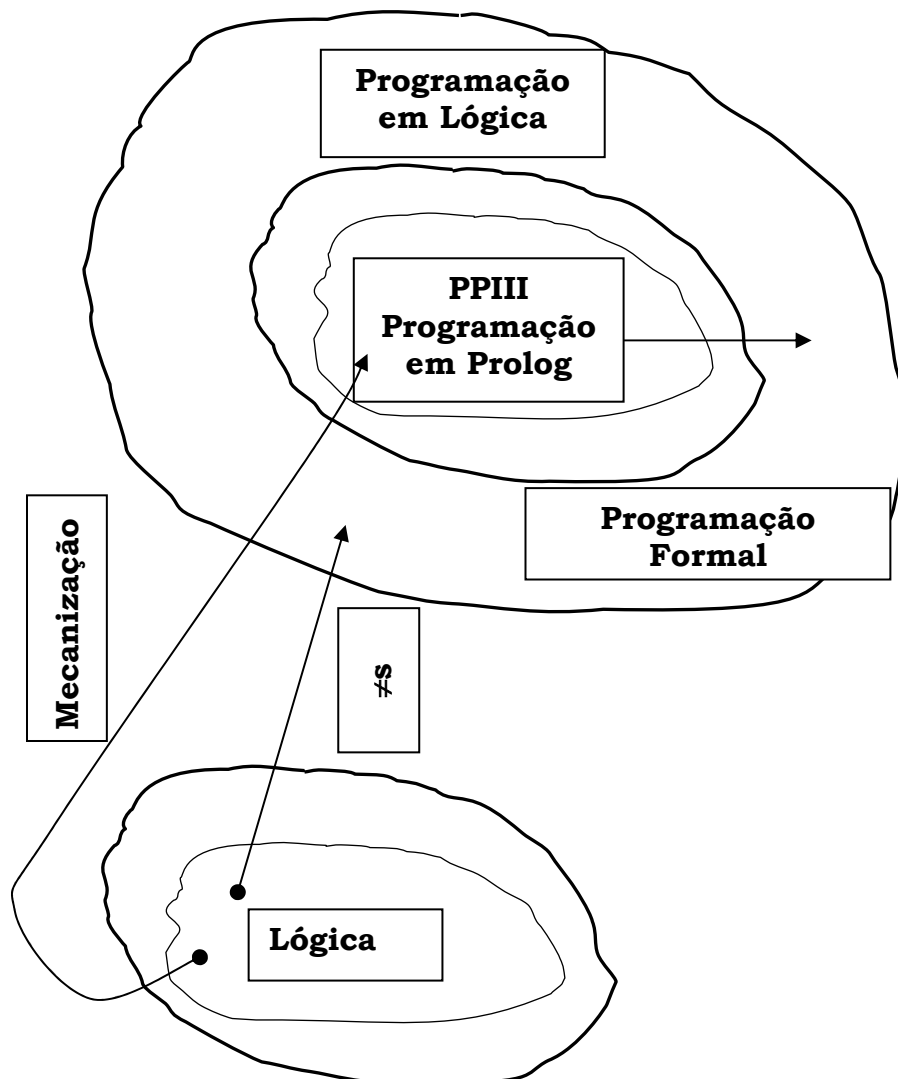


CONTEXTO DA DISCIPLINA

É importante perceber-se que nesta disciplina o objectivo é explicar o que se deve entender por **Programação numa lógica formal**, ou seja, apresentar um paradigma alternativo de programação, e não tecer considerações matemáticas e formais sobre o que é **Programação formal em lógica**. Assim, as nossas considerações nesta disciplina podem ser vistas como um subconjunto da **Programação formal em lógica**, tratando-se de facto de introduzir o paradigma de programação que usa um subconjunto mecanizável da lógica de 1ª ordem, as cláusulas de Horn, e de uma máquina de interpretação das mesmas, o **Prolog** (cf. Figura seguinte).



EXPRESSÃO DE CONHECIMENTO

A lógica, enquanto sistema matemático formal, é uma “ferramenta” precisa na qual é possível exprimir proposições matemáticas.

Os lógicos pretendem ter linguagens com as quais possam expressar diversas regras matemáticas ou asserções (axiomas) que caracterizam determinados domínios de conhecimento, e a partir das quais, usando um sistema matemático de prova, novo conhecimento sobre esses domínios possa ser inferido, gerado e/ou provado.

Objectivos :

Expressão de $\left\{ \begin{array}{l} \textit{Conhecimento} \\ \textit{Objectivos} \\ \textit{Hipóteses} \end{array} \right.$

- **Dedução de consequências** a partir de um conjunto de premissas;
- **Estudo de veracidade** (ou falsidade) de afirmações com base na veracidade (ou falsidade) de outras;
- **Verificação da coerência** de um dado raciocínio;
- **Verificação da validade** de um dado argumento.

Porém:

Conhecimento matemático	\neq	Conhecimento comum
------------------------------------	--------	-------------------------------

Torna-se pois necessário conceber um cálculo que torne a expressão do conhecimento e a noção de consequência precisas e de fácil manipulação matemática.

Exemplo: *Lógica de 1ª ordem*

Termos, proposições, formulas, frases e regras.

Constantes, funções, variáveis, predicados, quantificadores e conectivas lógicas.

LN: Todos gostam de aprender.

$(\forall X) (\text{gosta-aprender}(X))$

LN: Todos programam em Prolog.

$(\forall X) (\text{programa-em}(\text{prolog}, X))$

LN: Todos gostam da sua mãe.

$(\forall X) (\text{gosta}(x), \text{mãe}(X))$

LN: Todo o livro tem um autor.

$(\forall X) (\exists Y \text{ autor } (Y, X) \leftarrow \text{livro}(X))$

LN: Não é verdade que todos que fazem férias as façam na praia ou no campo.

$\neg (\forall X) (\text{praia}(X) \vee \text{campo}(X) \leftarrow \text{faz-férias}(X))$

INTRODUÇÃO AO PARADIGMA DA PROGRAMAÇÃO EM LÓGICA

Características principais

- **Uso de lógica para representar o conhecimento** sobre dada área ou problema; *mas que lógica?*
- **Uso de inferência** no processamento desse conhecimento, i.e., dedução de conclusões a partir de tal conhecimento;
- **Noção de consequência lógica**, i.e., a possibilidade de provar por dedução, que certo conhecimento é inferível do existente;
- Programação em lógica – **bases**:
 - Conhecimento + **Raciocínio mecanizado**
 - Lógica + **Controlo**
- Programação em lógica **é declarativa**.

Expressa-se o conhecimento sobre os problemas e não sobre as suas possíveis soluções.

Nota importante:

Programação em lógica	\supset	Programação em Prolog
	\neq	

Problema:

A lógica de 1ª ordem é semi-decidível, ou seja, nem sempre é possível encontrar um procedimento que, em tempo finito e dado certo conhecimento, permita determinar se um dado argumento é correcto ou não, ou se um dado consequente é inferível de tal conhecimento.

Porém:**Matemáticos descobriram que:**

Existem subconjuntos da lógica de 1ª ordem que possibilitam a implementação de procedimentos de decisão efectivos, e que possuem adicionalmente o poder de expressar qualquer sequência lógica expressável no sistema completo.

Robinson - 65

*** dedução automática**

Green - 69

*** sistema de resolução em Lisp**

Kowalsky - 70

Assim:

Em 1972 um grupo de investigadores da Universidade de Marselha desenvolveu um sistema de resolução para um subconjunto da lógica de 1ª ordem designado por

CLÁUSULAS DE HORN

(cf. Kowalsky, Colmerauer)

FORMA CLAUSAL; CLÁUSULAS DE HORN

- Uma forma normal da lógica de 1ª ordem é:

(quantificadores) (disjunção de literais)

- Programas em lógica são simplesmente sequências de cláusulas, apresentadas, em geral, sem quantificadores, e substituindo sequências $\vee \neg$ por \leftarrow (se).

gosta(ana, X) \leftarrow gosta(X, desporto)
gosta(luis, Y)

- Reescrever frases lógicas em forma clausal preserva modelos (semântica) e consequências lógicas.

A forma clausal tem uma grande vantagem pois permite a simplificação da **interpretação procedimental da lógica**

Assim:

PROLOG *foi o primeiro interpretador procedimental da lógica das cláusulas de Horn*

$P \text{ se } P_1 \wedge P_2 \wedge \dots \wedge P_r$

P \leftarrow P₁, P₂, ..., P_r

Kowalsky mostrou que um dado axioma escrito como

P \leftarrow P₁, P₂, ..., P_r

possui duas leituras (ou interpretações, ou semânticas),

Lógica: *P é verdadeiro se é verdadeiro $P_1 \underline{\text{e}} P_2 \underline{\text{e}} \dots P_r$*

Procedim.: *Para executar (ou resolver) P, executar (ou resolver) $P_1 \underline{\text{e}} P_r$*

PROGRAMAÇÃO EM LÓGICA

Programa	=	“conjunto” de axiomas , ou regras, definindo relações entre objectos;
Resultado	=	uma consequência dos axiomas;
Computação	=	dedução da consequência a partir dos axiomas;
Semântica do programa	=	todas as consequências dedutíveis dos axiomas.

Note-se a importância das palavras em negrito na caracterização do paradigma da Programação em Lógica, e na diferenciação deste de outros paradigmas de programação.

Não se caracteriza de momento quais as formas de denotação dos objectos de interesse para a expressão do conhecimento, assumindo-se assim que tais entidades mantêm a sua representação usual em lógica de 1ª ordem.

Vamos a partir de agora estudar estas características da Programação em Lógica tal como se encontram representadas na linguagem PROLOG. Todas as considerações teóricas serão realizadas perante exemplos concretos apresentados durante o estudo da linguagem que serve de apoio ao estudo do paradigma.

PROGRAMAÇÃO EM LÓGICA EM PROLOG

A LINGUAGEM PROLOG

- **Programa PROLOG** = “conjunto” de cláusulas de Horn;

Cláusula de Horn

<cabeçalho> :- <corpo>.

gosta (ana, X) :- gosta(X, prolog). $\forall X$

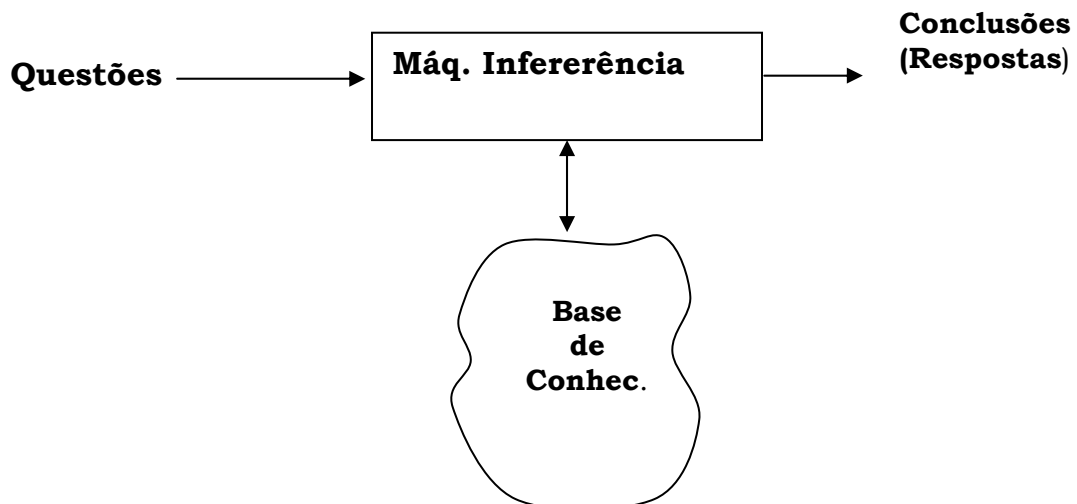
tem-ferias(z) :- tem-tempo(Z), tem-\$ (Z). $\forall Z$

avó(X,Z) :- mãe(X,Y), progenitor(Y,Z) $\exists Y$

- Programar em PROLOG
 - 1.- Declarar factos sobre objectos
 - 2.- Definir regras sobre relações entre objectos
 - 3.- Questionar, inferir sobre objectos e suas relações

1 + 2 = **Representação do Conhecimento**

3 = **Processo de Inferência**



CONSTRUÇÕES BÁSICAS

1.- FACTOS

Expressões que afirmam propriedades ou atributos de objectos, ou estabelecem uma relação entre objectos.

<code>homem(rui).</code>	<i>“rui é um homem”</i>
<code>mulher(ana).</code>	<i>“ana é uma mulher”</i>

- Relações entre objectos designam-se na gíria Prolog por Predicados.
- Os predicados anteriores são unários, sendo os seus argumentos constantes.

<code>media(braga, 15).</code>	}	Predicados binários
<code>media(porto, 14).</code>		
<code>media(beja, 18).</code>		

<code>ensina(joão, inglês).</code>	}	Predicados n-ários
<code>ama(luís, ana).</code>		
<code>ama(luís, sara).</code>		
<code>ama(sara, rui)</code>		
<code>somar(0, 0, 0).</code>		
<code>somar(0, 1, 1).</code>		
<code>somar(0, 2, 2).</code>		
<code>pais(joão, maria, ana).</code>		
<code>pais(joão, maria, rita).</code>		
<code>pais(rui, sara, luís).</code>		

Obs: *Notar a relevância semântica da ordem dos argumentos!*

FACTOS UNIVERSAIS

- Variáveis, em factos, representam universalidade, e permitem sumarizar um número virtualmente infinito de factos.

Exº: **Todos gostam de música**

`gosta(X, musica).` $\forall X$

↑ **variável**; iniciada por maiúsculas

Exº: **Qualquer número adicionado a 0 é ele próprio.**

`soma(0, W, W).` $\forall W$

Exº: **Todos acreditam na Natureza**

`acredita(Z, natureza).` $\forall Z$

Nota: Um *nome* refere um indivíduo particular. A interpretação do *nome* e a consistência da interpretação fica a cargo do programador. Para o Prolog são *aceitáveis e verdadeiros* os factos transmitidos, mesmo que na realidade não o sejam.

Exºs:

`mineral(vidro).`

`primo(15).`

`presidente(frança, pedro).`

`detesta(X, musica).`

`adora(Z, matematica).`

Nota:

- Uma *variável lógica* representa um indivíduo não especificado;
- Por um processo de *substituição*, uma variável pode ser associada a uma entidade; o resultado de tal *associação* (ou associações) variável-entidade designa-se por *instância*;

Exº:

`gosta(joão, ana)` é uma possível *instância* de:



`gosta(W, ana)`

- Definir-se-à mais tarde de modo rigoroso o que se entende por:

- **Substituição**
- **Instância**
- **Instanciação**

Nota:

Um **facto Prolog** corresponde a uma *cláusula de Horn sem corpo*, daí designar-se por **Cláusula Unitária**

2.- “QUERIES” OU INTERROGAÇÕES.

- Um conjunto de factos, só por si, constitui uma base de conhecimento sobre um dado domínio.
- É agora necessário introduzir as construções que permitam, com base em, pelo menos, tais factos, extrair informação da base de conhecimento.
- Esta extracção de informação, em casos simples apenas uma interrogação sobre se dada afirmação ou conclusão é inferível ou dedutível do conhecimento, faz-se através de um Query, expresso em Prolog como uma **cláusula negativa**, i.e., *que apenas possui corpo*, antecedida do símbolo ?-

Exº:

Dado o conhecimento disponível e apenas este !!!

- **É (possível provar ser) 15 múltiplo de 3?**

?- multiplo(15,3) .

- **É o João pai do Nuno?**

?- pai(joao,nuno) .

