica com manuel. tata(X) unifica com tata(Y) porque duas estruturas são unificáveis se tiverem o mesmo símbolo de predicado, neste caso tata, se tiverem o mesmo número de argumentos, neste caso 1, e se cada um dos argumentos for também unificável, neste caso X unifica com Y. Como tata(X) unifica com tata(Y), a query "tata(X) \= tata(Y)" falha.

6 Perguntas com variáveis

Por exemplo, se quisermos saber onde viu o João a Diana, perguntaremos:

```
?- viu(joao,diana,Onde).
Onde = napoles ;
false.
```

A escolha do nome Onde para a variável foi apenas porque faz sentido, mas podíamos usar um X, Xtatata, ou Lugar (qualquer palavra que começe com uma maiúscula e que seja válida sintáticamente para o Prolog).

O que o Prolog faz é ir a cada facto, pela ordem de cima para baixo, com viu como símbolo de predicado e com 3 argumentos e tentar unificá-lo com a "query". Neste caso só há um facto: *viu(joao,diana,napoles)*.

Note que ao pedirmos mais soluções através de um ; a resposta foi false. Não existe nenhum facto que indique que o João tenha visto a Diana noutro lugar. Mas se perguntarmos quem foi visto pelo João em Nápoles, teremos duas respostas, uma de cada vez.

```
?- viu(joao,Quem,napoles).
Quem = diana ;
Quem = ausenda ;
false.
```

Se quisermos perguntar quem é que o João viu em Nápoles que também o tenha visto, teremos de usar uma conjuncão e perguntaremos:

```
?- viu(joao,X,napoles),viu(X,joao,napoles).
X = diana ;
false.
```

Mas claro que poderemos fazer perguntas com mais variáveis, tal como:

```
?- viu(Viu,Visto,Onde).
Viu = joao,
Visto = diana,
Onde = napoles ;
Viu = joao,
Visto = pedro,
Onde = pitoesDasJunias ;
Viu = joao,
Visto = ausenda,
Onde = napoles ;
Viu = joao,
Visto = pedro,
Onde = vilaVerde ;
Viu = gasosa,
Visto = bonga,
Onde = luanda ;
Viu = diana,
Visto = joao,
Onde = napoles;
```

Para perguntar se alguém se viu a si próprio em Pitões das Júnias, embora isso não tenha acontecido, (tanto quanto sabemos não há espelhos em Pitões das Júnias) , faremos:

```
?- viu(Viu,Viu,pitoesDasJunias).
false.
```

Também poderíamos ter perguntado da seguinte maneira:

```
?- viu(Viu,Visto,pitoesDasJunias), Viu = Visto.
false.
```

E se perguntarmos por quem viu o Pedro e em que aldeia, perguntaremos:

```
?- viu(Viu,pedro,Aldeia), aldeia(Aldeia).
Viu = joao,
Aldeia = pitoesDasJunias ;
false.
```

Notem que não basta chamar a uma variável de Aldeia para o Prolog perceber que essa variável tem de ser unificada com uma "aldeia": Vila Verde não é uma aldeia.

```
?- viu(Viu,pedro,Aldeia).
Viu = joao,
Aldeia = pitoesDasJunias ;
Viu = joao,
Aldeia = vilaVerde ;
false.
```

E se perguntarmos se alguém viu a Diana que não seja o João e em que lugar, podemos usar o operador de não unificação.

```
?- viu(Viu,diana,Onde), Viu \= joao.
Viu = gasosa,
Onde = luanda ;
false.

?- Viu \= joao, viu(Viu,diana,Onde).
false.
```

O que aconteceu por termos invertido a ordem dos elementos da conjunção? A "query" falhou porque uma variável sem valor unifica sempre com uma constante falhando "Viu \= joao" e falhando consequentemente a "query" conjunção.

7. Variáveis anónimas

Podemos estar interessados em saber se o João viu alguém em Pitões das Júnias sem queremos saber quem é que ele viu. Para isso precisamos de ter variáveis anónimas, que em Prolog são indicadas por _ ("underscore").

