

Lógica para Programação

LEIC-Alameda

2022

Ana Paiva

Prolog
(S3-S4)

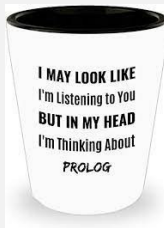
(estes slides são fortemente baseados nos slides gentilmente cedidos pelas Professoras Inês Lynce e Luísa Coheur, e qualquer gralha é da minha responsabilidade)

- ~~• Conceitos Básicos (Livro: 1.1)~~
- ~~• Lógica Proposicional – sistema dedutivo (2.1, 2.2.1, 2.2.2 e 2.2.4)~~
- ~~• Lógica Proposicional (ou Cálculo de Predicados) – resolução (3.1)~~
- ~~• Lógica de Primeira Ordem – sistema dedutivo (4.1, 4.2)~~
- ~~• Lógica de Primeira Ordem – resolução (5.1 e 5.2)~~
- ~~• Programação em Lógica (6)~~
- Prolog (7 + Apêndice A: manual de sobrevivência em Prolog)
- Lógica Proposicional (ou de Predicados) – sistema semântico (2.3, 2.4, 3.2)

- Programação orientada a objectos (e.g. C++, Java)
- Programação funcional (e.g. Scheme, Lisp)
- Programação em lógica (e.g. PROLOG)
- ...

- Introduzido nos anos 70 no âmbito do processamento de linguagem natural (1972, Alain Colmerauer et al.)

Colmerauer, A., Kanoui, H., Roussel, P. and Pasero, R. “Un système de communication hommemachine en français”, Groupe de Recherche en Intelligence Artificielle, Université d’Aix-Marseille. 1973.



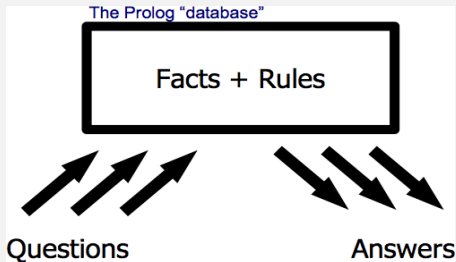
- Programa
 - $P(x, z) \leftarrow Q(x, y), P(y, z)$
 Prolog: $p(X, Z) :- q(X, Y), p(Y, Z).$
 - $P(x, x) \leftarrow$
 Prolog: $p(X, X).$
 - $Q(a, b) \leftarrow$
 Prolog: $q(a, b).$
- Objectivo
 - $\leftarrow P(x, b)$
 Prolog: $?- p(X, b).$

Função de selecção do Prolog (o primeiro literal do objectivo):

$$S_1(\leftarrow \alpha_1, \dots, \alpha_n) = \alpha_1$$

- Regra de procura do Prolog: começa a procurar cláusulas para unificar de cima para baixo

Programação em lógica: um programa responde a perguntas



- Programa: conjunto de cláusulas determinadas
- Afirmações
- Regras
- Objectivos

Exemplos Iniciais : programa (factos e regras)

`f(a).`
`f(b).`
`f(c).`
`g(b).`
`g(c).`
`r(d).`

Factos

`r(X) :- f(X), g(X).` ← Regras

Programa

`?- r(X).` ← Objetivo

- O Prolog segue estratégia de procura em profundidade com retrocesso



Pode até nunca encontrar solução
(mesmo que ela exista)... caso
siga por um caminho infinito

Termos: Um termo pode ser uma constante, uma variável ou um termo composto (correspondendo à aplicação de uma função ao número apropriado de argumentos). Em BNF é:

$$\langle \text{termo} \rangle ::= \langle \text{constante} \rangle \mid \langle \text{variável} \rangle \mid \langle \text{termo composto} \rangle$$

Constantes: Uma constante pode ser um átomo ou um número.

$$\langle \text{constante} \rangle ::= \langle \text{átomo} \rangle \mid \langle \text{número} \rangle$$

- Nota: um átomo é qualquer sequência de caracteres que comece com uma letra minúscula, qualquer cadeia de caracteres (delimitados por plicas) ou um conjunto de símbolos (átomos especiais).

Constantes: Uma constante pode ser um átomo ou um número.

`<constante> ::= <átomo> | <número>`

Um **átomo** é qualquer sequência de caracteres que podem incluir letras, dígitos e (underscore) e que têm de ser iniciadas por letra minúscula, ou qualquer cadeia de caracteres (delimitados por plicas), ou um conjunto de símbolos (átomos especiais).