- **O número 7 é um número par?**

?- par(7).

Nota:

- *Responder a uma interrogação corresponde a determinar se esta é uma **consequência lógica** do programa.;*
- *Consequências lógicas obtêm-se pela aplicação de regras de dedução que introduziremos gradualmente.*

1ª REGRA DE DEDUÇÃO: IDENTIDADE.

A partir de P pode deduzir-se P.

Assim:

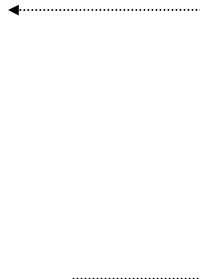
Um “Query” é sempre uma consequência lógica de um facto idêntico.

Operacionalmente:

Procurar um facto no programa que implique o “Query” formulado (para queries simples).

Exº:

```
gosta(ana, rui).  
gosta(ana, pedro).  
gosta(rui, ana)  
gosta(luis, rita).  
gosta(to, ana).
```



?- gosta (rui, ana). ↵

yes

?- gosta (joao, rita) . ↵

no



**falso ou
inferência
impossível**

- Queries mais interessantes são os que envolvem variáveis, pois correspondem a questões (*goals* ou *objectivos*) mais gerais.

Exº:

- **De quem gosta a ana?**

$\exists X ?$

?- gosta(ana, X).

- **Que livros escreveu Eça?**

?- autor(eça, L).

- **O Rui é pai de quem?**

?- pai(rui, F).

Nota:

Do ponto de vista lógico, variáveis em “queries” são existencialmente quantificadas, pelo que um “query” do tipo

?- le(Y, hemingway).

Deve ler-se

“Existirá algum Y tal que Y lê hemingway?”

A resposta poderá ser:

no

ou cada um dos Y que satisfazem o *Goal*,

{X=josé} ; {X=rui} ; {X=tó};

Exemplo Informal:

BASE DE CONHECIMENTO	
autor(camoes, lusiadas).	
autor(eca, mandarim).	obra = mandarim
autor(eca, tragedia).	obra = tragedia
autor(heminGway, fiesta).	
autor(heminGway, velho).	X=heminGway
autor(heminGway, sinos).	
autor(uris, exodus).	
autor (gregor, ponte).	
autor (gorki, mac).	

Query-Goal \Rightarrow Resultado

?- autor (eca, fiesta). no = falso
no

?- autor (virgilio, odisseia). no = ???
no

?- autor (camoes, lusíadas). cf. $P \Rightarrow P$
yes

?- autor (X, velho).
X = heminGway ;⌊
no

?- autor (eca, obra).
obra = mandarim ; ⌊
obra = tragedia ; ⌊
no

Nota: *variáveis são inicialmente não instanciadas. Cada instanciação corresponde a uma solução que é apresentada. ; corresponde a pedir outra instanciação*

ALGORITMO INFORMAL DE PROCURA:

- As variáveis do query são inicialmente **não inicializadas**.
- A procura na base de conhecimento permite encontrar uma instânciação que corresponda ao objectivo (\Rightarrow este é uma consequência da base). Então, tal instânciação (valores das variáveis) é apresentada.
- Se o utilizador pretende novas soluções (i.e. novas instânciações) introduz ; senão <RET>.
- O *facto* da base de conhecimento que garantiu a última instânciação é *marcado* (para evitar recursividade infinita).
- A procura continua (;) com as variáveis de novo não inicializadas.
- O sistema de inferência do Prolog tenta, nestes casos, resatisfazer o objectivo, procurando novas instânciações, indicando cada uma das encontradas, e prosseguindo recursivamente.
- A procura termina quando o “Goal” não puder ser mais satisfeito (inexistência de mais instânciações) por exaustão da base, ou por decisão do utilizador (cf. <RET>).

Nota: Veremos posteriormente o tratamento formal destas questões!

NOTA FORMAL

- Numa BC formada apenas por FACTOS, o significado de cada facto é que o mesmo se verifica (i.e. é verdadeiro) no domínio representado.
- Assim, BC tem por significado a conjunção lógica de tais factos, i.e.,

$$\mathbf{BC} = \mathbf{F}_1 \wedge \mathbf{F}_2 \wedge \dots \wedge \mathbf{F}_n$$

- Uma BC deve ser logicamente consistente, ou seja, não deverá possuir factos que entrem em contradição com outros, em função do significado que lhes é atribuído. Assim, numa BC consistente os factos são, quanto ao seu significado independentes.
- Admitindo uma BC consistente, o valor lógico de um query é o que resulta de este ser ou não uma consequência lógica da BC, ou seja, de se poder afirmar que

$$\mathbf{BC} \Rightarrow \mathbf{Q}$$

- Em termos lógicos $\mathbf{BC} \Rightarrow \mathbf{Q}$ se

$$\neg \mathbf{BC} \vee \mathbf{Q} = \text{true} \quad \text{ou ainda se}$$

$$\mathbf{BC} \wedge \neg \mathbf{Q} = \text{false} \quad \leftarrow !!$$

- Demonstrar que

$$\mathbf{BC} \wedge \neg \mathbf{Q} = \text{false} \quad \text{é pois equivalente a demonstrar que}$$
$$\mathbf{BC} \Rightarrow \mathbf{Q}$$

- Demonstrar que $\mathbf{BC} \wedge \neg \mathbf{Q} = \text{false}$ é, por outro lado, demonstrar que é inconsistente a negação de Q com a veracidade de BC, ou seja, que Q é de facto uma consequência de BC.

- Se BC tem apenas factos, poder-se-à afirmar que

$$\mathbf{BC} \Rightarrow \mathbf{Q} \text{ se } \exists \mathbf{F_i. F_i} \Rightarrow \mathbf{Q}$$

ou seja se $\mathbf{F_i} \wedge \neg \mathbf{Q} = \underline{\text{false}}$

- Note-se finalmente que, de momento,

$$\mathbf{F_i} \Rightarrow \mathbf{Q}$$

se se verificar um dos seguintes princípios da dedução

Identidade: $\mathbf{F_i} = \mathbf{Q}$

Generalização: $\mathbf{F_i} = \mathbf{Q[X_1/T_1, ..., X_n/T_n]}$

*($\mathbf{F_i}$ é uma instância particular de \mathbf{Q} via dada substituição
então $\mathbf{F_i} \Rightarrow \mathbf{Q}$ por generalização)*

Exº:

pratica(ana, ténis) $\Rightarrow \exists \mathbf{X. pratica(X, ténis)} = \mathbf{TRUE}$
(em particular para $\mathbf{X=ana}$)

DEFINIÇÕES RIGOROSAS

DEF1 : TERMO

Termo := Termo-Simples | Termo-Composto

Termo-Simples := Constante | Variável

Constante := Literal | Valor

Termo-composto := Predicado Termo*
(Functor)

Ex^os:

abc	30	X
$f(a)$		soma(1, 1, 2)
lista(a, lista (b, nil))		pai(X, rui)
membro(X, $\underbrace{[X \mid L]}$)		

DEF2: SUBSTITUIÇÃO

Uma **substituição** é um conjunto finito de pares $X_i = T_i$ onde X_i é uma variável e T_i um termo, e $\forall i \neq j. X_i \neq X_j$, e X_i não ocorre em $T_j \forall i, j$.

Uma substituição denota-se em geral por θ .

O resultado de aplicar uma substituição θ a um termo A denota-se $A\theta$, e é o termo que se obtém substituindo cada ocorrência de X por T , para cada par $X = T$ de θ .

Ex^o:

$$\theta = \{ X = \text{prolog} \}$$
$$\text{estuda}(\text{rui}, X)\theta \rightarrow \text{estuda}(\text{rui}, \text{prolog})$$

DEF3 : INSTÂNCIA

Um termo A diz-se *uma instância de* B se existe uma certa substituição θ tal que $A = B\theta$

Ex^o:

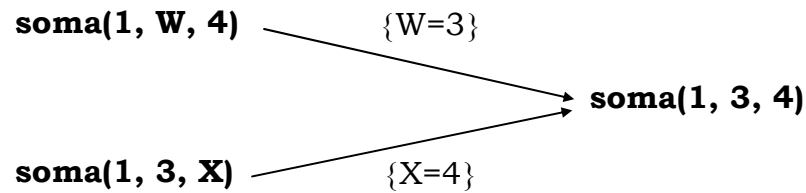
O “query” $?- \text{pai}(\text{rui}, X)$ é uma generalização do facto $\text{pai}(\text{rui}, \text{ana})$. Por outro lado, o facto $\text{mae}(\text{ana}, \text{pedro})$ é uma instância de $\text{mae}(\text{ana}, Z)$ para a substituição $\theta = \{Z = \text{pedro}\}$

DEF4: INSTÂNCIA COMUM

C é uma *instância comum* de A e B se é simultaneamente uma instância de A e de B, ou seja, se é possível encontrar substituições θ_1 e θ_2 tais que:

$$C = A\theta_1 \text{ e } C = B\theta_2.$$

Exº:



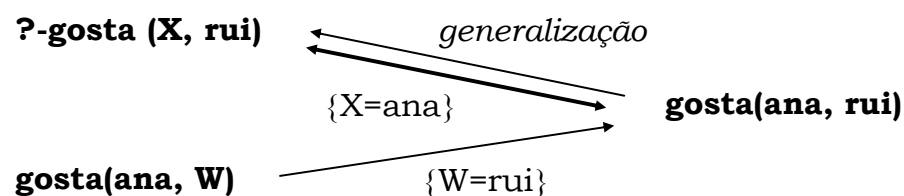
Nota:

Responder a um “Query” existencial usando um facto universal via uma instância comum envolve 2 passos dedutivos.

1) A instância é deduzida do facto por *instanciação* $F \Rightarrow I$
inst.

2) O “Query” é deduzido da instância por *generalização* (ver exemplo atrás!) $I \Rightarrow Q$
gen.

Exº:



2.1.- “QUERIES” CONJUNTIVOS.

“Queries” conjuntivos correspondem a uma conjunção lógica de “Goals” a serem satisfeitos, assumindo a forma geral.

$$?- Q_1, Q_2, \dots, Q_n.$$

Os mais simples que podem construir-se designam-se “queries” **conjuntivos de base** (“ground”) por envolverem apenas constantes.

$$?- \text{pai}(\text{rui}, \text{ana}), \text{pai}(\text{rui}, \text{pedro}).$$

“É o rui pai da ana e do pedro?”


“Queries” conjuntivos com variáveis são bastante mais interessantes, em particular se existirem variáveis partilhadas pelos “sub-queries”.

Exº:

- Existe alguém que estude inglês e pratique futebol?

$$?- \text{estuda}(X, \text{ingles}), \text{pratica}(X, \text{futebol}).$$

- Alguém é pai de alguém que pratica sumo?

$$?- \text{pai}(X, \underline{Y}), \text{pratica}(\underline{Y}, \text{sumo}).$$


Variáveis partilhadas são usadas como forma de restringir um “query” por restrição da gama de valores associáveis

Nota: Considere o “query”:

$$?- \text{pai}(\text{rui}, X), \text{pai}(X, Y).$$

Interp1 (sobre a variável X)

Existe algum filho de rui que seja também pai?

Interp2 (sobre a variável Y):

Existe alguém cujo pai seja filho do rui?

NOTA FORMAL

Um “Query” conjuntivo Q_1, Q_2, \dots, Q_n ? é uma consequência lógica duma BC, i.e.,

$$BC \Rightarrow Q_1, Q_2, \dots, Q_n$$

se, sendo as variáveis partilhadas instanciadas com os mesmos valores em todos os “Sub-Queries”, todos os “Sub-Queries” forem consequências lógicas da BC, ou seja, se

$$(BC \Rightarrow Q_1) \wedge (BC \Rightarrow Q_2) \wedge \dots (BC \Rightarrow Q_n)$$

INTERPRETAÇÃO OPERACIONAL

Operacionalmente, a resolução de um query conjuntivo Q_1, Q_2, \dots, Q_n ? usando uma base BC envolve encontrar uma substituição θ tal que $Q_1\theta \wedge Q_2\theta \wedge \dots \wedge Q_n\theta$ são instâncias básicas dos factos em BC.

A aplicação da mesma substituição θ a todos os “sub-queries” garante que as instanciações das variáveis são comuns a todo o “query”, cf. determinado pelas *Regras de Scope*.

Como é implementada em Prolog esta interpretação operacional?

Exemplo:

Considere-se a BC factual seguinte

fala(joão, inglês).

fala(joão, francês).

fala(ana, inglês).

fala(rui, francês).

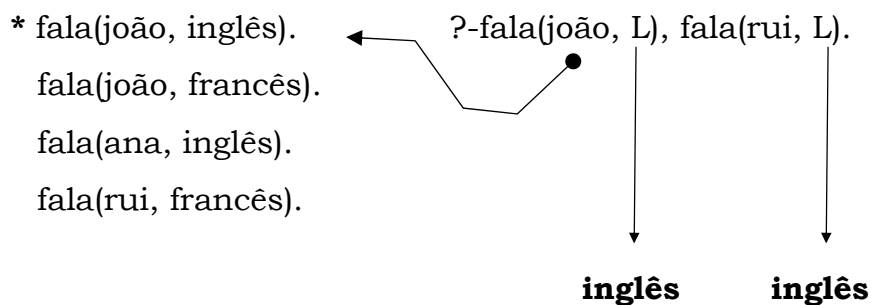
e a seguinte questão:

?- fala(joão, L), fala(rui, L).

Qual o procedimento Prolog para responder à questão?

1º Passo:

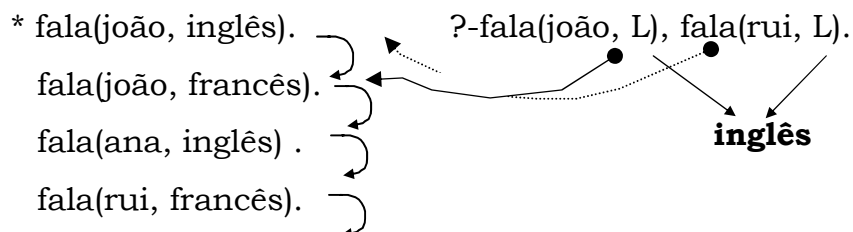
- O Prolog procura satisfazer o 1º “Goal”, **fala(joão, L)** iniciando uma pesquisa na BC de um facto a partir do qual este seja dedutível;
- Como a variável L se encontra não instanciada, pode ser instanciada (ou fazer “match”) com qualquer termo.



- A procura tem sucesso. A variável L foi instanciada e o facto correspondente marcado como o antecedente usado de momento.

2º Passo:

- O Prolog vai procurar satisfazer o 2º objectivo, **fala(rui, inglês)** após a instanciação encontrada.



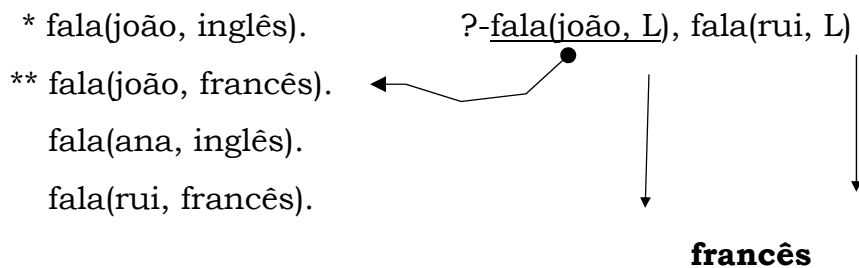
- A procura não tem sucesso. Não foi encontrada uma substituição/instanciação que satisfaça a conjunção dos objectivos.