Se utilizarmos uma variável anónima numa "query", essa variável será única e nenhuma instanciação dessa variável será mostrada.

```
?- viu(joao,_,pitoesDasJunias).
true ;
false.
```

Mas o facto de usarmos variáveis anónimas não impede o retrocesso e a obtenção de mais respostas, que seriam desnecessárias. Mais à frente iremos perceber como impedir ou controlar o retrocesso. Se quisermos saber se o João viu alguém em algum lugar, faremos

```
?- viu(joao,_,_).
true .
```

Ou se alguém viu o Gasosa em algum lugar:

```
?- viu(_,gasosa,_).
false.
```

Cada ocorrência de _ corresponde a uma variável diferente — mesmo dentro da mesma cláusula duas variáveis anónimas não serão variáveis iguais.
Se quisermos saber se o Gasosa viu alguém numa aldeia, a seguinte "query" não será correcta porque não é verdade que isso tenha acontecido—ele só viu gente em Luanda que é uma cidade.

```
?- viu(gasosa,_,_),aldeia(_).
true.
```

Como cada variável anónima na pergunta corresponde a uma variável distinta, é equivalente a fazermos a pergunta seguinte com três variáveis não anónimas, com a única diferença que, neste caso, os valores das variáveis serão mostrados, o que não acontece quando se usam variáveis anónimas.

```
?- viu(gasosa,Visto,Onde),aldeia(Aldeia).
Visto = diana,
```

```
Onde = luanda,
Aldeia = pitoesDasJunias.
```

6 Igualdade entre expressões (sem avaliação aritmética)

Para testar a igualdade de termos utilizamos o operador ==. O operador \== denota a desigualdade de termos. Reparem que não há qualquer unificação na igualdade de termos. Duas variáveis para serem iguais têm de ser exactamente a mesma variável.

```
?- X == 2.
false.

?- X == Y.
false.

?- X = 2, Y = X, Y == X.
X = Y, Y = 2.

?- aldeia(X) == aldeia(pitoesDasJunias).
false.

?- X=pitoesDasJunias, aldeia(X) == aldeia(pitoesDasJunias).
X = pitoesDasJunias.

?- X \== 2.
true.

?- X \== Z.
true.

?- X = 4, Y = 6, Y \== X.
X = 4,
Y = 6.

?- X = 4, Y = X, Y \== X.
false.
```

Notem que tanto a unificação como a igualdade não forçam a avaliação de expressões aritméticas.

```
?- X == 2 + 4.
false.
```

```
?- X = 2 + 4.
X = 2+4.

?- X + 2 + 5 = 2 + 2 + Y.
X = 2,
Y = 5.

?- 2 + 6 \== 8.
true.
```

7 Igualdade/Diferença e comparação entre avaliação de expressões aritméticas

Para compararmos os valores resultantes da avaliação de expressões aritméticas, teremos de usar o operador =:= ou a sua negação =\=. Para compararmos expressões aritméticas é preciso que todos os operandos estejam instanciados.

```
?- 3 + 4 - 2 =:= 2 + 8 - 5.
true.

?- X = 2, 4 - X =:= 2.
X = 2.

?- X + 2 + 5 =:= 2 + 2 + Y.
ERROR: =:=/2: Arguments are not sufficiently instantiated

?- X = 2, X - 12 =\= 6 + 2.
X = 2.

?- X = Y, X = 2, 2 + 6 >= X + Y.
X = Y, Y = 2.

?- X = Y, X = 2, 2 + 6 =< X + Y.
false.
```

8 Atribuir a uma variável ainda não instanciada o resultado de avaliar uma expressão aritmética (IS)

Só faz sentido usar o operador *is* (um avaliador aritmético built-in) se tivermos uma variável do lado esquerdo e uma expressão aritmética totalmente instanciada no lado direito. "X is E" calcula primeiro a expressão aritmética E e unifica o resultado com X.

```
?- X is 2 + 3 + 4.
X = 9.

?- Z is 2 + 4 / 5 * 6.
Z = 6.8000000000000001.

?- X + X is 2 + 2.
false.
```

No entanto, se a variável no lado esquerdo do operador is estiver instanciada, compara-se com o valor que resulta da avaliação da expressão à direita.

```
?- X = 2, X is 3 - 4 + 3.
X = 2.
```

Na verdade, uma variável já com um valor numérico é equivalente a utilizar directamente o número à esquerda do operador i*s*.

```
3 is 5 - 2.
true.
```

Se quisermos incrementar o valor de uma variável teremos de arranjar outra variável. Isso porque uma variável nunca muda de valor, excepto quando o Prolog faz retrocesso.

```
?- Y = 2, Y is Y + 1.
false.

?- Y = 2, X is Y + 1.
Y = 2,
X = 3.
```

9 Escrever e Ler

Podemos utilizar o *write/1* para escrever no ecrã e o *nl* para mudar de linha.

```
?- write(2).
2
true.

?- X = 2,write('X toma o valor de '),write(X).
X toma o valor de 2
X = 2.

?- write(23),nl,write(34).
23
34
true.
```