Exemplos átomos:

ana, nuno, ana silva, x_25, 'ana', 'Ana Silva'

Átomos especiais

`[]{}!; . . .`

Números podem ser Inteiros e reais

1, -97, 3.1415, -0.0035

- Cadeias de caracteres que podem incluir letras, dígitos e _
- Têm de ser iniciadas por letra **Maiúscula** ou _

Ex: X, Resultado, Lista_participantes, _x23

A variável _ representa uma variável sem nome.

- O domínio de uma variável corresponde a uma única cláusula
- Atenção: chamam-se variáveis **singleton** às que aparecem uma única vez numa cláusula; devem ser substituídas por _

Termos compostos: um termo composto corresponde à aplicação de uma letra de função (em Prolog designada por functor) ao numero apropriado de argumentos.

Permitem criar objectos que agrupam um conjunto de objectos:
– `tipo-de-objecto(componente-1, componente-2, ...)`

Exemplos

```
data(Dia, Mes, Ano).  
    data(14, novembro, 2007).  
    data(25, dezembro, 2000).
```

Permite ter:

```
dataDeNasc(fern_pessoa, data(13, junho, 1888)).
```

Um literal corresponde à **aplicação de um predicado ao número apropriado de termos.**

Exemplos

```
ad(rui, X)
ad(joão, manuel)
ant(rui, luísa)
urso(winnie)
```

Ou:

```
pai(joao, pai(joão)).
```

```
?- pai(joao,X).
X = pai(joão)
```

Sim!



Mas então podemos ter predicados com o mesmo nome e um número diferente de argumentos?

Programa: factos + regras

Factos

- correspondem a afirmações;
- correspondem a cláusulas de Horn com um literal positivo;
- consistem em letras de predicado (iniciados com minúscula) com 1 ou + argumentos e terminados com um ponto final.

Exemplos

`mulher(luísia).`

`frequenta(rui, lp).`



Então e como é que
se
programa???

Programa: factos + regras

Regras

- permitem realizar inferência;
- correspondem a cláusulas de Horn com um literal positivo e pelo menos um literal negativo;
- são usados os símbolos “:-”, “,”, “;” e terminam com um ponto final.

A interpretação destes símbolos é a seguinte:

- O operador “:-” deve ser interpretado como “se”;
- O “,” deve ser interpretada como “e” (e o “;” como “ou”).

Exemplo

% X 'é avô de Z se X for ascendente directo de Y e Y for ascendente directo de Z

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



Então e como é
que se
programa???

Como executar um programa???

Executar um programa implica colocar questões (correspondem a objectivos).

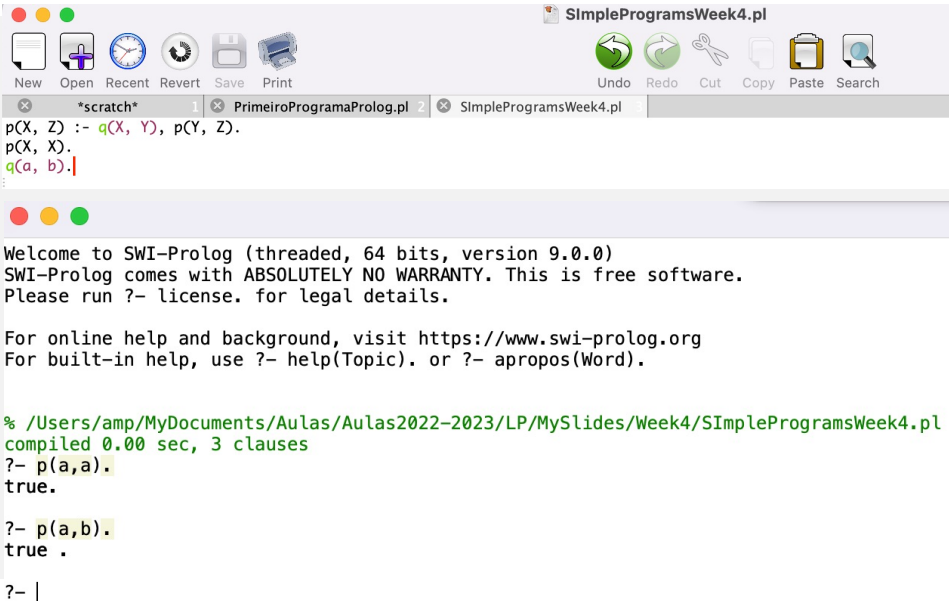
Objectivos são cláusulas de Horn que têm somente literais negativos.