Solução? Procurar "voltar atrás" e tentar encontrar *uma nova satisfação* do 1º "Goal". O identificador L é reinicializado.

Voltar atrás \Rightarrow **BACKTRACKING**

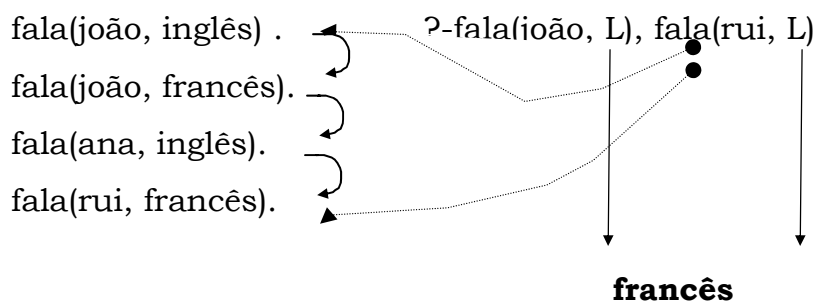
3º Passo:

- A tentativa de resatisfazer o 1º objectivo é iniciada procurando a partir do facto anteriormente marcado



- A procura tem sucesso para $\{L = \text{francês}\}$, e o facto é marcado.

4º Passo: Satisfação do 2º "Goal"?



Sucesso para $\theta = \{L = \text{francês}\}$

5º Passo Opcional: uso de ;

- Procura de resatisfação dos objectivos, procurando novas substituições, desprezando assim factos anteriormente marcados como instâncias.
- No exemplo apresentado a resposta será **no**.

RESUMO:

Execução de um Query ?- P, Q, R,

1. - Tenta satisfazer P, Q, R, ... (ESQ \rightarrow DIR)
2. - *Sucesso* \Rightarrow **Marca + Instanciação**
3. - *Insucesso* \Rightarrow **Backtracking** (ESQ \leftarrow DIR)
Resatisfação
Desinstanciação
4. - Insucesso no “Goal” mais à esquerda \Rightarrow
insucesso na conjunção !
5. - Opcional: Resatisfação a partir do início.

3.- REGRAS

- “Queries” conjuntivos interessantes definem ou explicitam, só por si, relações semânticas. Por exemplo o query

`?- pai(rui, X), homem(X).`

é uma forma de interrogar a BC sobre os *filhos-de rui*.

A relação *filho-de(X, Y)* não foi definida mas o “query” conjuntivo, na sua formulação, define-a *implicitamente*.

- **A definição explícita de novas relações a partir de relações existentes** faz-se na programação em lógica através de **regras**.

REGRAS assumem em Prolog a forma genérica:

$R :- R_1, R_2, \dots, R_n.$

Ex^{os}:

- **X é filho de Y se Y é pai de X e X é homem.** $\forall X, Y$

`filho(X, Y) :- pai(Y, X), homem(X).`

- **X passa a pp3 se X estuda e faz os trabalhos de pp3.**

`passa(X, pp3) :- estuda(X, pp3),
faz-trabalhos(X, pp3).`

- **de quem gosta X ?**

`gosta (X, Z) :- gosta(Z, X),
pratica(Z, tenis),
licenciado(Z).`

REGRAS COMO QUERIES

A primeira leitura que se pode fazer de uma regra é que ela é uma forma de expressar novas (ou mais complexas) interrogações tendo por base outras mais simples.

A regra

tio(X, Y) :- irmao(X, Z), pai (Z, Y).

permite que “queries” envolvendo o predicado *tio* possam ser feitos, cf.

?- tio(X,ana).

sendo certo que este “query” é transformado, segundo a regra, no “query” conjuntivo

?- irmao(X, Z), pai(Z, ana).

Esta perspectiva corresponde à *visão procedimental de uma regra*:

“Para responder à questão Quem é tio de quem, deve responder-se ao query conjuntivo é X irmão de Z e é Z o pai de Y?”

Nota:

Tal como para os factos, variáveis em regras são universalmente quantificadas, excepto as que são usadas no corpo da Cláusula e não aparecem no cabeçalho da mesma.

REGRAS: VISÃO LÓGICA

A visão ou interpretação lógica de uma regra consiste em assumi-la como uma *cláusula de Horn completa* onde o símbolo \leftarrow (i.e. :- no Prolog) denota uma implicação lógica (lendo-se **se**).

A regra

$$\text{avô}(X, Z) :- \text{pai}(X, Y), \text{pai}(Y, Z).$$

pode, logicamente, ser interpretada como:

Para todo X, Y, Z ,

X é o pai de Z se

X é o pai de Y e

Y é o pai de Z

ou ainda, se tal for mais conveniente, e invocando no corpo e não no cabeçalho,

Para todo X, Z ,

X é o avô de Z se

existe um Y tal que

X é o pai de Y e

Y é o pai de Z

A introdução de regras no esquema de dedução que vimos estudando, faz com que a *Lei do Modus Ponens* deva passar a ser considerada.

Modus Ponens:

$$A \leftarrow B$$
$$B \Rightarrow A$$
$$B$$
$$B$$
$$\text{-----}$$
$$A$$
$$\text{-----}$$
$$A$$

Regras recursivas permitem expressar certas relações complexas de forma simples, e são também adequadas para exprimir a manipulação de estruturas inerentemente recursivas (cf. listas e árvores)

Exº.:

$$\text{descende}(X, Z) :-$$
$$\text{filho}(X, Y),$$
$$\text{descende}(Y, Z).$$

$$\left\{ \begin{array}{l} \text{membro}(X, [X|T]) . \\ \text{membro}(X, [H|T]) :- \text{membro}(X, T) . \end{array} \right.$$

← Cláusula de
terminação

REGRAS DISJUNTIVAS

Prolog aceita também cláusulas disjuntivas. Por exemplo, a regra:

$$P :- Q ; R.$$

deve ler-se: “**P é verdadeiro se Q é verdadeiro ou R é verdadeiro**”

Esta regra é a síntese das suas regras seguintes:

$$P :- Q.$$

$$P :- R.$$

Nota:

$$P :- Q, R.$$

$$P :- S, T, U.$$

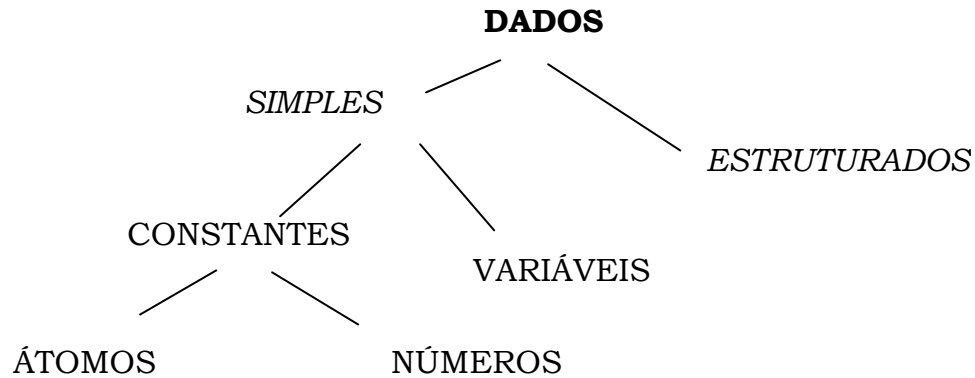
$$\Leftrightarrow$$

$$P :- Q, R; S, T, U;$$

lida como
 $P :- (Q, R); (S, T, U)$

PROLOG: SINTAXE E SEMÂNTICA.

DADOS



- **Átomos:**

rui	ana	x25	X	X_Y
==>	:::	..;	< - - >	
'Ana'	'Portugal'		'Prolog'	

- **Números:**

inteiros:	0	1975	-17
reais: 3.14	-0.007	10.75	

- **Variáveis:**

X	Res	Lista	__Y15	__W17
__	(variável anónima)			
tem-filhos(X) :- pai (X, __).				

*Âmbito léxico de 1 constante = **programa***

Âmbito léxico de 1 variável = **cláusula**

DADOS ESTRUTURADOS

- *ESTRUTURAS*
- *LISTAS*

REPRESENTAÇÃO DE ESTRUTURAS

- Os seus componentes são objectos simples ou estruturados, constantes ou variáveis;
- Ainda que sendo compostos são tratados como um simples e único objecto;
- Os componentes são agregados num único objecto através de um *FUNCTOR*, assumindo portanto a forma de um predicado.

Exº:



segmt(ponto(10,15), ponto(0,0))
data(1, 11, 1995)
mul(soma(3,10), sub(10,5)).

- *Estruturas* são representadas por termos Prolog, podendo ser vistas como árvores.
- A comparação de estruturas é um processo de comparação de termos designado por **Matching**.

MATCHING

Matching é um processo que aceita por entrada dois termos e verifica se estes são *equivalentes* ou idênticos, ainda que tal possa ser o resultado de certas instanciações das suas variáveis.

Exº:

data(d, m, 1910) = data(D1, maio, A1) ??

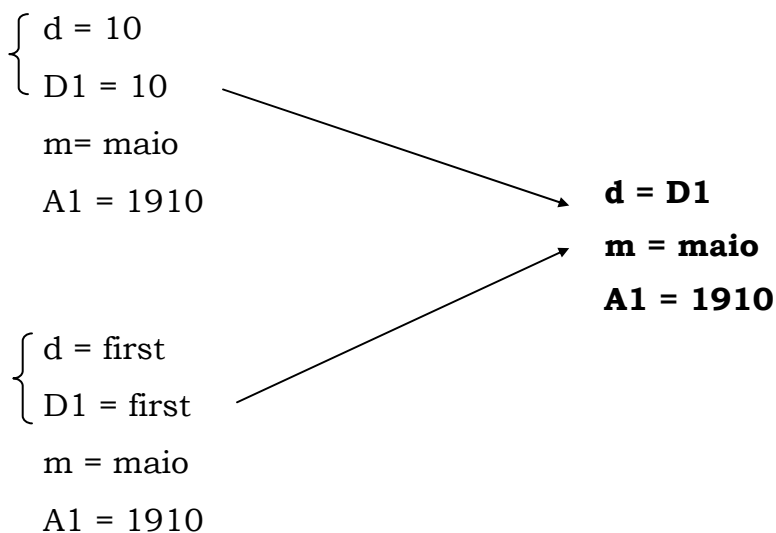
“match” para **d = D1**
as instanciações **m = maio**
A1 = 1910

Em Prolog, o teste é feito segundo o query:

?- data(d, m, 1910) = data(D1, maio, A1).

Nota:

No Prolog, “matching” resulta sempre na **mais geral instânciação** possível. De notar também o operador de matching **=**.



?- data(d, m, 1910) = data(D1, maio, A1),

`data(d, m, 1910) = data(15, m, A).`

`d = D1`

`m = maio`

`A1 = 1910`

`d = 15`

`D1 = 15`

`m = maio`

`A1 = 1910`

`A = 1910`

*instanciações
são
sucessivamente
mais específicas!*

ALGORITMO DE MATHCING

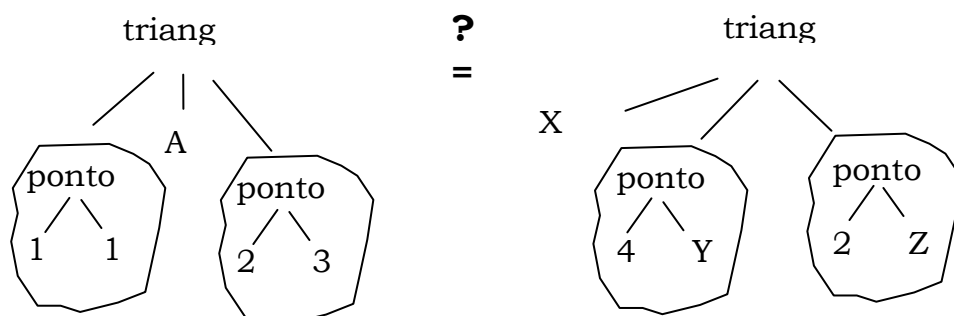
Dados dois termos S e T diz-se que “fazem match” nas seguintes circunstâncias:

1. - Se S e T são constantes fazem match se forem o mesmo objecto;
2. - Se S é uma variável, sendo T um termo qualquer, então fazem match, sendo S instanciada com T. Se T é, ao invés, uma variável e S um termo qualquer que não uma variável, T é instanciado com S.
3. - Se S e T são estruturas, então fazem match se:
 - a) têm o mesmo functor principal;
 - b) os argumentos correspondentes fazem match.
4. - A instanciação encontrada é determinada pelo “matching” dos vários componentes.

Nota:

O processo global de “matching” pode pois ser decomposto numa sequência de operações de “match” mais simples.

Exº 1:



```
?- triang(ponto(1, 1), A, ponto(2, 3)) =
   triang(X, ponto(4, Y), ponto(2, Z)).
```



triang = triang

ponto(1,1) = X

A = ponto(4,Y) \Rightarrow X = ponto(1,1)

ponto(2,3) = ponto(2,Z) Z = 3

Exº 2: **Matching como computação**

Consideremos dois factos universais sobre segmentos de recta que estabelecem as propriedades de segmentos horizontais e verticais.

```
vertical(seg(ponto(X,Y), ponto(X, Y1))).
```

```
horizontal(seg(ponto(X,Y), ponto(X1, Y))).
```

Procure agora, com base no algoritmo de "match" apresentado, e nestes factos, explicar as seguintes conclusões obtidas em função dos diferentes "queries".

?- vertical(seg(ponto(1,1), ponto(1,2))).

yes

?- vertical(seg(ponto(1,1), ponto(2,Y))).

no

?- horizontal(seg(ponto(1,1), ponto(2,Y))).

Y = 1

?- horizontal(seg(ponto(2,4), P)).

P = ponto(_165,4)

Pergunta bem mais interessante, e com resposta surpreendente, é a seguinte:

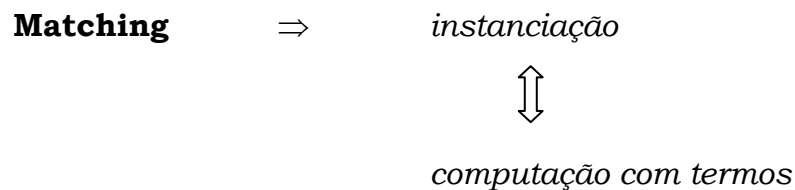
Existirá algum segmento simultaneamente vertical e horizontal?

?- vertical(Seg), horizontal(Seg).

Seg = seg(ponto(_120,_170), ponto(_120,_170))

onde _120 e _170 são variáveis cujos nomes são gerados internamente pelo Prolog.

Nota:



Ou seja, *matching de termos* é, só por si, um processo computacional, neste caso de características sintácticas já que se baseia na manipulação das representações dos termos Prolog.

REPRESENTAÇÃO DE LISTAS

Uma **lista** é uma estrutura que representa uma *sequência de objectos*, simples ou estruturados, sequência essa eventualmente nula.

A **lista vazia** representa-se em Prolog pelo átomo

[]

A **representação por enumeração** é realizada na forma,

[10, ana, [1 x, y], rui]

Considerando a sua definição **[Head | Tail]**, listas podem ser escritas de forma equivalente como:

$$[a, b, c] = [a | [b, c]] = [a, b | [c]] = [a, b, c | []]$$

Considerando a cauda de uma lista como um todo indivisível, são também representações particulares de listas genéricas, as seguintes:

[Head | Tail]

[H | T]

[Item1, Item2, ... | Resto]

OPERAÇÕES SOBRE LISTAS: DEFINIÇÕES.