Como podem confirmar, as variáveis internamente têm um identificador diferente da sua versão "pretty".

```
?- write(X).
_G251
true.
```

Como podem confirmar, as variáveis anónimas (_) são sempre únicas.

```
?- write(_),nl, write(_).
_G251
_G253
true.
```

Se tiverem um nome para além do underscore, essas variáveis já não são anónimas.

```
?- write(_M),nl,write(_),nl,write(_M).
_G251
_G253
_G251
true.
```

E em dois momentos diferentes (duas "queries") a mesma variável X tem exactamente o mesmo identificador, mas perde o seu valor porque o "tempo de vida" ou "scope" da variável dura só uma "query".

```
?- write(X), nl, X = 20, write(X).
_G251
20
X = 20.

?- write(X).
_G251
true.
```

Para termos variáveis globais, temos de as guardar em factos e ir buscar os respectivos valores. Mais à frente explicaremos como poderemos mudar os valores dessas variáveis globais.

O write pode ser muito útil para compreender como o Prolog funciona. Basta-nos tentar saber quais os homens que o João viu.

```
?- viu(joao,H,L),write('Tentando '),write(H),nl,homem(H).
Tentando diana
Tentando pedro
H = pedro,
L = pitoesDasJunias ;
Tentando ausenda
Tentando pedro
H = pedro,
L = vilaVerde.
```

Uma "query" conjunção é executada da esquerda para a direita e sempre que um dos componentes da conjunção falha, dá-se um retrocesso e o Prolog regressa ao elemento da conjunção imediatamente anterior. A primeira pessoa que o João viu é a Diana mas como não é homem, o Prolog falha e tenta resatisfazer o predicado nl, que só tem uma solução, sucede uma única vez com o efeito lateral de fazer o cursor mudar de linha. Então continua a retroceder, da direita para a esquerda, mas o predicado write/1 nunca sucede e chega de novo ao predicado viu(joao,H,L). Reparem que já tínhamos valores para as variáveis H e L, mas como o Prolog falhou a seguir, ele tenta resatisfazer este predicado que tem uma nova solução para H=pedro e L=pitoesDasJunias. As variáveis perderam os valores apenas porque o Prolog falhou e tentou mais soluções. Como Pedro é homem então a "query" sucede. Ao pedirmos mais soluções é como se forçássemos a falha e o Prolog começa a retroceder, forçado por nós, pelo símbolo ;. Não

há novas soluções para os predicados nl e write/1 e o viu(joao,H,L) dá uma nova solução, etc. etc.

O read/1 serve para ler valores do teclado.

```
?- write('Escreva qualquer número: '), read(X), Y is X + 1.
Escreva qualquer número: 234.
X = 234,
Y = 235.
```

Mas, se o que for lido não for número teremos um erro.

```
?- write('Escreva qualquer número: '),read(X),Y is X + 1.
Escreva qualquer número: erg.
ERROR: is/2: Arithmetic: `erg/0' is not a function
```

Podemos verificar se o valor lido é um número:

```
?- write('Escreva qualquer número: '),read(X), number(X),Y
is X + 1.
Escreva qualquer número: dada.
false.
```

Mas, se quisermos fazer um ciclo que peça o número até que o valor lido seja um número, usaremos o predicado *repeat*, que tem a qualidade de voltar a suceder sempre, como se desse um número infinito de respostas. É como se fosse uma parede que força a execução do Prolog para a direita, quando este a encontra ao retroceder (ele retrocede quando falha numa "query" ou se pedimos mais soluções usando o ;)

```
?- repeat.
true ;
true ;
true ;
true ;
true ;
…nunca mais pararia de dar soluções se colocarmos um ;
```

Vejam como podemos fazer a entrada de dados que repete a mensagem até que o valor lido do teclado satisfaça os critérios, neste caso, ser um número.

```
?- repeat,write('Escreva um número: '), read(X), number(X),
Y is X + 1.
Escreva um número: er.
Escreva um número: eeee.
Escreva um número: 345.
X = 345,
Y = 346 .
```

Notem que se um dos "goals" falhar, o Prolog retrocede e tenta resatisfazer o "goal" anterior. Neste caso, é o "goal" number(er) que falha e força o retrocesso. O read(X) não volta a suceder porque o predicado read/1 nunca tem mais do que uma soluçao, forçando o retrocesso para o "goal" anterior: write('Escreva qualquer número: '), mas o write/1 nunca volta a suceder, forçando o retrocesso no repeat. O predicado repeat volta a suceder sempre e de novo o Prolog avança para a conjunção de "goals" a partir do repeat da esquerda para a direita.