Exemplo

`:- avo(pedro,nuno).`

- Na janela de interação
`?- avo(pedro,nuno).`

Como executar um programa???



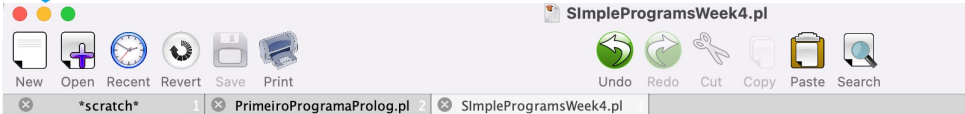
The screenshot shows a Prolog IDE window titled "SimpleProgramsWeek4.pl". The menu bar includes "New", "Open", "Recent", "Revert", "Save", "Print", "Undo", "Redo", "Cut", "Copy", "Paste", and "Search". The tab bar shows three tabs: "*scratch*", "PrimeiroProgramaProlog.pl", and "SimpleProgramsWeek4.pl". The editor contains the following Prolog code:

```
p(X, Z) :- q(X, Y), p(Y, Z).  
p(X, X).  
q(a, b).|
```

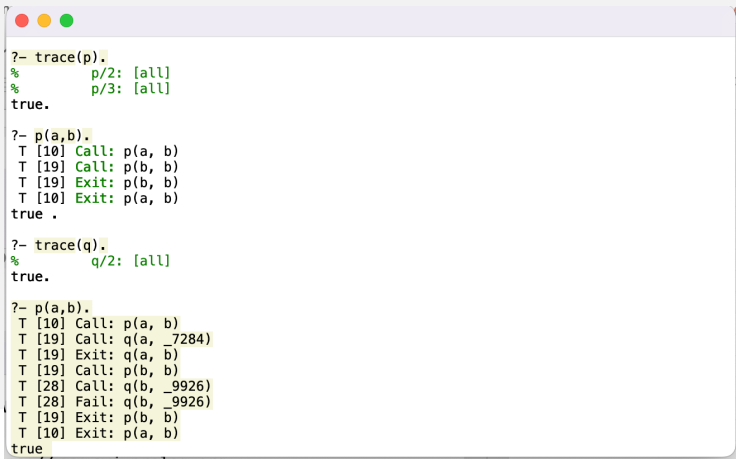
Below the editor, the SWI-Prolog console displays the following output:

```
Welcome to SWI-Prolog (threaded, 64 bits, version 9.0.0)  
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.  
Please run ?- license. for legal details.  
  
For online help and background, visit https://www.swi-prolog.org  
For built-in help, use ?- help(Topic). or ?- apropos(Word).  
  
% /Users/amp/MyDocuments/Aulas/Aulas2022-2023/LP/MySlides/Week4/SimpleProgramsWeek4.pl  
compiled 0.00 sec, 3 clauses  
?- p(a,a).  
true.  
  
?- p(a,b).  
true .  
  
?- |
```

Como executar um programa???



```
p(X, Z) :- q(X, Y), p(Y, Z).
p(X, X).
q(a, b).|
```



Exemplo básico de interacção em Prolog (cont.)

Tipicamente programas são escritos num ficheiro (.pl)

?- [<nome-programa>]. /*carrega o programa*/

Objectivos também podem ser incluídos no ficheiro antecedidos por :-

Objectivos com várias respostas: <Enter> aceita uma resposta,

“;” pede a resposta seguinte



```
Welcome to SWI-Prolog (threaded, 64 bits, version 9.0.0)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.
```

```
For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).
```

```
% /Users/amp/MyDocuments/Aulas/Aulas2022-2023/LP/MySlides/Week4/SimpleProgramsWeek4.pl
compiled 0.00 sec, 3 clauses
```

```
?- p(a,a).
true.
```

```
?- p(a,b).
true .
```

```
?- |
```

% Example of the previous classes

```
ant(X,Y):- ad(X,Y).
```

```
ant(X, Z):- ant(X,Y), ad(Y,Z).
```

```
ad(pedro, luisa).
```

```
ad(rui, pedro).
```

```
ad(rui, luísa).
```

```
Welcome to SWI-Prolog (threaded, 64 bits, version 9.0.0)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.
```

```
For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).
```

```
% /Users/amp/MyDocuments/Aulas/Aulas2022-2023/LP/MySlides/Week4/SimpleProgramsWeek4.pl
compiled 0.00 sec, 8 clauses
```

```
?- ant(rui,luisa).
true .
```

```
?- ant(X,luisa).
X = pedro ;
X = rui .
```

```
?-
```