- **membro**

membro(X, [X | Tail]).

membro(X, [X | Tail]) :- membro(X, Tail).

- **adição**

add(X, L, [X | L]).

- **remoção**

del(X, [X, T], T).

del(X, [Y | T], [Y | T₁]) :- del(X, T, T₁).

Ex^{os}: ?- del(a, [a, b, a, a], L).

L = [b, a, a];

L = [a, b, a];

L = [a, b, a];

no

}

não determinismo

?- del(10, L, [3, 7]).

L = [10, 3, 7];

L = [3, 10, 7];

L = [3, 7, 10];

no

} **possíveis
listas
originais**

? del(X, [17, 15, 10], [17, 10]).

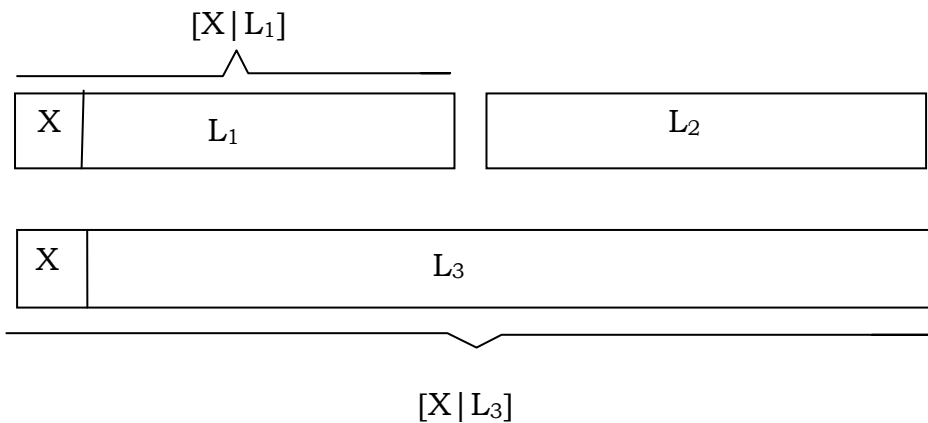
- **inserção em qualquer lugar**

inserir (X, Li, Lf) :- del (X, Lf, Li).

- **concatenação**

conc([], L, L).

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



- **membro usando concatenação**

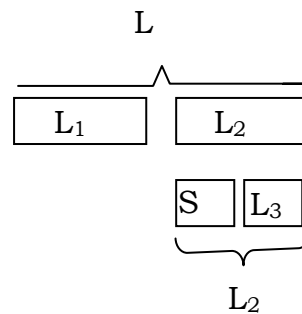
membro₁(X, L) :- conc(L₁, [X | L₂], L).

- **último elemento de uma lista**

- a) $\text{last}(X, L) \text{ :- conc}(_, [X], L).$
- b) $\text{last}(X, [X]).$
 $\text{last}(X, [X | T]) \text{ :- last}(X, T).$

- **sublista usando concatenação**

$\text{subl}(S, L) \text{ :-}$
 $\text{conc}(L_1, L_2, L),$
 $\text{conc}(S, L_3, L_2).$



EXERCÍCIOS SOBRE LISTAS:

- 1.- Escreva um predicado (ou procedimento) que determine o número total de ocorrências do átomo X numa lista L .
- 2.- Defina um predicado que determine se uma dada lista de números inteiros está ordenada.
- 3.- Escreva um predicado que determine o maior elemento de uma lista de números inteiros.
- 4.- Escreva um predicado que ordene uma lista de inteiros.
- 5.- Dada uma lista de inteiros, positivos e negativos, escreva um predicado que dê como resultado uma lista formada pela lista de todos os positivos e pela lista de todos os negativos da lista inicial.

ÁRVORES DE EXECUÇÃO / PROVA

Exemplo:

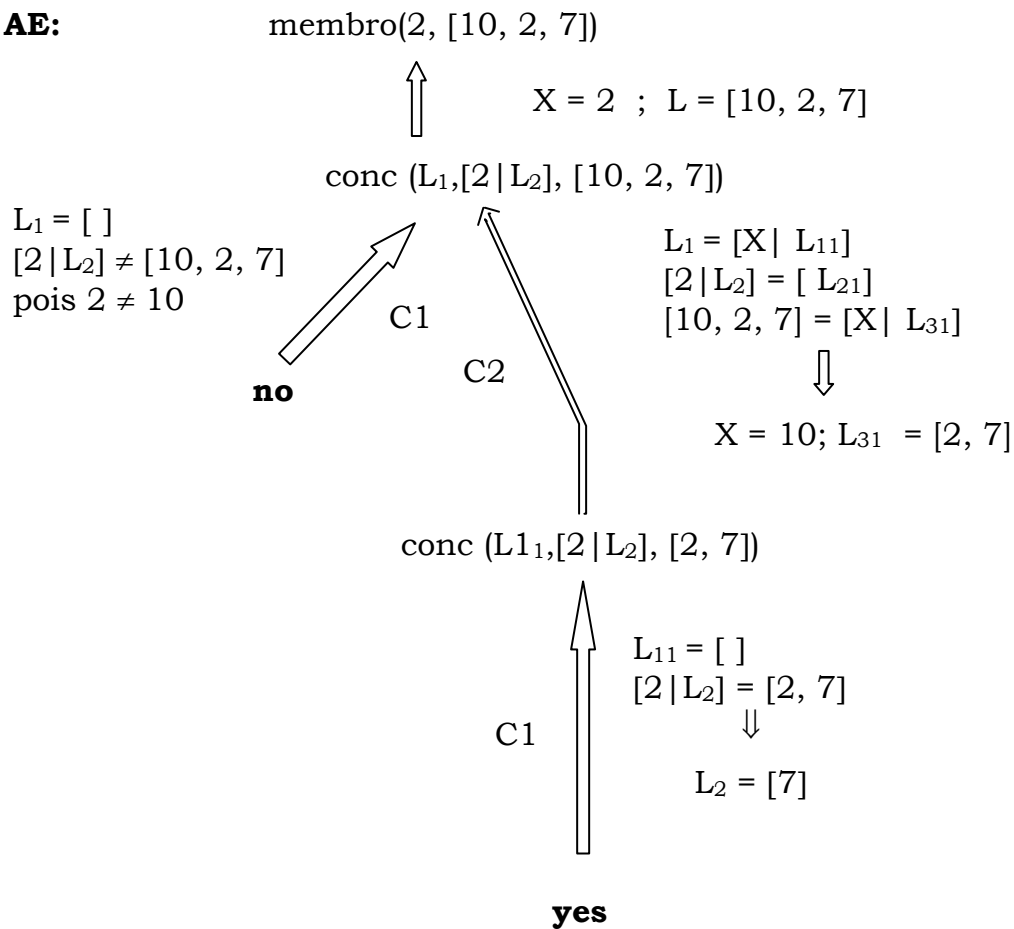
C1: $\text{conc}([], L, L).$

C2: $\text{conc}([X|L_1], L_2, [X|L_3]) \text{ :- conc}(L_1, L_2, L_3).$

$\text{membro}(X, L) \text{ :- conc}(L_1, [X|L_2], L).$

`?- membro(2, [10, 2, 7]).` % o query a executar !

AE:



- **inversão**

reverse([], []).

reverse([H | T], Lr) :-
 reverse(T, Tr),
 conc(Tr, [H], Lr).

OPERADORES ARITMÉTICOS e RELACIONAIS

São operadores *infixos* mas que só por si não implicam a realização do cálculo do valor dos termos onde surgem como funtores.

São os usuais:

+	-	*	/	mod	div
>	<	>=	=<	:=	=\	=

Assim, o “query” seguinte produz o resultado apresentado,

?- Z = 3*4.

Z = 3 * 4

ou seja, o resultado esperado de um normal “matching” de termos.

A instanciação de uma variável com o valor resultante do cálculo (“evaluation”) de um termo numérico é realizada recorrendo ao operador **is** que “força” a realização do cálculo.

?- Z = is 3*4.

Z = 12

Nota: O argumento (ou termo à direita) do operador **is** é uma expressão aritmética envolvendo constantes, operadores e variáveis.

Dado que o operador **is** força o cálculo do valor de tal expressão, é fundamental que tais variáveis estejam já instanciadas.

Nota: Expressões envolvendo operadores relacionais são resolvidas naturalmente nos valores verdadeiro ou falso.

Exº:

?- pop(Pais, Pop), Pop >= 50.000, Pop =< 5.000.

Nota:

X = Y “matching” \Rightarrow não \exists cálculo

X ::= Y igualdade \Rightarrow cálculo

Exº:

?- 3 * 5 = 10 + 5.

no

?- 2 + 4 ::= 0 + 6.

yes

?- 3 + X ::= Y + 1.

X = 1

Y = 3

Exº: **Máximo divisor comum entre X e Y**

1) Se $X = Y \Rightarrow \text{mdc}(X, Y) = X$

mdc(X, X, X).

2) Se $X < Y \Rightarrow \text{mdc}(X, Y) = \text{mdc}(X, Y-X)$

mdc(X, Y, D) :- X < Y, Y1 is Y-X, mdc(X, Y1, D).

3) Se $X > Y \Rightarrow$ o mesmo que 2 trocando X com Y

mdc(X, Y, D) :- X > Y, mdc(Y, X, D).

Exº: Número de elementos de uma lista

Versão 1:

```
len1([ ], 0).
```

```
len1([_ | T], N) :- len1(T, N1), N = 1 + N1.
```

```
?- len1([a, [b, c], d, e], c).
```

```
c = 1 + (1 + (1 + (1+0))).    ⇐ Porquê?
```

Versão 2:

```
len2([ ], 0).
```

```
len2([_ | T], N) :- N = 1 + N1, len2(T, N1),
```

```
?- len2([[a, b], c, d], c).
```

```
c = 3
```

Versão 3 \equiv Versão 2

```
len3([ ], 0).
```

```
len3([_ | T], 1+ N) :- len3(T, N).
```

Versão 4 (usa a versão 1):

```
?- len1([a, b, c], c), L is C.
```

```
c= 1 + (1 + (1+0)).
```

```
L = 3.
```

Versão 5:

```
len5([], 0).
```

```
len5([_ | T], N) :- len5(T, N1), N is 1 + N1.
```

Nota:

Estude cada caso. Construindo as respectivas árvores de execução para o mesmo “Query”.

Exº: Máximo de uma lista não vazia

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

```
maxList([X, Y | T], Max) :-  
    maxList([Y | T], Maxim),  
    max(X, maxim, max).
```

Exº: Somatório dos elementos de uma lista

```
somaLista([], 0).
```

```
somaLista([H | T], Soma) :-  
    somaLista(T, SomaResto),  
    soma is H+ SomaResto.
```

Exº: Concatenação distribuída, ou seja, dada uma lista de listas, reduzi-la a uma lista simples.

```
concdist([1, 2], [4], 5, 6, [7, 8])
```

```
→ [1, 2, 4, 5, 6, 7, 8]
```

```
concdist([], []).
```

concdist(X, [X]).

concdist([H | T], Lfinal) :- concdist(H, Tratahead),
concdist(T, Tratatail),
conc(Tratahead, Tratatail, Lfinal).

Nota: Procure executar e analisar os resultados deste programa.

MODELO DE EXECUÇÃO DO PROLOG

SEMÂNTICA PROCEDIMENTAL

- **Escolha de “Goal”** \Rightarrow o mais à esquerda

$? - g_1, g_2, \dots, g_n$

- **Procura de uma Cláusula** \Rightarrow Sequencial + BackTracking
- **Mecanismo de Base** \Rightarrow “Goal Stack”
- **Estratégia** \Rightarrow pop (“Goal”) \rightarrow push (derivados)
- **Travessia das Árvores de Procura** \Rightarrow “Depth First” (profundidade)
- **Ramo Infinito numa Árvore de Procura** \Rightarrow Computação Infinita

Nota:

Não sendo importantes do ponto de vista lógico, do ponto de vista de execução são relevantes as seguintes questões:

- a) A ordem das regras tem influência na execução ou não?
- b) A ordem dos “subgoals” de um “Query”, ou das sub-cláusulas de uma regra, tem influência na execução e resultados ou não?
- c) A execução é controlável para efeitos de optimização?

A resposta às questões a) e b) será dada através de um exemplo e respectiva generalização de resultados.

Exº: Considere-se a seguinte base de conhecimento formada por 6 factos e definições alternativas de uma relação *antepassado* baseada numa relação *pai*.

Factos:

pai(pedro, bino).
pai(toni, bino).
pai(toni, luisa).
pai(bino, ana).
pai(bino, paulo).
pai(paulo, joão).

Regras Alternativas

$\text{antep}_1(X,Z) \text{ :- pai}(X, Z).$
 $\text{antep}_1(X,Z) \text{ :- pai}(X, Y), \text{antep}_1(X,Z).$
 $\text{antep}_2(X,Z) \text{ :- pai}(X, Y), \text{antep}_2(Y,Z).$
 $\text{antep}_2(X,Z) \text{ :- pai}(X, Z).$

$\text{antep}_3(X,Z) \text{ :- pai}(X, Z).$
 $\text{antep}_3(X,Z) \text{ :- antep}_3(X, Y), \text{pai}(Y, Z).$

$\text{antep}_4(X,Z) \text{ :- antep}_4(X, Y), \text{pai}(Y, Z).$
 $\text{antep}_4(X,Z) \text{ :- pai}(Y, Z).$

Analisemos agora para cada uma destas definições a efectiva execução do “Query”

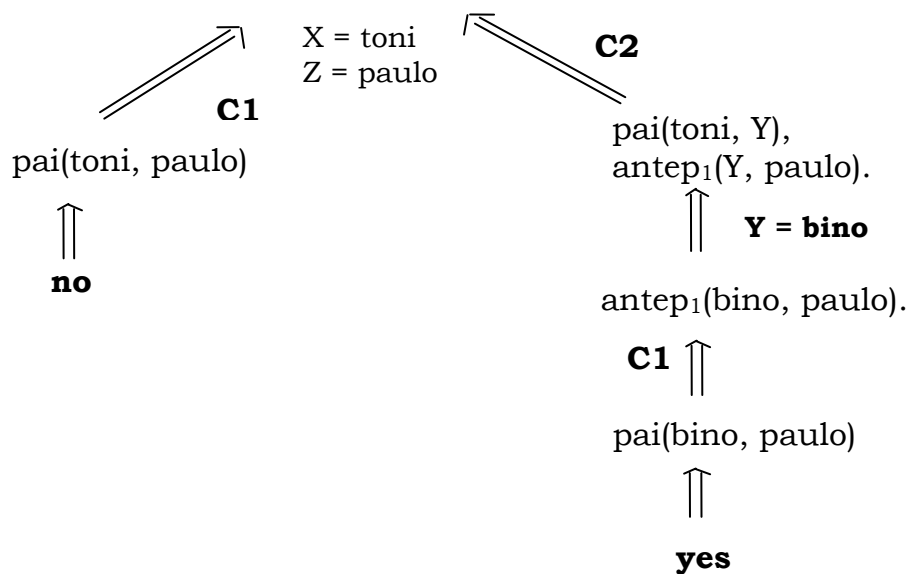
?- antepX(toni, paulo).

A) $\text{antep}_1(X,Z) \text{ :- pai}(X, Z).$ C1

$\text{antep}_1(X,Z) \text{ :- pai}(X, Y), \underline{\text{antep}_1(X,Z)}.$ C2

?- antep₁(toni, paulo).