10 Negação por falha (\+) e a disjunção (;)

O predicado \+/1 fornece a negação por falha. A pergunta **?- \+ Obj** sucede se não conseguir provar (satisfazer) Obj. Se uma prova for encontrada para Obj então \+ Obj falha. Notem que se Obj tiver variáveis sem valores atribuídos, as variáveis não terão valores mesmo que \+ Obj suceda.

Se quisermos perguntar primeiro em que lugar que não é uma aldeia é que o João viu alguém, faremos

```
?- viu(joao,_,M),\+ aldeia(M).
M = napoles ;
M = vilaVerde.
```

Ou se quisermos perguntar por um homem que não tenha sido visto por ninguém, faremos:

```
?- homem(H),\+ viu(_,H,_).
H = gasosa.
```

Note que não precisamos de colocar variáveis não anónimas nem no primeiro nem no último argumentos do predicado viu/3. Mas, se tivéssemos colocado, o resultado seria o mesmo

```
?- homem(H),\+ viu(Q,H,L).
H = gasosa.
```

porque para o objectivo viu(Q,H,L) falhar, quando H se liga a uma constante, é preciso que não hajam instanciações de Q e de L que o satisfaçam. E como dissemos atrás, nunca \+ viu(Q,H,L), com H ligado a uma constante, quando é satisfeita pode devolver valores para Q e L, precisamente porque Q e L não estão instanciados a nenhum valor quando é executada o predicado argumento de \+. Assim, também a conjunção homem(H), \+ viu(Q,H,L) não poderia nunca devolver valores em Q e L, sendo equivalente a usarmos variáveis anónimas.

Vamos executar de novo, agora traçando a execução. Começa pelo primeiro homem: João que falha porque foi visto, depois passa ao Pedro que falha também, a seguir tenta Gonzaga que não sucede de novo e só depois Gasosa.

```
?- homem(H),write('Tentando '), write(H),nl,\+ viu(Q,H,L).
Tentando joao
Tentando pedro
Tentando gonzaga
Tentando gasosa
H = gasosa.
```

Essa ordem depende dos factos homem/1 que estão por essa ordem na memória de trabalho, como pode comprovar fazendo listing(homem/1).

```
?- listing(homem/1).
homem(joao).
homem(pedro).
homem(gonzaga).
homem(gasosa).

true.
```

Mas se quisermos saber quem é homem ou mulher, teremos de usar o operador de disjunção, o ponto e vírgula (;). Na verdade já o usámos para pedir ao Prolog, durante uma pergunta, para nos dar mais respostas. No fundo, a disjunção de respostas.

```
?- homem(X) ; mulher(X).
X = joao ;
X = pedro ;
X = gonzaga ;
X = gasosa ;
X = diana ;
X = ausenda.
```

Primeiro o Prolog tenta dar todas as respostas possíveis de homem(X) e só depois retrocede para obter as soluções de mulher(X).

Se quiseremos saber quem é que não foi visto, faremos:

```
?-  (homem(H);mulher(H)),write('Tentando  '),  write(H),  nl,
\+ viu(Q,H,L).
Tentando joao
Tentando pedro
Tentando gonzaga
Tentando gasosa
H = gasosa ;
Tentando diana
Tentando ausenda
H = ausenda.
```

No entanto, se invertermos a ordem dos operandos na disjunção teremos as mesmas respostas, mas começaremos por procurar primeiro entre as mulheres e só depois entre os homens.

```
?- (mulher(H) ; homem(H)),write('Tentando '), write(H), nl,
\+ viu(Q,H,L).
Tentando diana
Tentando ausenda
H = ausenda ;
Tentando joao
Tentando pedro
Tentando gonzaga
Tentando gasosa
H = gasosa.
```

Vamos agora traduzir algumas frases em perguntas e executá-las.

É verdade que nenhum homem viu alguém?

```
?- \+ (homem(X), viu(X,_,_)).
false.
```

É verdade que nenhum homem viu uma mulher?

```
?- \+ (homem(X), viu(X,M,_), mulher(M)).
false.
```

Quem é o homem que viu todas as mulheres em algum sítio?

```
?- homem(X),\+ (mulher(M), \+ viu(X,M,_))).
false.
```

É verdade que todos os homens viram alguém?

```
?- \+ (homem(X), \+ viu(X,_,_)).
false.
```

É verdade que todos os homem viram uma mulher?

```
?- \+ (homem(X), \+ (mulher(M),viu(X,M,_)) ).
false.
```

Todos os homens viram todas as mulheres em algum sítio?

```
?- \+ (homem(X), mulher(M), \+ viu(X,M,_)).
false.
```