% Example of the previous classes

```
ant(X,Y):- ad(X,Y).
```

```
ant(X, Z):- ant(X,Y), ad(Y,Z).
```

```
ad(pedro, luisa).
```

```
ad(rui, pedro).
```

```
ad(rui, luisa).
```

For online help and background, visit <https://www.swi-prolog.org>
For built-in help, use `?- help(Topic).` or `?- apropos(Word).`

```
% /Users/amp/MyDocuments/Aulas/Aulas2022-2023/LP/MySlides/Week4/SIMpleProgramsWeek4.pl  
compiled 0.00 sec, 8 clauses
```

```
?- ant(rui,luisa).  
true .
```

```
?- ant(X,luisa).  
X = pedro ;  
X = rui .
```

```
?- ad(_,pedro).  
true.
```

```
?- ant(X,pedro).  
X = rui ;
```

```
ERROR: Stack limit (1.0Gb) exceeded
```

```
ERROR: Stack sizes: local: 0.9Gb, global: 84.2Mb, trail: 0Kb
```

```
ERROR: Stack depth: 11,029,769, last-call: 0%, Choice points: 5
```


```
ERROR: Probable infinite recursion (cycle):
```

```
ERROR: [11,029,768] user:ant(rui, _22076390)
```

```
ERROR: [11,029,767] user:ant(rui, _22076410)
```

```
?- |
```

Unificação e Manipulação de Dados

- Unificação 
- Comparação
- Reconhecimento

Predicado de unificação : =

$\langle t1 \rangle = \langle t2 \rangle$

tem sucesso se os termos $\langle t1 \rangle$ e $\langle t2 \rangle$
podem ser unificados;

se sim são feitas as substituições

Negação do predicado de unificação: \=

$\langle t1 \rangle \neq \langle t2 \rangle$

tem sucesso se $\langle t1 \rangle$ e $\langle t2 \rangle$ não podem ser
unificados

?- a = b.

false

?- f(X, a) = f(b, Y).

X = b, Y = a

?- X = a.

X = a

?- X = X.

true

?- X = Y.

X = Y

?- f(X, a) = f(b, X).

false

?- xpto(rui, ana) = xpto(X, Y).

X = rui, Y = ana

?- f(_ , X, _) = f(a, b, c).

X = b

- Uma vez unificadas, é uma relação para sempre!

```
?- X=1.
```

```
X = 1.
```

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

```
X = 2.
```

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

```
false.
```

```
?- X=1.
```

```
X = 1.
```

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


```
X = 2.
```

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

```
false.
```

```
?-
```

Manipulação de Dados

- Unificação
- Comparação 
- Reconhecimento

Operadores de comparação

`==`

`\==`

testam se dois termos são iguais
(diferentes)

NOTA: a comparação não instancia
as variáveis
(e não avalia numericamente)

?- 4 == 1 + 3.

false

?- a == a.

true

?- a == 'a'.

true

?- a == b.

false

?- X == Y.

false

?- X == a.

false

?- X = a, X == a.

X = a

?- X == a, X = a.

false

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

X = a, Y = a

Operadores de comparação

@>

@<

tem sucesso se <n1> considerado como cadeia de caracteres aparece depois (ou antes) de <n2>

```
?- a @> b.
```

```
false.
```

```
?- b @> a.
```

```
true.
```

```
?- aa @> a.
```

```
true.
```

```
?- a @> aa.
```

```
false.
```

Predicado	Utilização	Significado
==	$\langle t1 \rangle == \langle t2 \rangle$	Predicado de identidade: tem sucesso apenas se os termos $\langle t1 \rangle$ e $\langle t2 \rangle$ são idênticos.
\==	$\langle t1 \rangle \backslash == \langle t2 \rangle$	Negação do Predicado de identidade: tem sucesso apenas se os termos $\langle t1 \rangle$ e $\langle t2 \rangle$ não são idênticos.
@>	$\langle n1 \rangle @> \langle n2 \rangle$	Predicado maior: tem sucesso se $\langle n1 \rangle$ considerado como cadeia de caracteres aparece depois de $\langle n2 \rangle$
@<	$\langle n1 \rangle @< \langle n2 \rangle$	Predicado menor: tem sucesso se $\langle n1 \rangle$ considerado como cadeia de caracteres aparece antes de $\langle n2 \rangle$
@>=	$\langle n1 \rangle @>= \langle n2 \rangle$	Predicado maior ou igual: tem sucesso se $\langle n1 \rangle$ considerado como cadeia de caracteres é igual ou aparece depois de $\langle n2 \rangle$
@<=	$\langle n1 \rangle @<= \langle n2 \rangle$	Predicado menor ou igual: tem sucesso se $\langle n1 \rangle$ considerado como cadeia de caracteres é igual ou aparece antes de $\langle n2 \rangle$