AE: $\text{antep}_1(\text{toni}, \text{paulo}).$



Nota:


pai(tony, Y), antep₁(Y, paulo).

subcláusula **antep₁(Y, paulo)**, pelo que o resultado seria **no**.

$$\text{antep}_2(X,Z) \text{ :- pai}(X, Y), \text{ antep}_2(Y,Z). \quad \text{C1}$$
$$\text{antep}_2(X,Z) \text{ :- pai}(X, Z). \quad \text{C2}$$

?- antep₂(toni, paulo).

AE: antep₂ (toni, paulo)

C1  **X = toni ; Z = paulo**

pai (toni, Y),
antep₂ (Y, paulo).

F2  Y = bino

antep₂(bino,
neule) ⇐ !!

c1 **c2**

↑ ↗

pai(bino, Y1),
antep2(y1, paulo). pai(bino, paulo)

↑



```

pai(bino, paulo)

```

yes

Y1 = paulo

Y1 = ana  
antep₂ (ana, paulo)

antep₂ (ana, paulo)

→ pai(ana, Y2),
antep₂(Y2, paulo).

no

antep₂(paulo, paulo)

```

    pai(paulo, Y2),
    antep2(Y2, paulo).

```

Y2 = joao

antep₂ (joao, paulo)

→ pai(joao, Y3),
antep₂(Y3, paulo).

no

AE



Que dizer quanto aos caminhos percorridos?

?- antep₄(toni, paulo).

AE

antep₄(toni, paulo)
C1 ↑ **X = toni ;Z =**
*antep₄(toni, Y),
pai (Y, paulo).
C1 ↑
*antep₄(toni, Y1),
pai (Y1, Y)
pai(Y2, paulo).
C1 ↑
*antep₄(toni, Y2),
pai (Y2, Y1)
pai (Y1, Y),
pai(Y, paulo).
C1 ↑
etc.
etc.
etc.

Nota:

Temos uma situação de recursividade infinita.

Trocar a ordem das cláusulas resolveria o problema?

OBS. RELATIVAS AO EXEMPLO:

- 1) O procedimento **antep** define-se usando 2 cláusulas, uma das quais com 2 subcláusulas ou “goals”. No total, 4 variações podem ser testadas para o mesmo exemplo.
- 2) O significado lógico ou declarativo é igual, mas o significado procedimental, ou de execução, é alterado.
- 3) As definições de **antep1** e **antep2** tentam em primeiro lugar as possibilidades mais simples de solução. As definições de **antep3** e **antep4** tentam em primeiro lugar as mais complexas.
- 4) **antep1** e **antep2** atingem sempre uma solução, qualquer que seja o “query”, sendo porém antep2 bastante ineficiente;

antep3 e **antep4** podem gerar recursividade infinita (cf. recursividade à esquerda), antep4 sempre e antep3 em certas situações

(exº ?- antep3(luísia, joão).)

Conclusões:

Ordem das regras	⇒		= <i>Árvore de Procura</i>
A, B			≠ <i>Ordem de Procura</i>
			≠ <i>Ordem nas Soluções</i>

Ordem dos “Goals”	⇒		≠ <i>Árvore de Procura</i>
A, C			≠ <i>Ordem nas Soluções</i>

Recursividade à Esquerda	⇒		<i>Procura Potencialmente Infinita;</i>
			<i>Técnica a evitar!</i>

Exº:

`casado(X, Y) :- casado(Y, X).`

⇓ **substituir por**

`sao_casados (X,Y) :- casado(X,Y).`
`sao_casados(X,Y) :- casado(Y,X).` } **a ∨ b**

**Eficiência
em Geral** ⇒

<i>Procurar satisfazer primeiro os objectivos mais simples!</i>

EXERCÍCIOS:

- 1) Procure amadurecer estas ideias criando pequenas bases de conhecimento onde são inseridos predicados do tipo *pai(-,-)*, *mae(-,-)*, *irmao(-, -)*, *irma(-, -)*, *homem(-)* e *mulher(-)*;

Escreva as regras para as relações *tio(-, -)*, *prima(-, -)*, etc., usando diferentes ordens para as regras e para as sub-cláusulas, e verifique as diferentes árvores de execução resultantes.

- 2) Recorde o procedimento **membro(-, -)** apresentado anteriormente. Faça o mesmo tipo de estudo relativamente à ordem das regras para o “query” particular:

`membro (7, [1, 6, 7]).`

CONTROLO DO BACKTRACKING

- A ordem de escrita das regras e dos “Goals” influencia, logo permite o controlo, a execução de um programa;
- O mecanismo de BACKTRACKING do Prolog é automático, como se viu. Para o programador tal é, em geral, uma vantagem.
- Casos existem porém em que um BACKTRACKING incontrolado conduz a programas ineficientes.
- O mecanismo existente em Prolog para evitar o BACKTRACKING a partir de um dado ponto da execução, designa-se por CUT e é representado pelo símbolo !, representando um predicado particular.
- Procuremos distinguir dois tipos de CUT:

GREEN CUTS *não afectam o significado lógico do programa, sendo usados visando maior eficiência pois permitem “cortar” (pruning) caminhos (ou passos) da computação que não conduzem a novas soluções.*

RED CUTS *alteram o significado lógico de um programa.*

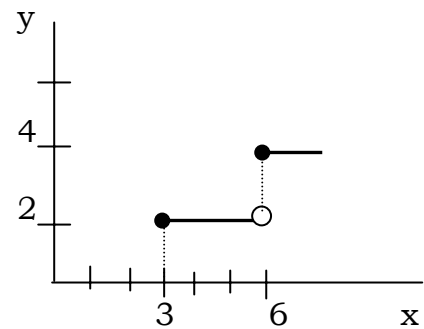
- Estudaremos o mecanismo de CUT como uma forma de implementação em Prolog da negação sob a forma de negação por falha.

Uma classe de problemas para os quais a utilização de CUT é útil, é a classe dos problemas determinísticos, os quais se formulam à custa de um conjunto de regras disjuntas.

Regras disjuntas \Rightarrow sendo mutuamente exclusivas, no máximo 1 regra poderá ter sucesso, logo sendo desnecessário realizar outras tentativas caso a candidata falhe.

Exemplo 1: Função em Degrau

$$f(x) = \begin{cases} 0, & x < 3 \\ 2, & 3 \leq x \wedge x < 6 \\ 4, & 6 \leq x \end{cases}$$



Teremos em Prolog a relação binária $f(x, y)$ definida como:

$$f(x, 0) :- x < 3 \quad \text{R1}$$

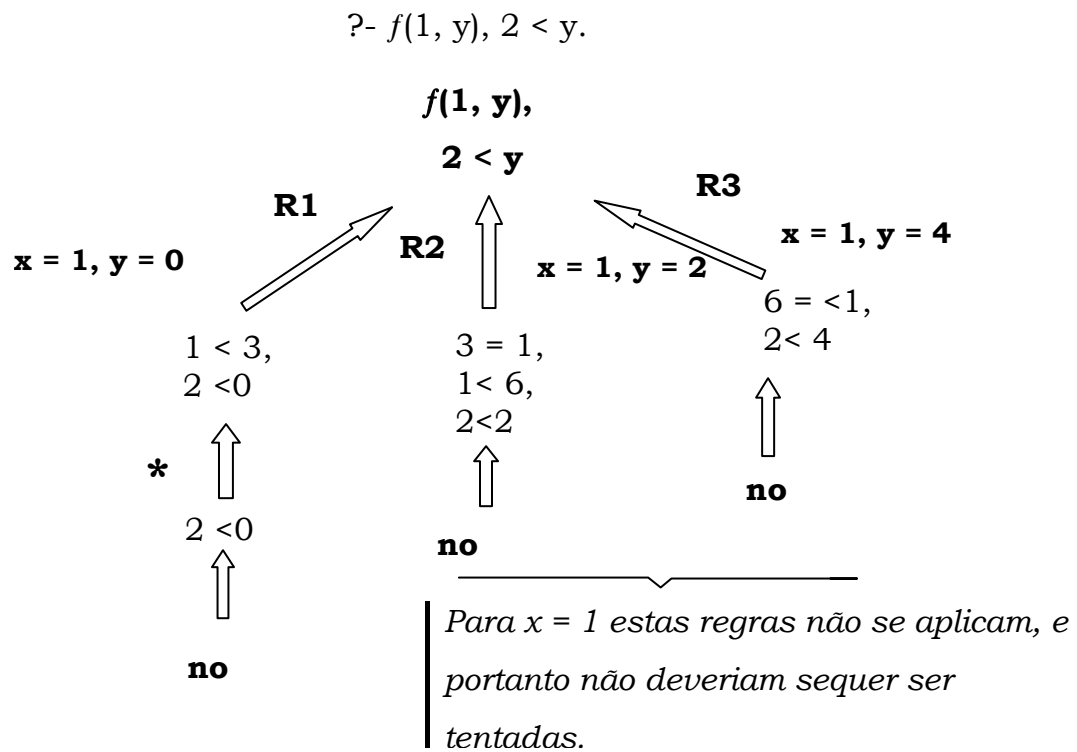
$$f(x, 2) :- 3 \leq x, x < 6 \quad \text{R2}$$

$$f(x, 4) :- 6 \leq x. \quad \text{R3}$$

Consideremos então o seguinte “query”:

$$?- f(1, y), 2 < y.$$

e analisemos a execução equivalente representada pela árvore de execução que se apresenta na página seguinte.



Nota:

Repare-se que a regra R1 sabe-se ser ou não satisfeita no ponto indicado com um *.

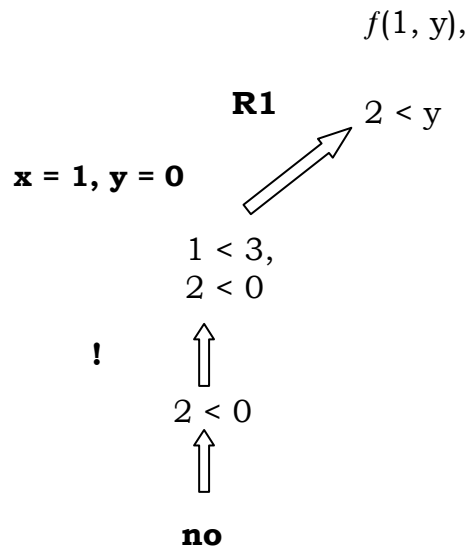
Se pretendermos, em tal ponto, e caso o seu resultado seja a *falha*, que para lá deste ponto o Prolog não realize *backtracking*, então a regra deverá ser escrita inserindo, em tal ponto, o símbolo de CUT. Teríamos então:

$f(x, 0) :- x < 3, !$

O “query”

?- $f(1, y), 2 < y$.

dará agora origem à seguinte árvore:



Nota:

A existência do CUT evita backtracking a partir do ponto indicado, pelo que as regras R2 e R3 não serão tentadas.

O programa final, logicamente equivalente mas mais eficiente, seria então:

```
f(x, 0) :- x < 3, !.
f(x, 2) :- 3 =< x, x < 6, !.
f(x, 4) :- 6 =< x.
```

Nota:

É de notar que se no programa anterior removermos os ! as soluções encontradas são as mesmas, ou seja, o significado declarativo não é modificado, apenas o seu significado procedimental, ou seja, a sua forma de execução.

Porém, tal nem sempre é verdade, como veremos adiante. Continuemos de momento a considerar **GREEN CUTS**.

Exº: **Máximo de 2 números**

$\max(x, y \ x) :- x \geq y.$ $\max(x, y \ y) :- x < y.$	}	<i>Mutuamente exclusivas</i>
---	---	----------------------------------

Sendo as regras exclusivas, podemos reformulá-las como:

```
se  $x \geq y$   
    então  $Max = x$   
    senão  $Max = y$ 
```

o que é equivalente ao seguinte programa Prolog

```
max( $x, y, x$ ) :-  $x > y, !$ .  
max( $x, y, y$ ).
```

Nota:

Tal como no exemplo anterior, o ! indica aqui a natureza mutuamente exclusiva das regras.

Exº: “Merge” de duas lista ordenadas

```
merge([ ],  $Y_0, Y_0$ ).  
merge( $X_0, [ ], X_0$ ).  
merge( $[X | X_0], [Y | Y_0], [X | L_0]$ ) :-  
     $x < y$ , merge ( $X_0, [Y | Y_0], L_0$ ).  
  
merge( $[X | X_0], [Y | Y_0], [X | L_0]$ ) :-  
     $x > y$ , merge( $[X | X_0], Y_0, L_0$ ).  
  
merge( $[X | X_0], [Y | Y_0], [X, Y | L_0]$ ): -  
     $x = y$ , merge( $X_0, Y_0, L_0$ ).
```

Mais uma vez, para cada possível caso, uma e uma só regra se aplica.

Em particular quanto à comparação entre X e Y, apenas um dos testes será verdadeiro pelo que os outros não devem ser tentados pois irão falhar.

Programa final com CUTS:

```
merge([ ], Y0, Y0):- !.  
  
merge(X0,[ ], X0):- !.  
  
merge([X|X0], [Y|Y0], [X|L0):-  
    x<y, !, merge(X0, [Y|Y0], L0).  
  
-----  
x>y,!, -----
```

Exº: **“CUT” como teste de pré-condição.**

Juntar um elemento a uma lista *mas sem introduzir duplicados.*

A) **Não Satisfatório**

```
add(X, L, L) :- member(X, L).  
  
add(X, L, [X|L]).  
  
?-add(10, [5, 10, 7], L).  
L = [5, 10, 7];  
L = [10, 5, 10, 7]
```

B) **Correcto**

```
add(X, L, L) :- member(X, L), !.  
  
add(X, L, [X|L]).  
  
?- add(10, [5, 10, 7], L).  
L = [5, 10, 7];  
  
?- add(x, [5, 9], L).  
L = [5, 9]  
X = 5                                para member(X, [X|T]) :- !
```

?-add(5, [4, 2, X], L).

L = [4, 2, 5]

X = 5

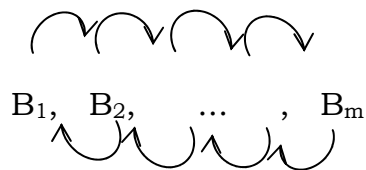
CUT: Caso Geral

Considere-se que um "Goal" G fez "match" com uma cláusula C da forma:

$$H :- B_1, B_2, \dots, B_m, !, B_n, \dots, B_z.$$

Teremos as seguintes regras de execução

1. Os "subGoals" de B_1 a B_m procuram satisfazer-se segundo as regras normais.



2. O "CUT" pode ser visto como um "Goal" que é automaticamente satisfeito a seguir a B_m .
3. Quando o "CUT" é encontrado e executado, a solução encontrada para os "SubGoals" de B_1 a B_m é fixada, e todas as possíveis alternativas desprezadas.

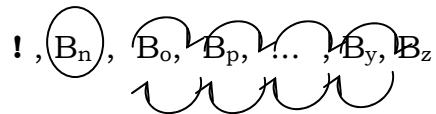
São em particular desprezadas:

- a) outras cláusulas C existentes, ou seja, com cabeçalhos H que poderiam fazer "match" com G

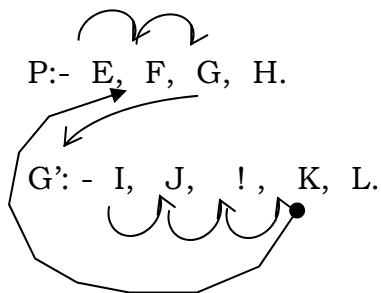
$$\begin{aligned} & \text{H} :- B_1, \dots, B_m, !, B_n, \dots, B_z. \\ \times & \text{H} :- B_{11}, \dots, B_{m1}, !, B_n, \dots, B_{z1}. \\ \times & \text{H} :- B_{111}, \dots \end{aligned}$$

- b) Outras possíveis instanciações dos "SubGoals" de B_1 a B_m .

4. A tentativa de satisfação dos “subGoals” à direita do !, excepto para o primeiro, segue o mecanismo normal



5. Backtracking é realizado apenas até ao !
6. Se o backtracking” atinge o !, porque B_n falhou, então o ! falha e a procura prossegue no “Goal” imediatamente anterior ao “Goal” G que fez “match” com H.



Nota:

Ao serem desprezados muitos possíveis caminhos da árvore de procura, o mecanismo de CUT comporta-se como um mecanismo de "corte" ou "poda" (cf. inglês *pruning*) de ramos da árvore.

Objectivo: *Eficiência* \Rightarrow $\left\{ \begin{array}{l} - \text{Espaço} \\ - \text{Tempo} \end{array} \right.$

NEGAÇÃO COMO FALHA

Com base no mecanismo de CUT o Prolog permite implementar uma forma limitada de *negação* designada por **negação como falha**, em particular

recorrendo a um predicado particular impossível de satisfazer, logo que falha sempre, por isso designado **fail**.

Negação \Rightarrow **Cut-Fail**

Exº.

Como dizer em Prolog que:

“O Rui gosta de todas as disciplinas excepto Inglês”.

P1: O Rui gosta de todas as disciplinas.

```
gosta(rui, X) :- disciplina(X).
```

P2: Excepto inglês (ou, mas não de inglês)

???

Reformulação:

“Se X é inglês

então é falso que ‘rui gosta de X’

senão se X é uma disciplina o rui gosta

```
gosta(rui, X) :-  
    ingles(X), !, fail.      }   É falso  
  
gosta(rui, X) :- disciplina(X).
```

Torna-se importante definir um predicado **not** tal que

not(Goal)

seja verdadeiro se Goal é falso, ie. não provável.

Esta definição pode escrever-se como:

```
not(P) :- P, !, fail.  
not(P).
```

\longleftarrow *P é metavariável;*
P \leftarrow predicado.

Temos pois que:

$P \text{ sucede} \Rightarrow \text{not}(P) \text{ falha}$

$P \text{ falha} \Rightarrow 2^{\text{a}} \text{ regra not}(P) \text{ sucede.}$

Considerando a partir de agora **not** como um predicado pré-definido e pré-fixado, poderíamos reescrever a regra anterior como:

```
gosta(rui, X) :- disciplina(X),          . not → \+   ou
                  not(ingles(X)).        . not(ingles(X))
```

Questão:

Qual o impacto do mecanismo de **cut-fail** na semântica declarativa ou lógica dos programas Prolog?

Resposta:

Pode fazer com que esta deixe de estar correctamente associada à semântica procedimental !

Ponto 1: Ordem das Regras

- A definição de not com base nas cláusulas

```
not(P) :- P, !, fail.
```

```
not(P).
```

contemplou apenas duas situações, P falha ou P tem sucesso.

- A ordem das regras é neste caso essencial.

Até aqui a ordem das regras apenas tinha influência na ordem das soluções.

Agora, a ordem das regras determina o significado do programa !

- Quando tal acontece, o procedimento deve ser escrito como uma única unidade, e não como uma colecção de cláusulas.

`not(P) :- P, !, fail ; not(P).`

Ponto 2: Terminação

- A terminação de **not(P)** depende da terminação de **P**

P termina \Rightarrow not(P) termina

P não termina \Rightarrow not(P) pode ou não terminar

Exº:

`casado(rui, ana)`

`casado(X,Y) :- casado(Y,X).`

?- not(casado(rui, ana)).

no (por falha)

?- casado(rui, ana).

yes

Exº:

`casado (x, y) :- casado (y, x).`

`casado(rui, ana)`

?- not(casado(rui, ana)).

memory error

?- casado(rui, ana)

memory error

Ponto 3: Ordem de Travessia

- Quando **not** é usado em conjunção com outros “goals” algumas anomalias podem igualmente acontecer, cf.


```
estudante_solteiro(X) :- not(casado(X)), estudante(X).  
estudante(rui).  
casado(paulo).
```

?- estudante_solteiro(E). \Rightarrow **no !!?**

```
not(casado(E)),                    E = rui ??  
estudante(E)  
  ↑ no  
casado(E), !, fail  
  ↑ E = paulo  
yes
```

Solução: troca das cláusulas.

```
estudante_solteiro(X) :- estudante(X),  
                        not(casado(X)).
```



?- estudante_solteiro(E).

E = rui

OBS:

A implementação de negação usando **cut-fail** nem sempre funciona correctamente para “goals” não-básicos (i.e. *non-Ground*), ou seja, “goals” contendo variáveis.

Compete ao programador, por análise estática do programa, verificar que cláusulas contendo *not* só são resolvidas quando apenas envolvem termos-base.

“RED CUTS”

- A introdução de certos cuts num programa Prolog pode fazer com que a semântica lógica deixe de corresponder à semântica resultante da execução.
- Considere-se o programa:
 $P :- A, B.$
 $P :- C.$

O significado lógico é dado por

$$P \Leftrightarrow (A \wedge B) \vee C$$

Se alterarmos a ordem das regras e/ou das subcláusulas temos variações lógicas equivalentes

$P :- C.$	$P :- B, A.$	$P :- C.$
$P :- A, B$	$P :- C$	$P :- B, A.$
$P \Leftrightarrow C \vee (A \wedge B)$	$P \Leftrightarrow (B \wedge A) \vee C$	$P \Leftrightarrow C \vee (B \wedge A)$

Vamos agora introduzir um **cut**

$P :- A, !, B.$

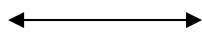
$P :- C.$

O significado lógico passa a ser

$$P \Leftrightarrow (A \wedge B) \vee (\neg A \wedge C)$$

Trocando ainda as regras,

$P :- C.$



$P :- C.$

$P :- A, !, B.$

$P :- A, B.$

teríamos:

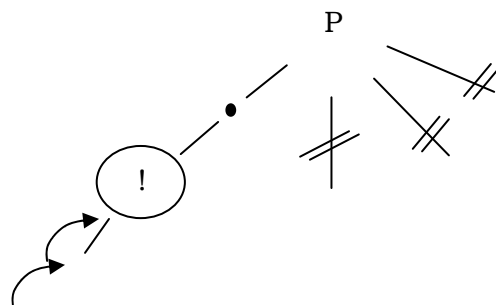
$$P \Leftrightarrow C \vee (A \wedge B).$$

Conclusão:

- Torna-se evidente que todos os cuts que alteram o significado declarativo dos programas devem ser tratados com cuidado.
- RED CUTS podem tornar os programas mais eficientes mas também os tornam mais ilegíveis do ponto de vista declarativo.
- Um profundo conhecimento da semântica procedimental, ou de execução, é exigido quando se programa com CUTS.

CUT e CUT-FAIL: Resumo

- A facilidade do CUT permite eliminar o “Backtracking” aumentando a eficiência da execução dos programas.



- O mecanismo de CUT é particularmente expressivo na formulação de problemas sob a forma de regras disjuntas, segundo o esquema geral:

se condição

então conclusão1

senão conclusão2

- A combinação do CUT com FAIL possibilita implementar uma forma de negação não lógica, por isso designada negação por falha ou seja, o “query”.

?- not(humano(rui)).

Ao produzir como resposta **yes**, apenas nos permite concluir que

“Não existe informação no programa para provar que humano(rui) é verdadeiro”.

Assim, **not(humano(rui))** *sucede* !

- De facto, o Prolog não tenta satisfazer o “goal” not(homem(rui)) directamente, antes tentando provar o oposto, ie. que é verdade homem(rui). O insucesso desta prova é assumida como o sucesso do “Goal”.
- Este raciocínio baseia-se na designada **assunção do mundo fechado** (de CWA: Closed World Assumption) que se baseia no princípio de que tudo o que é verdadeiro existe na BC ou dela é inferível.
- Assim, tudo o que segundo tal princípio não existe no programa ou não é do mesmo inferível é *falso*, sendo a sua negação verdadeira.
- Esta assunção do mundo fechado é muito importante que seja entendida já que, em geral, não pensamos deste modo. De facto, só porque não introduzimos numa BC que

humano(paulo)

não pretendemos afirmar que

“paulo não é humano”

ainda que o “query”

?- not(humano(paulo)).

dê como resultado **yes**, exactamente interpretado (erroneamente!) desse modo.

- Finalmente, qualquer que seja a utilização do CUT ela deve ser realizada com muito cuidado, o que pressupõe uma boa compreensão da sua semântica procedimental.
- É em particular importante entender as circunstâncias em que a utilização do CUT preverte a semântica formal de um programa Prolog no sentido em que

SEMÂNTICA LÓGICA ≠ SEMÂNTICA PROCEDIMENTAL

- Na situação anterior, uma possível, ainda que insuficiente, documentação adicional do programa é aconselhável !
- Um predicado muito interessante existente no Prolog é o predicado **true** inverso do **fail** (e não do inexistente false!!). Ainda que a sua semântica lógica seja óbvia, a sua utilização no Prolog não o é tanto.

Não se esclarecendo este último ponto, deixam-se dois exemplos de utilização que poderão revelar a sua utilidade.

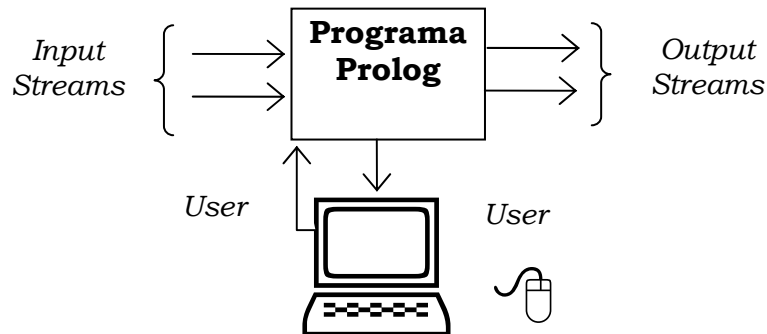
P :- A, B, true, C.

Q :- true, X, Y.

Nota: Combine true com !

I/O EM PROLOG

Comunicação com ficheiros



Filename (lógico) \longrightarrow **Stream (físico)**

DEC-10 Prolog	}	predicados sobre "filenames"
SICSTUS-Prolog		predicados sobre "streams"

I/O em Prolog	{	File I/O
		Character I/O
		Stream I/O
		Term I/O
		(Socket I/O)

Analisaremos de seguida os vários tipos de predicados de I/O, apresentando-os e fornecendo exemplos da sua utilização.

FILE/STREAM I/O

- No início da execução de um programa

Input stream	→	user-input	(stdin)
Output stream	→	user-output	(stdout)
		user-error	(stderr)

- A input stream pode ser associada a um outro ficheiro usando o predicado

see(Filename)

- A input stream pode ser associada a um outro ficheiro pelo predicado

tell(Filename)

- Um ficheiro especial é designado por user. É associado às “streams” normais I/O, pelo que ler do teclado consiste em escrever

see(user)

e escrever no ecrã consiste em usar

tell(user)

- Os predicados 0-ádicos see e close permitem fechar as “streams” correntes e voltar a activar as “streams” iniciais.
- Os predicados

seeing(F)

telling(F)

permitem determinar os nomes dos ficheiros actualmente associados às “streams” de I/O.

- Os predicados seen e told permitem fechar, respectivamente, os actuais ficheiros de entrada e de saída.

O predicado que permite associar um dado nome de ficheiro a uma dada “stream”, para leitura, escrita/criação, escrita/no fim, é o predicado

open(+Fname, +modo, -stream)

onde	+	=	argumento instanciado
	-	=	argumento a instanciar

FILE I/O

- Programas

consult(:Files) ?- [file].

reconsult(:Files) \longleftrightarrow ?- [user].

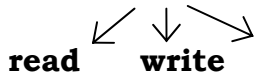
compile(:Files) ?- [a, b, c].

load(:Files)

Nota:

- :Files pode ser uma lista de “ficheiros”.
- *reconsult* reescreve, ou seja, sobrepõe factos e regras !

Principais Predicados pré-existentes

read(?Termo)	lê um termo, terminado por . e instancia Termo. ? significa que o argumento pode estar instanciado antes ou não.
write(?Termo)	→ output stream
display(?Termo)	→ standard output stream
print(?Termo)	→ “pretty printing”
nl	New line
get0(?N)	N is -1 se EOF; N is ASCII(char)
get(?N)	N is ASCII(char), char ≠ caracter de “layout”
put(+N)	escreve o caracter com código = N
tab(+N)	escreve N espaços
open(+Filemane, + modo, -stream)	
	 <pre> graph TD open[open] --> read[read] open --> write[write] open --> append[append] </pre>
close(+FouS)	fecha a “stream” dada ou a “stream” associada ao “Filemane” argumento
current_input(?Stream)	determina a “stream” corrente para I/O
current_output(?Stream)	
set_input(+ Stream)	define “stream” actual
set_output(+Stream)	
compile(:Files)	compilação de ficheiros
load(:Files)	“carrega” ficheiros compilados
Term1 == Term2	comparação “literal” de instâncias
Term1 \== Term2	
sort(+list1, ?list2)	ordena list1 por ordem crescente

keysort(+list1, ?list2)

construindo ?list2.

sendo list1 uma lista de pares,
(chave, valor), ordena-a por ordem
crescente de “chaves”.

\+ P

not(P)

P → Q ; R

if P then Q else R

P → Q

if P then Q else fail

repeat

gera uma sequência infinita de
“Backtrackings”; usado em ciclos.

call(:Term)

gera a execução do termo
instanciado, como se fosse a sua
literal substituição

listing

lista as cláusulas do actual
programa interpretado

listing(:spec)

cf. especificado

?- listing ([member/3, conc/3, reverse, go/2-3]).

var(?X)

testa se X está instanciada

nonvar(?X)

\neg var(?X)

ground(?X)

testa se X é um termo de base (ou
seja sem variáveis)

atom(?X)

X é átomo \neq número

float(?X)

integer(?X)

number(?X)

atomic(?X)

átomo ou número

functor(+Term, ?Nome, ?Aridade)

functor(?Term, +Nome, +Aridade)

+Term = ..?Lista

?Term = .. +Lista

Lista é uma lista cuja “cabeça” é o átomo correspondente ao principal functor de Term, e cuja “cauda” é a lista dos argumentos de Term.

?- prod(0, n, n-1) = .. L

L = [prod, 0, n, n-1]

?- n-1 = .. L

L = [-, n, 1]

?- OK = .. L

L = [OK]

name(+const, ?charList)

name(?const, +charList)

?- name(product, L).

L = [112, 114, 111, 100, 117, 99, 116]

?- name(:-), L).

L = [58,45]

?- name('1995', L).

L = [49,57,57,53]

?- name(X, [58,45]).

X = :-

?- name(X, [57,57,53]).

X = 995

assert(: cláusula)

asserta(: C)

assert(: cláusula, :ref)

assertz(:C)

adiciona a cláusula parâmetro à BC

retract(: cláusula)

retractall(:Head)

elimina a cláusula parâmetro da BC

setof(?Template, :Goal, ?Set)

bagof(?Template, :Goal, ?Bag)

findall(?Template, :Goal, ?Bag)

procura o armazenamento de soluções repetidas ou não!