Exemplo

Igualdade aritmética: `==`

`<n1> == <n2>`

tem sucesso se `<n1>` e `<n2>` são o mesmo número (avalia numericamente);

Negação do predicado da igualdade aritmética: `= \ =`

`<n1> = \ = <n2>`

tem sucesso se `<n1>` e `<n2>` não são o mesmo número;

```
?- 4 == 1 + 3.
```

```
false.
```

```
?- 4 == 1 + 3.
```

```
true.
```

```
?- X = 4, X == 3 + 1.
```

```
false.
```

```
?- X = 4, X == 3 + 1.
```

```
X = 4.
```

```
?- X = 4, 3 + 1 == X.
```

```
X = 4.
```

```
?- X = 4, 3 + 1 == X.
```

```
false.
```

```
?- X.
```

```
?- X.
```

```
% ... 1,000,000 ..... 10,000,000 years later
```

```
%
```

```
%      >> 42 << (last release gives the question)
```

```
-
```

Unificação de termos (= versus is)

Exemplo

Predicado de unificação : =

$\langle t1 \rangle = \langle t2 \rangle$

tem sucesso se os termos $\langle t1 \rangle$ e $\langle t2 \rangle$
podem ser unificados;

se sim são feitas as substituições

MAS: **mas não faz avaliação numérica!**

Predicado de unificação : is

$\langle t1 \rangle \text{is } \langle t2 \rangle$

“is” avalia numericamente a expressão do
lado direito e unifica com a do lado
esquerdo

?- A = 1+2, A == 3.
false

?- A is 1+2, A == 3.
A = 3

Operação	Utilização	Significado
$+$ $-$ $*$ $/$	$\langle t1 \rangle + \langle t2 \rangle$ $\langle t1 \rangle - \langle t2 \rangle$ $\langle t1 \rangle * \langle t2 \rangle$ $\langle t1 \rangle / \langle t2 \rangle$	Soma, subtração, multiplicação e divisão entre $\langle t1 \rangle$ e $\langle t2 \rangle$ Também podem ser escritos: $+(\langle t1 \rangle, \langle t2 \rangle)$, etc
$**$	$\langle t1 \rangle ** \langle t2 \rangle$	Potencia com base $\langle t1 \rangle$ e expoente $\langle t2 \rangle$
$//$	$\langle t1 \rangle // \langle t2 \rangle$	Divisão inteira entre $\langle t1 \rangle$ e $\langle t2 \rangle$
mod	$\langle t1 \rangle \text{ mod } \langle t2 \rangle$	Resto da divisão inteira entre $\langle t1 \rangle$ e $\langle t2 \rangle$
round	round ($\langle t2 \rangle$)	Inteiro mais proximo
sqrt	sqrt ($\langle t2 \rangle$)	Raíz quadrada

Com utilização de:

, para a conjunção e

; para disjunção

Podemos usar , e ; entre expressões aritméticas...

Exemplo

?- X is 3+4.

X = 7

?- 3 ::= 4.

false.

?- (6>5, 3 ::= 4, X is 3).

false.

?- (6>5, 3 ::= 4, X is 3; X is 8).

X = 8

?- (6>5, 3 \= 4, X is 3; X is 8).

X = 3

```
?- 2 + 3 = +(2, 3).
true.
?- 2 + 3 = +(3, 2).
false.
?- X = +(2, 3).
X = 2+3.
?- 2 + X = Y + 3.
X = 3,
Y = 2.
?- 5 < 7.
true.
?- 3 + 5 > 12.
false.
?- 3 + 5 >= +(4, +(2, 2)).
true.
?- X > 12.
```

ERROR: >/2: Arguments are not sufficiently
instantiated

Nota: Predicado de unificação : =

$\langle t1 \rangle = \langle t2 \rangle$

tem sucesso se os termos $\langle t1 \rangle$ e $\langle t2 \rangle$
podem ser unificados;

se sim são feitas as substituições

MAS: *mas não faz avaliação numérica!*

Os nossos melhores amigos em Prolog: Listas

Lista = sequência de elementos limitados por parêntesis rectos

Exemplos:

- [1, . . . , -5]
- [ola, 2, ola(2, 4), X]
- [1, ola(2, 4), [ola, 2 , Z], Y]

Importante:

[] representa a lista vazia

O símbolo | permite referir o(s) elementos à cabeça e na cauda da lista (que é uma lista)
(ex: [Cabeça | Cauda])

Listas

Lista = sequência de elementos limitados por parêntesis rectos

Importante:

[] representa a lista vazia

O símbolo | permite referir o(s) elementos à cabeça e na cauda da lista (que é uma lista)

(ex: [Cabeça | Cauda])

```
?- [a,b] = [X,Y].
```

```
X = a,
```

```
Y = b.
```

```
?- [a,b] = [X|Y].
```

```
X = a,
```

```
Y = [b].
```

```
?- is_list(a).
```

```
false.
```

```
?- is_list([a,b]).
```

```
True.
```

Exemplo: a lista [ca, hf, vn] unifica com?

[ca, hf, vn]

- [Cabeça | Cauda]
Cabeça = ca, Cauda = [hf, vn]
- [ca | Cauda]
Cauda = [hf, vn]
- [E11, E12 | Cauda]
E11 = ca, E12 = hf, Cauda = [vn]
- [ca, hf | Cauda]
Cauda = [vn]
- [ca, hf, vn | []]
- ...

Escrever um programa que determina se um dado elemento pertence a uma lista.

Quer dizer-> Definir o predicado membro/2 (com aridade 2), em que o literal membro(E,L) afirma que E pertence à lista L.

membro(X, [X|_]). ← 1. Caso terminal

membro(X, [_|R]) :- membro(X,R). ←

?- membro(1,[1,2,3,4,5]). 2.Parte recursiva
true.

?- membro(6,[]).

false.

?- membro(6,[1,2,3,4,5]).

false.

?- membro(3,[1,2,3,4,5]).

True

?- membro(X,[1,2,3,4,5]).

X = 1 ;

X = 2 ;

X = 3 ;

X = 4 ;

X = 5 ;

false.

Exemplo: último de uma lista (o primeiro conseguimos sempre obter com o |)

Escreva um predicado que permite obter o último de uma lista.

```
ultimo([X], X).  
ultimo([_|Cauda], X) :- ultimo(Cauda, X).
```

```
?- ultimo([a,b,f,d,s,r],X).  
X = r  
Unknown action: , (h for help)  
Action? .
```

```
?- ultimo([3,f(a),[z,l]],X).  
X = [z, l] ;  
false.
```

```
?- ultimo([3|[4,5,6]],X).  
X = 6 .
```

```
?-
```


Escrever um programa
que junta duas listas

Quer dizer-> Definir o
predicado junta/3 (com
aridade 3), em que o
literal junta(L1,L2,LF)
afirma que LF é a junção
de L1 e L2.

```
junta([], L,L).
junta([P|R],L1, [P|L2]) :- junta(R,L1,L2).
```

1. Caso terminal

2. Parte recursiva

```
?- junta(a,b,X).
false.
?- junta([a,b,c],[d,e],X).
X = [a, b, c, d, e].
?- junta([a,b,c], Y, [a, b, c, d, e]).
Y = [d, e].
?- junta([],a,X).
X = a.
?- junta(a,[],X).
false.
?- junta([a|[g,p]],[],X).
X = [a, g, p].
```

comprimento([], 0). 1. Caso terminal
 comprimento([_|R], C+1) :- comprimento(R, C).

2. Parte recursiva

Escrever um programa
 que determina o
 comprimento de uma lista

Quer dizer -> Definir o
 predicado comprimento/2
 (com aridade 2), em que o
 literal comprimento(L, C)
 afirma que C é o
 comprimento de L

?- comprimento([], X).

X = 0.

?- comprimento([1,2,3,4], X).