EXEMPLOS DE UTILIZAÇÃO DOS DIVERSOS PREDICADOS PROLOG

Exº: **Cálculo Interactivo de X^2 .**

```
quad :-  
    write('Valor? :'),  
    read(V),  
    calcula_com(V).  
  
calcula_com(fim) :- !.  
  
calcula_com(X) :-  
    R is X*X,  
    write('Quadrado de:'), write (X), write( '=' ),  
    quad.
```

Nota:

- Este programa funciona; Usa recursividade;
- Ilustra a possibilidade de usar programas Prolog de forma interactiva;

Porém:

- Não obedece ao princípio da separação entre código computacional e código interactivo.
- Este princípio é muito importante para a modularidade e portabilidade dos programas, pelo que qualquer que seja o paradigma de programação usado deve ser respeitado.

Exº: **Programa que dada uma lista de listas apresenta cada lista-elemento numa linha separada.**

```
?- displaylistlist([[1, a, 7], [2 b], [x, 'a']]).
```

```
1      a      7
2      b
x      'a'
```

```
displaylistlist([ ]).
```

```
displaylistlist([L | LL] ) :- emlinha(L), nl, displaylistlist(LL).
```

```
emlinha([ ]).
```

```
emlinha([X | L]) :- write(X), tab(1), emlinha(L).
```

Exº: **Histograma (*) a partir de uma lista de inteiros ≤ 80 e ≥ 0 .**

```
histo([ ]).
```

```
histo([N | L]) :- aster(N), nl, histo(L).
```

```
aster(N) :- N>0, write (~*), N1 is N-1, aster(N1).
```

```
aster(N) :- N=< 0.
```

Exº: **Abrir um ficheiro, processar todos os seus elementos e devolver o controlo do input ao teclado.**

```
P :- read(F), see(F), processa(F), see(user)
```

```
processa(F):-
```

```
    read(Termo),
```

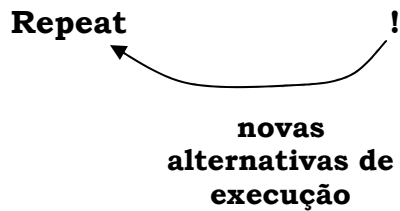
```
    procTermo(Termo).
```

```
procTermo(end_of_file) :- !.
```

```
procTermo(T):- trata(T),      % genérico
               processa(F).
```

Alternativa comum usando *repeat*

```
processa(F) :-
    seeing(Oldinput),
    see(F),
    {
        repeat,
            read(Termo),
            processaTermo(Termo),
            Termo == end_of_file,
        !,
        seen,
        see(Oldinput).
    }
```



Forma Geral do predicado repeat

```
repeat,
    gerar(Dado),
    processar(Dado),
    testar(Dado),
    !
```

Recursividade vs. Repeat ⇒ ✓ **eficiência em espaço**

Exº: **Cálculo de Quadrados com REPEAT**

quad:-

```
repeat,  
  read(X),  
  ( X = stop, !  
  ;  
  R is X * X,  
  write(C) , fail  
  ).
```

X = stop, !	⇒	não há “backtracking”
	⇒	não há repetição
write(R), fail	⇒	“backtracking”
	⇒	repeat

Registo de múltiplas soluções

setof (?Template, :Goal, ?SetSols)

SetSols = {*instância*.(?*Template*):

instância(?*Template*) : $\frac{\text{sat}}{\text{true}}$ }

Nota: SetSols é um conjunto \Rightarrow \nexists duplicados
SetSols é ordenada.

Exº:

?- setof(X, gosta(X, inglês), S).

S = [ada, ana, paulo]

?- setof(X, gosta(X, Y), S).

Y = 'Porto', S = [pedro, rui, zé]

Y = 'Braga', S = [rita, sara, toni]

?- setof(Y/S, setof(X, gosta(X, Y), S), F).

F = [?Porto?/ [pedro, rui, zé],

'Braga'/[rita, sara, toni].

?- setof(X, Y^(gosta(X, Y)), F).

F = [ada, ana, paulo, pedro, rita, rui,]

Nota: $Y^{(gosta(X, Y))} \Leftrightarrow \exists y.gosta(X, Y)$

Exº: **Seja a BC**

idade(pedro, 9).

idade(rita, 11).

idade(ana, 11).

idade(tiago, 7).

?- setof (C, idade(C, 11), L).

L = [ana, rita]

**?- setof (C, Idade^idade(C, Idade), Lc),
setof (I, Cri^idade(Cri, I), Li).**

Lc = [ana, pedro, rita, tiago]

Li = [7, 9, 11]

?- setof (I/C, idade(C, I), L).

L = [7/tiago, 9/pedro, 11/ [ana, rita]]

- **bagof(X, P, L)**

Semelhante a setof(X, P, L) excepto no facto de que admite duplicados e não invoca o predicado sort/2

- **findall(X, P, L)**

todos os X, independentemente de instanciações diferentes, segundo P não partilhadas com X.

?- findall (C, idade(C, I), L).

L = [ana, pedro, rita, tiago]

findall(X, P, L) :- % implementação caso não existisse !!

call(P), % procura 1 solução

assertz(queue(X)), % regista-a!

fail, % tenta outras

assertz(queue(bottom)), % regista fim de soluções

collect(L). % colecciona soluções


```

collect(L) :-
    retract(queue(X) !,           % remove próxima solução
    (X == bottom, !, L = [ ]      % fim das soluções?
    ;                               % tenta outras
    L = [X | R ], collect (R)).    % adiciona o resto

```

Nota:

- Cada solução imediatamente gerada é escrita na BC por forma a não se perder ao ser encontrada outra solução.
- Após todas as soluções serem geradas devem ser coleccionadas numa lista e removidas da base.
- Todas as soluções geram uma Fila.
A Fila é posteriormente consumida.
O fim da Fila deve ser marcado pelo átomo Bottom.
- O predicado *findall* funciona cf. a implementação apresentada

ANEXO

EXERCÍCIOS RESOLVIDOS

PARADIGMAS DE PROGRAMAÇÃO III

PROGRAMAÇÃO EM LÓGICA

1995/96

EXERCÍCIOS

Questão 1:

Defina um predicado *atomos* que dada uma lista e um átomo conte o número de ocorrências de tal átomo na lista parâmetro (Nota: o predicado Prolog *atom(X)* sucede se *X* é um átomo).

Questão 2:

Defina o predicado *lista(X)* que reconhece se *X* é uma lista.

Questão 3:

Defina um procedimento *separa(Lista, Pos, Neg)* que dada uma *Lista* de números a separe numa lista de positivos, *Pos*, e numa lista de negativos, *Neg*.

Questão 4:

Considere o seguinte programa Prolog abstracto:

p(1).

p(2) :- !.

p(3).

Indique quais as respostas obtidas para os seguintes "queries":

a) ?- *p*(*X*).

b) ?- *p*(*X*), *p*(*Y*).

c) ?- *p*(*X*), !, *p*(*Y*).

Questão 5:

Defina um predicado *entre(N1, N2, X)* que gere, usando backtracking, todos os inteiros X entre N1 e N2.

Questão 6:

Considere o seguinte programa Prolog abstracto:

```
q(a).
q(b).
q(c).
r(b, b1).
r(c, c1).
r(a, a1).
r(a, a2).
r(a, a3).
p(X, Y) :- q(X), r(X, Y).
p(d, d1).
p1(X; Y) :- q(X), r(X, Y), !.
p2(X, Y) :- q(X), !, r(X, Y).
p3(X, Y) :- !, q(X), r(X, Y).
p3(d, d1).
```

Apresente as árvores de programa correspondentes aos "queries" seguintes, marcando os ramos "cortados" usando o símbolo //.

- a) ?- p(X, Y).
- b) ?- p3(X, Y).

Questão 7:

Admitindo que existem na base de conhecimento factos sobre alunos com a estrutura apresentada em predicados da forma *aluno(Numero, Curso, Media)*,

a) Escreva um predicado Prolog que permita remover todos os factos *aluno* de uma base de conhecimento.

b) Escreva um outro predicado que remova apenas os alunos com média igual a 12.

Questão 8:

Escreva um procedimento que leia termos com a estrutura $p(X, Y)$ a partir de um ficheiro F, e que trate tais termos escrevendo em linhas separadas qual a soma de X com Y.

Questão 9:

Exprima numa expressão de lógica de 1ª ordem a semântica lógica (ou declarativa) do seguinte programa Prolog abstracto:

P :- X, !, Y.

P :- Z.

Questão 10:

Expresse em predicados Prolog as seguintes regras, baseadas em factos observáveis num dado contexto, designadamente, *ganha(X,Y)*.

a) Z é campeão se Z ganha mais vezes que todos os outros;

b) W é Bom se tem mais vitórias que derrotas.

```

/*-----*/
/*          PARADIGMAS DE PROGRAMAÇÃO III          */
/*          */
/*          (correção executável em SICSTUS Prolog)  */
/*          */
/*-----*/

/* QUESTÃO 1:                                     */
/*-----*/
/* As versões mais correctas não contam as possíveis instanciações de A */
/* */
/* com variáveis que possam ser introduzidas na lista parâmetro, dado */
/* que uma variável não é um átomo, cf. atom(variável) ==> no          */
/* Por exemplo, as invocações                                          */
/* */
/*      atomos(a, ["a", a, [1,2,3], Z, a, c, 12], X)                  */
/*      atomos1(a, ["a", a, [1,2,3], Z, a, c, 12], X)                 */
/* */
/* dão como resultado X = 2, o que é correcto. Outras definições podem, */
/* */
/* neste caso, gerar resultados errados (cf. atomos2, atomos3, etc.). */
/*-----*/

atomos(_, [], 0).
atomos(A, [H|T], N) :- atom(H), A = H, !,
                      atomos(A, T, N1),
                      N is N1 + 1.
atomos(A, [_|T], N) :- atomos(A, T, N).

/*-----*/

atomos1(_, [], 0).
atomos1(A, [A|T], N) :- !, atomos1(A, T, J), N is J + 1.
atomos1(A, [_|T], N) :- atomos1(A, T, N).

/*-----*/

atomos2(_, [], 0).
atomos2(A, [A|T], N) :- !, atomos2(A, T, N1), N is N1 + 1.
atomos2(A, [_|T], N) :- atomos2(A, T, N).

/*      atomos2(a, ["a", a, [1,2,3], Z, a, c, 12], X)                  */
/* */
/* dá como resultado X = 3, Z = a, o que é incorrecto.                */
/*-----*/

atomos3(_, [], 0).
atomos3(A, [H|T], N) :- A = H, atomos3(A, T, X), N is X + 1.
atomos3(A, [_|T], N) :- atomos3(A, T, N).

/* tal como o anterior                                             */
/* */
/*      atomos3(a, ["a", a, [1,2,3], Z, a, c, 12], X)                  */
/* */
/* dá, pelas mesmas razões, X = 3 e Z = a, o que é incorrecto.      */
/*-----*/

/* QUESTÃO 2: */
/* */

lista([]) :- !.                                     /* uma lista vazia , uma lista ! */
lista(_|_).                                         /* uma lista Head+Tail , lista   */

```

```

/* Exemplos: */
/* */
/* lista("a") ==> yes */
/* lista([a, "a", c]) ==> yes */
/* lista(123) ==> no */
/* lista([8/2]) ==> yes */
/* lista([a, X, 12, "a"]) ==> yes */
/* lista(a) ==> no */
/* lista(X) ==> X = [] ; no (caso não se usasse ! daria yes) */
/*-----*/
/* */
/* QUESTÃO 3: */
/* */
separa([], [], []).
separa([Num|Tl], [Num|Tp], Ln) :- Num >= 0, !, separa(Tl, Tp, Ln).
separa([Num|Tl], Lp, [Num|Tn]) :- separa(Tl, Lp, Tn).

/* Nota: O ! usado na 2ª cláusula é fundamental !! */
/* Experimentar sem este ! e verificar resultados. */
/*-----*/
/* */
/* QUESTÃO 4: */
/* */
/* a) ?- p(X). */
/* X = 1 ; X = 2 */
/* b) ?- p(X), p(Y). */
/* X = 1, Y = 1 ; X = 1, Y = 2 ; */
/* X = 2, Y = 1 ; X = 2, Y = 2 ; */
/* c) ?- p(X), !, p(Y). */
/* X = 1, Y = 1 ; X = 1, Y = 2 ; */
/*-----*/
/* */
/* QUESTÃO 5: */
/* */
entre(N1, N2, N1) :- N1 <= N2.
entre(N1, N2, X) :- N1 < N2, !, Y is N1 + 1, entre(Y, N2, X).
/* */
/* Nota: Uma definição do tipo entre1, apresentada a seguir, */
/* sem utilização do ! daria problemas */
/* */
entre1(N1, N2, N1) :- N1 <= N2.
entre1(N1, N2, X) :- N1 < N2, Y is N1 + 1, entre(Y, N2, X).

/*-----*/
/* */
/* QUESTÃO 6: */
/* */
/* Árvore de Programa para ?- p(X, Y). */
/* */

```

```

      p(X, Y)
     /  \
    /    \
   /      \
  /        \
 q(X), r(X, Y) ----- x = d, Y = d1
 /  \  \
X = a /  \ X = b X = c
   /  \  \
  r(a, Y) r(b, Y) r(c, Y)
 /  \  \ /  \  \
Y = a1 Y = a2 Y = a3 Y = b1 Y = c1
 /  \  \ /  \  \
yes  yes yes yes yes
/* Árvore de Programa para ?-p3(X, Y). */
/* */

```

```

      p3(X, Y)
     /  \
    /    \
   /      \
  /        \
 /          \
q(X), r(X, Y) -----
 /  \      |
X = a /    | X = b      X = c
   /  \    |   \
  r(a, Y) r(b, Y) r(c, Y)
   /  \    |   \
Y = a1 /    | Y = a2 \ Y = a3 Y = b1 Y = c1
   /  \    |   \
yes   yes  yes  yes  yes  yes  yes
*/
/*-----*/
/*
/* QUESTÃO 7:
/*
/* a) Soluções possíveis :
/*

    retractall(aluno(_,_,_)).

    abolish(aluno).

    remtodos :- retract(aluno(_,_,_)), fail.
    remtodos.

/* b) Soluções possíveis :
/*

    retractall(aluno(_,_,12)).

    reml2 :- retract(aluno(_,_,12)), fail.
    reml2.
/*-----*/
/*
/* QUESTÃO 8:
/*
/* Versão recursiva :
/*

ler(F) :- see(F), processar.
processar :- read(T), tratar(T).
tratar(end_of_file) :- seen.
tratar(p(X, Y)) :- S is X + Y, write(S), nl, !, processar.

/* Versão iterativa :
/*

lerl(F) :- see(F),
    repeat,
        read(T),
        (T == end_of_file, seen, !
        ;
        T = p(X, Y), S is X + Y, write(S), nl, fail).

/*-----*/
/*
/* QUESTÃO 9 :
/*
/* A semântica lógica do programa    P :- X, !, Y.
/*                                  P :- Z.      é a seguinte:
/*                                  P <=> (X e Y) ou (no X e Z).
/*                                  P <=> (X /\ Y) /\ (~X /\ Z).