X = 0+1+1+1+1.

```
comprimento1([], 0).  
comprimento1(_|R,Z) :- comprimento1(R,C),  
                        Z is C+1.
```

```
?- comprimento1([],X).
```

```
X = 0.
```

```
?- comprimento1([1,2,3,4],X).
```

```
X = 4.
```

```
comprimento1([], 0).
```

```
comprimento1(_|R,Z) :- comprimento1(R,C),  
                        Z is C+1.
```

```
% /Users/amp/MyDocuments/Aulas/Aulas2022-2023/LP/MySlides/Week4/SimpleProgramsWeek4.pl  
compiled 0.00 sec, 2 clauses  
?- comprimento([1,2,3,4,5,6],X).  
X = 0+1+1+1+1+1+1.
```

```
?- trace(comprimento1).  
%      comprimento1/2: [all]  
true.
```

```
?- comprimento([1,2,3,4,5,6],X).  
T [10] Call: comprimento([1, 2, 3, 4, 5, 6], _28010)  
T [19] Call: comprimento([2, 3, 4, 5, 6], _29352)  
T [28] Call: comprimento([3, 4, 5, 6], _30286)  
T [37] Call: comprimento([4, 5, 6], _31220)  
T [46] Call: comprimento([5, 6], _32154)  
T [55] Call: comprimento([6], _33088)  
T [64] Call: comprimento([], _34022)  
T [64] Exit: comprimento([], 0)  
T [55] Exit: comprimento([6], 1)  
T [46] Exit: comprimento([5, 6], 2)  
T [37] Exit: comprimento([4, 5, 6], 3)  
T [28] Exit: comprimento([3, 4, 5, 6], 4)  
T [19] Exit: comprimento([2, 3, 4, 5, 6], 5)  
T [10] Exit: comprimento([1, 2, 3, 4, 5, 6], 6)  
X = 6.
```

Versão interativa

```
comprimento3(L,C):- comprimento_i(L,0,C).
comprimento_i([ ], Ac, Ac).
comprimento_i([_ | R],Ac,C):- Ac_N is Ac + 1,
                               comprimento_i(R,Ac_N,C).
```

```
T [10] Exit: comprimento3([1, 2, 3, 4, 5, 6], 6)
X = 6.

?- trace(comprimento_i).
%      comprimento_i/3: [all]
true.

?- comprimento3([1,2,3,4,5,6],X).
T [10] Call: comprimento3([1, 2, 3, 4, 5, 6], _10012)
T [19] Call: comprimento_i([1, 2, 3, 4, 5, 6], 0, _10012)
T [28] Call: comprimento_i([2, 3, 4, 5, 6], 1, _10012)
T [37] Call: comprimento_i([3, 4, 5, 6], 2, _10012)
T [46] Call: comprimento_i([4, 5, 6], 3, _10012)
T [55] Call: comprimento_i([5, 6], 4, _10012)
T [64] Call: comprimento_i([6], 5, _20)
T [73] Call: comprimento_i([], 6, _20)
T [73] Exit: comprimento_i([], 6, 6)
T [64] Exit: comprimento_i([6], 5, 6)
T [55] Exit: comprimento_i([5, 6], 4, 6)
T [46] Exit: comprimento_i([4, 5, 6], 3, 6)
T [37] Exit: comprimento_i([3, 4, 5, 6], 2, 6)
T [28] Exit: comprimento_i([2, 3, 4, 5, 6], 1, 6)
T [19] Exit: comprimento_i([1, 2, 3, 4, 5, 6], 0, 6)
T [10] Exit: comprimento3([1, 2, 3, 4, 5, 6], 6)
X = 6.
```

?-

Considere o predicato `escolhe/3` tal que o literal `escolhe(L1,E,L2)` afirma que `L2` é a lista que se obtém de `L1`, retirando-lhe o elemento `E`.

```
escolhe([P | R], P, R).
```

```
escolhe([P|R],E, [P|S]) :- escolhe(R,E,S).
```

```
?- escolhe([1,2,3,4,5,6],3, X).
```

```
X = [1, 2, 4, 5, 6] .
```

```
?- escolhe([1,2,3,4,5,6], 8,X).
```

```
false.
```

```
?- escolhe([1,2,3,4,5,6,1], 1,X).
```

```
X = [2, 3, 4, 5, 6, 1] ;
```

```
X = [1, 2, 3, 4, 5, 6] .
```