```



```

/*-----*/
/*
/* QUESTÃO 10:
/*
/* A seguinte Base de Conhecimento , usada como exemplo
/*

ganha(a, b).
ganha(a, s).
ganha(b, c).
ganha(b, d).
ganha(a, d).
ganha(d, c).
ganha(c, f).
ganha(d, c). /* propositadamente duplicada. Não afecta resultados ! */
ganha(g, h).
ganha(g, c).
ganha(g, s).
ganha(g, d).

/* O predicado setof quando falha não devolve uma lista vazia, que daria
/* 0 em qualquer teste de comprimento, como seria de esperar. */
/* Daí a importância da 2ª cláusula em vitorias_de() e derrotas_de(). */
/* O ! introduzido em vitorias_de() e derrotas_de() é fundamental para */
/* o correcto funcionamento do programa. */

lista_venced(L) :- setof(Venc, Perd^ganha(Venc, Perd), L).

vitorias_de(X, N) :- setof(Op, ganha(X, Op), Lv), !, length(Lv, N).
vitorias_de(_, 0).

derrotas_de(X, N) :- setof(Op, ganha(Op, X), Lv), !, length(Lv, N).
derrotas_de(_, 0).

/* ----- cria uma lista de pares Vencedor/Nº de vitórias -----*/

criapares([], []).
criapares([V|Lv], [V/Nv | Lp]) :- vitorias_de(V, Nv), criapares(Lv, Lp).

/* ----- dada uma lista de pares Vencedor/Nº de vitórias -----*/
/* ----- dá o identificador do jogador com mais vitórias. -----*/
/* ----- Caso haja igualdade dá o primeiro que encontra. -----*/

ganha_mais([J/_], J) :- !.
ganha_mais([J1/N1 | Rp], J1) :- ganha_mais(Rp, J4), vitorias_de(J4, N),
                                N1 >= N, !.
ganha_mais([_ /N1 | Rp], J4) :- ganha_mais(Rp, J4), vitorias_de(J4, N),
                                N1 < N.

campeao(Z) :- lista_venced(L), criapares(L, Lp), ganha_mais(Lp, Z).

bom(Z) :- vitorias_de(Z, N1), derrotas_de(Z, N2), N1 > N2.

```

PARADIGMAS DE PROGRAMAÇÃO III

PROGRAMAÇÃO EM LÓGICA

1995/96

EXERCÍCIOS

Questão 1:

Defina um predicado *maiorPalList* que determine a maior palavra de uma lista de palavras.

Questão 2:

Defina um predicado *conjuntoOk* que analise se uma dada lista é uma representação válida de um conjunto matemático (i.e. não possui duplicados).

Questão 3:

Numa base de conhecimento registam-se factos sob a forma *invocou(utilizador, procedimento)* indicativos de que um dado utilizador de uma aplicação utilizou um dado procedimento.

Escreva de seguida predicados que lhe permitam responder aos seguintes "queries":

- a) *Qual o procedimento da aplicação mais vezes utilizado ?*
- b) *Qual o utilizador que mais procedimentos utilizou ?*
- c) *Qual o procedimento utilizado por um maior numero de utilizadores ?*
- d) *Determinar a lista de pares utilizador-procedimentos invocados.*

Questão 4:

Considere uma base de conhecimento contendo factos do tipo:

telefone(X, T) / o nº de telefone de casa da pessoa X é T */*

visita(X, Y) / a pessoa X está de visita à pessoa Y */*

$\text{emcasa}(X, Y)$ /* a pessoa X está em casa da pessoa Y */

$\text{contacto}(X, N)$ /* a pessoa X é contactável via o nº de telefone N */

Escreva um predicado $\text{encontra}(P)$ que permita determinar qual o número de telefone para contacto com a pessoa P, sabendo-se que :

- a) Se a pessoa P não está de visita a ninguém então estará em casa;
- b) Se a pessoa P está de visita a alguém é contactável através de um número de telefone que depende da pessoa visitada estar em casa ou de visita a alguém.

Questão 5:

Considere o seguinte programa Prolog abstracto:

```
q(a).
q(b).
r(b, b1).
r(c, c1).
r(a, a1).
r(a, a2).
p(X, Y) :- q(X), r(X, Y).
p(d, d1).
p1(X, Y) :- q(X), r(X, Y), !.
p2(X, Y) :- q(X), !, r(X, Y).
p3(X, Y) :- !, q(X), r(X, Y).
p3(d, d1).
```

Apresente as árvores de programa correspondentes aos "queries" seguintes, marcando os ramos "cortados" usando o símbolo //.

- a) $?- p1(X, Y).$
- b) $?- p3(X, a1).$

Questão 6:

Admitindo que existem numa base de conhecimento predicados sobre alunos com a estrutura *aluno(Numero, Curso, Media)*,

- a) Escreva um predicado que permita remover da base de conhecimento todos os factos *aluno* para alunos com um número compreendido entre os valores N1 e N2 (sendo $N1 \leq N2$).
- b) Escreva um outro predicado que remova os alunos de um dado Curso com média inferior a um valor M dado.
- c) Escreva um outro predicado que leia de um ficheiro F predicados *aluno* com a estrutura dada e insira na base de conhecimento todos os que pertencem a um dado Curso, tendo em atenção que não devem ser introduzidos duplicados.
- d) Escreva um predicado que determine o número total de alunos de um dado Curso com média superior a um valor dado, M, e introduza tal lista na base de conhecimento.

Questão 7:

Escreva um procedimento que leia termos com a estrutura *ponto(X, Y)* a partir de um ficheiro F, e que trate tais termos escrevendo num outro ficheiro as distâncias entre pontos consecutivos sob a forma de predicados *distancia(ponto1, ponto2, d)*.

Questão 8:

Expresse em predicados Prolog os seguintes "queries" sobre uma base de conhecimento com factos da forma *rua(nome, rua_inicio, rua_fim, lista_de_acessos)*.

- a) Lista das ruas cruzadas pela rua R;
- b) Lista de pares de ruas que se cruzam;

```

/*-----
*/
/* PARADIGMAS DE PROGRAMAÇÃO III */
/*
*/
/*                                CORRECÇÃO                                */
/*-----*/
/* 1.-                                */

maiorPalList([], "") :- !.
maiorPalList([P], P) :- !.
maiorPalList([P|Resto], P) :-
    length(P, N), maiorPalList(Resto, P1), length(P1, N1),
    N >= N1, !.
maiorPalList([P|Resto], P1) :-
    length(P, N), maiorPalList(Resto, P1), length(P1, N1), N1 > N.

/*----- OU TAMBEM -----*/

mPalList([], "") :- !.
mPalList([P], P) :- !.
mPalList([P1, P2 | Resto], MaxP) :-
    mPalList([P2|Resto], Max), maiorPal(P1, Max, MaxP).

maiorPal(P1, P2, P1) :- length(P1, N1), length(P2, N2), N1 >= N2, !.
maiorPal(_, P2, P2).

/*-----*/
/* 2.-                                */

conjOk([]).
conjOk([H|T]) :- \+ member(H, T), conjOk(T).

/*-----*/
/* 3.-                                */
*/

invocou(ze, inserir).
invocou(maria, init).
invocou(maria, inserir).
invocou(nuno, remover).
invocou(ana, remover).
invocou(rui, remover).
invocou(ze, consultar).
invocou(manel, inserir).
invocou(luis, inserir).

utilDeProc(Proc, Num) :- bagof(Proc, Util^invocou(Util, Proc), L), !,
length(L, Num).
utilDeProc(_, 0).

totalInv(Util, Num) :- bagof(Proc, invocou(Util, Proc), L), !, length(L,
Num).
totalInv(_, 0).

utilizadores(Lutil) :- setof(Util, Proc^invocou(Util, Proc), Lutil).

procedimentos(Lproc) :- setof(Proc, Util^invocou(Util, Proc), Lproc).

parProcNumUtil([], []).
parProcNumUtil([Proc|Lp], [Proc/Num|Resto]) :-
    utilDeProc(Proc, Num), parProcNumUtil(Lp, Resto).

parUtilProcs([], []).

```

```

parUtilProcs([Util|Lutil], [Util/Num|Resto]) :-
    totalInv(Util, Num), parUtilProcs(Lutil, Resto).

quemInvocaProc(Proc, Lu) :- setof(Ut, invocou(Ut, Proc), Lu).
quantosUsamProc(Proc, N) :- quemInvocaProc(Proc, L), length(L, N).

/* a) Qual o procedimento mais vezes invocado ? */

maisInvocado([P/_],P) :- !.
maisInvocado([P1/N1|Lp], P1) :- maisInvocado(Lp, P2),
                                utilDeProc(P2, N2), N1 >= N2, !.
maisInvocado([_/_|Lp], P2) :- maisInvocado(Lp, P2).

maisInv(P) :- procedimentos(Lp), parProcNumUtil(Lp, Lpares),
    maisInvocado(Lpares, P).

/* b) Qual o utilizador que mais procedimentos usou ? */

maisUsou([U/_],U) :- !.
maisUsou([U1/N1|Lp], U1) :- maisUsou(Lp, U2),
                             totalInv(U2, N2), N1 >= N2, !.
maisUsou([_/_|Lp], U2) :- maisUsou(Lp, U2).

queMaisUsou(Util) :- utilizadores(Lutil), parUtilProcs(Lutil, L),
    maisUsou(L, Util).

/* c) Procedimento invocado por mais utilizadores */

parProcNumUtil([], []).
parProcNumUtil([P|Lp], [P/Num|Lpnu]) :- quantosUsamProc(P, Num),
                                         parProcNumUtil(Lp, Lpnu).
usadoPorMais(P) :- procedimentos(Lp), parProcNumUtil(Lp, Lpnu),
    usadoPorMaisUtil(Lpnu, P).

usadoPorMaisUtil([P/_], P) :- !.
usadoPorMaisUtil([P/N1|Lpnu], P) :- usadoPorMaisUtil(Lpnu, P1),
                                     quantosUsamProc(P1, N2),
                                     N1 >= N2, !.
usadoPorMaisUtil([_/_|Lpnu], P1) :- usadoPorMaisUtil(Lpnu, P1).

/* d) Lista de pares Utilizador-Procedimentos invocados */

procInvocadosPor(Util, Lprocs) :- setof(Proc, invocou(Util, Proc),
    Lprocs).

parUtilProcInv([], []).
parUtilProcInv([U|Lu], [U/Lpinv| Resto]) :-
    procInvocadosPor(U, Lpinv), parUtilProcInv(Lu, Resto).

/*-----*/
/* 4.- */

emcasa(X, Z) :- visita(X, Y), emcasa(Y, Z).
contacto(X, Nt) :- emcasa(X, Y), telefone(Y, Nt), !.
contacto(X, Nt) :- telefone(X, Nt).

encontra(P) :- contacto(P, Nt), write('Telefonar para: '), write(Nt), nl.
encontra(_) :- write('Nao conheco !!').

/*      OU AINDA */

encontra(P, Nt) :- contacto(P, Nt).
encontra(_, ?).
/* 5.- */
*/

```

```

/*      DESENHO DAS ARVORES DE PROVA
*/
/*-----*/
/* 6.-                                         */

:-dynamic(aluno/3).

aluno(10, c, 12).
aluno(15, x, 11).
aluno(22, f, 19).
aluno(35, a, 13).
aluno(33, z, 12).
aluno(12, c, 14).
aluno(99, z, 15).
aluno(44, f, 13).

doCurso(C, L) :- setof(Num, Num^Md^aluno(Num, C,Md), L).
doCurso(_, []). /* caso a clausula anterior falhe ! */

/* a) Remover os alunos com números entre N1 e N2
*/

removeEntreNums(N1, N2) :-
    aluno(N, C, M), N >= N1, N <= N2, retract(aluno(N, C, M)), fail.
removeEntreNums(_, _).

/* b) Remover os alunos do curso Crs com média menor que Md
*/

remCursoMedia(Crs, Md) :-
    aluno(N, C, M), M <= Md, Crs = C, retract(aluno(N, C, M)), fail.
remCursoMedia(_, _).

/* c) Ler de ficheiro alunos de dado Curso e inserir na BC sem duplicados
*/

le_e_trata_alunos_de(F, C) :- see(F), le_e_trata_alunos(C), seen.

le_e_trata_alunos(Cr) :- doCurso(Cr, L),
    repeat,
        read(Aluno),
        ( Aluno == end_of_file, !
        ;
        Aluno = aluno(N, Cr, M),
        \+ member(N, L),
        assert(aluno(N,Cr,M)), fail).

/* d) Número total de alunos de dado Curso com média superior a M */

lstCursoMedia(C, M, Total) :- doCurso(C, L), melhoresQue(M, L, L1),
    length(L1, Total),
    assert(util_curso_media(C, M, L1)).

melhoresQue(_, [], []) :- !.
melhoresQue(M, [Num|T], [Num|Lr]) :- aluno(Num, _, Md), Md >= M,
    melhoresQue(M, T, Lr).

/*-----*/

/* 7.-                                         */
:- dynamic(ponto/2).
le_e_trata_Pontos(Fi, Fo) :- see(Fi), tell(Fo), le_Pontos_e_Calcula,

```

```

        seen, told.

/* le_Pontos_e_Calcula :- read(Ponto1), Ponto1 = ponto(X1, Y1),
/*                      ( Ponto1 == end_of_file, !
*/
/*                      ;
*/
/*                      repeat,
*/
/*                      read(Ponto2),
*/
/*                      ( Ponto2 == end_of_file, !
*/
/*                      ;
*/
/*                      Ponto2 = ponto(X2, Y2),
*/
/*                      dist(X1, X2, Y1, Y2, D), write(D), nl,
*/
/*                      -----> X1 is X2, Y1 is Y2, fail)).
*/

/* Embora aparentemente correcta esta soluç o n o funciona. As cl usulas
*/
/* X1 is X2 e Y1 is Y2, que permitiriam a permuta falham !!
*/
/* Raz o : Est -se a tentar usar uma atribui  o imperativa !!!
*/
/* Solu  o : assert e retract
*/

le_Pontos_e_Calcula :- read(Ponto1),
                      ( Ponto1 == end_of_file, !
                      ;
                      Ponto1 = ponto(X1, Y1),
                      assert(ponto(X1, Y1)),
                      repeat,
                      read(Ponto2),
                      ( Ponto2 == end_of_file, !
                      ;
                      Ponto2 = ponto(X2, Y2),
                      ponto(X, Y),
                      dist(X, X2, Y, Y2, D), write(D), nl,
                      retract(ponto(_, _)),
                      assert(ponto(X2, Y2)),
                      fail)).

dist(X1, X2, Y1, Y2, D) :- DifX is (X1 - X2), DifY is (Y1 - Y2),
                          QDifX is (DifX * DifX),
                          QDifY is (DifY * DifY),
                          D is sqrt(QDifX + QDifY).

/*-----*/
/* 8.-                                         */

rua(a, ia, fa, [x, y, z]).
rua(ia, iia, fia, [c, d, z]).
rua(fa, q, r, [x, w, c]).

cruzadas_Por(R, L) :- calc_cruzam(R), colect_cruzam(R, L).
calc_cruzam(R) :- rua(N, _, _, Lac), member(R, Lac), assert(cruza(R, N)),
                  fail.
calc_cruzam(_).

```



```

colect_cruzam(R, L) :- setof(N, cruza(R, N), L).
colect_cruzam(_, []).

fazTodososPares([], []) :- !.
fazTodososPares([R|Lr], [R/Lpr|Lp]) :- rua(R,_,_,Lc),
                                         fazParesporRua(R, Lc, Lpr),
                                         fazTodososPares(Lr, Lp).

fazParesporRua(_, [], []) :- !.
fazParesporRua(R, [H|T], [(R, H)| Lr]) :- fazParesporRua(R, T, Lr).

ruas(Lr) :- findall(Nr, rua(Nr, _,_,_), Lr).

listaPares(Lp) :- ruas(Lr), fazTodososPares(Lr, Lp).

criarL([], []).
criarL([H|T], [H/X|Tl]) :- rua(R,_,_,[X|Tl]),
                           rua(X,_,_,_), criarL(T, Tl).
criarL([_|T], [_|Tl]) :- criarL(T, Tl).
lpares(L) :- setof(X, rua(X, _,_,_), L1), criarL(L1, L).

```