Considere o predicato `parte/4` tal que recebe uma lista e um inteiro, e que os divide de forma a que todos os elementos menores que o inteiro ficam numa lista e os outros noutra lista.

```
parte([ ], N, [ ], [ ]).  
parte([P|R],N, [P|R1],L2) :- P< N,  
                             parte(R,N,R1, L2).  
parte([P|R],N, L1, [P|R2]) :- P>= N,  
                             parte(R,N,L1, R2).  
  
?- parte([1,2,3,4,5,6],3, L1, L2).  
L1 = [1, 2],  
L2 = [3, 4, 5, 6]
```

```
parte([ ], N, [ ], [ ]).
```

```
parte([P|R],N, [P|R1],L2) :- P< N,  
                             parte(R,N,R1, L2).
```

```
parte([P|R],N, L1, [P|R2]) :- P>= N,  
                             parte(R,N,L1, R2).
```

```
?- parte([1,2,3,4,5,6],3, L1, L2).
```

```
T [10] Call: parte([1, 2, 3, 4, 5, 6], 3, _1174, _1176)
```

```
T [19] Call: parte([2, 3, 4, 5, 6], 3, _2570, _1176)
```

```
T [28] Call: parte([3, 4, 5, 6], 3, _3520, _1176)
```

```
T [37] Call: parte([4, 5, 6], 3, _3520, _4470)
```

```
T [46] Call: parte([5, 6], 3, _3520, _5420)
```

```
T [55] Call: parte([6], 3, _3520, _6370)
```

```
T [64] Call: parte([], 3, _3520, _7320)
```

```
T [64] Exit: parte([], 3, [], [])
```

```
T [55] Exit: parte([6], 3, [], [6])
```

```
T [46] Exit: parte([5, 6], 3, [], [5, 6])
```

```
T [37] Exit: parte([4, 5, 6], 3, [], [4, 5, 6])
```

```
T [28] Exit: parte([3, 4, 5, 6], 3, [], [3, 4, 5, 6])
```

```
T [19] Exit: parte([2, 3, 4, 5, 6], 3, [2], [3, 4, 5, 6])
```

```
T [10] Exit: parte([1, 2, 3, 4, 5, 6], 3, [1, 2], [3, 4, 5, 6])
```

```
L1 = [1, 2],
```

```
L2 = [3, 4, 5, 6]
```



```
?- read(X).  
|: a.  
X = a.  
?- read(b).  
|: a.  
false.  
?- X = b, read(X).  
|: a.  
false.  
?- read(X).  
|: 3 + 2.  
X = 3+2.  
? - read(X).  
|: 3 mais 2.  
ERROR: Stream user input:0:113 Syntax error:  
Operator expected
```

```
?- write(a), write(b).  
ab  
true.  
?- writeln(a), write(b). a  
b  
true.  
?- write(+(2,3)).  
2+3  
true.
```



Se podemos ter predicados com o mesmo nome e um número diferente de argumentos, como é que funciona?

É possível fazer overloading de predicados

- O mesmo predicado pode ter um número de argumentos diferente!

Polimodalidade

Capacidade de utilizar múltiplos modos de interacção com um programa (diferentes argumentos instanciados).

```
soma_5_e_duplica(X, Y) :- Y is 2 * (X + 5).
```

```
?- soma_5_e_duplica(10, Y).
```

```
Y = 30.
```

```
?- soma_5_e_duplica(10, 30).
```

```
true.
```

```
?- soma_5_e_duplica(X, 30).
```

```
ERROR: is/2: Arguments are not sufficiently  
instantiated
```

Semântica declarativa: aquilo que um programa afirma.

Assim, $a(X) :- b(X), c(X).$

pode ser lida como “se $b(X)$ e se $c(X)$ se verificam para uma dada substituição para a variável X , então podemos concluir que $a(X)$ também se verifica para essa substituição”.

Semântica procedimental: como provar um objectivo com um determinado programa.

O exemplo anterior $a(X) :- b(X), c(X).$

pode ser lido como “para provar $a(X)$ temos de provar uma instância de $b(X)$ e depois $c(X)$, com as substituições adequadas de X ”.

- Programa: **sequência** de cláusulas determinadas
- Execução (prova) de um objectivo: **refutação** SLD cuja função de selecção devolve o primeiro literal e a **regra de procura** escolhe a primeira cláusula unificável

Relembrar...Call, redo e amigos



Figura A.3: Eventos associados à prova do objetivo obj.

- **Call**. Este evento ocorre no momento em que o `PROLOG` inicia a prova de um objetivo. Este evento corresponde à passagem por um nó da árvore SLD no sentido de cima para baixo. O objetivo indicado corresponde ao primeiro literal do objetivo em consideração.
- **Exit**. Este evento ocorre no momento em que o `PROLOG` consegue provar o objetivo, ou seja, quando o objetivo tem sucesso.
- **Redo**. Este evento ocorre no momento em que o `PROLOG`, na sequência de um retrocesso, volta a considerar a prova de um objetivo.
- **Fail**. Este evento ocorre no momento em que o `PROLOG` falha na prova de um objetivo.