

Programação III

Paradigmas de Programação Avançados

Salvador Abreu, Universidade de Évora, 2022/23

Representação de Informação

Valores (dados): termos

Qualquer valor é expresso como um **termo**, que pode tomar várias formas:

- Um **constante**, i.e. um literal. Pode ser
 - Um **número** (inteiro ou em vírgula flutuante)
 - Um **átomo** (um string que é hashado e representado de forma muito compacta)
 - Um átomo é expresso como um string entre ‘plicas’
 - Se não tiver caracteres estranhos (p/ex espaços) não precisa das plicas
 - Não deve começar por uma maiúscula
- Um **termo composto**, i.e. um termo com
 - Uma “etiqueta”, ou cabeça, que é um átomo
 - Um ou mais **subtermos**, indexados (i.e. na posição 1, 2, ... N)
 - Diz-se que um termo composto tem **aridade** N se tiver N subtermos
- Uma **variável** (livre)
 - Sintacticamente é como um “identificador”
 - Primeiro character uma **maiúscula** ou um “_”

Representação de Informação

Exemplos de termos numéricos

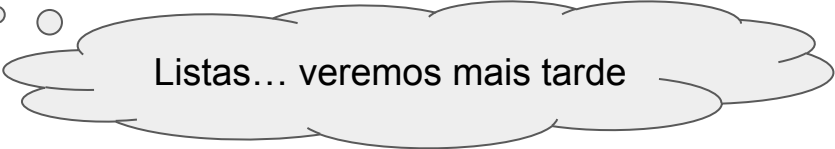
- Inteiros: 123, -1, 55322
- Vírgula flutuante: 1.33, -42.21, 1.567e+3

Exemplos de átomos

- fabrica, conversa, +, -, *, /
- :-, \=
- 'Isto é um átomo'

Exemplos de termos compostos

- xpto(1), xpto(a, b, 1, 2), um(termo, composto)
- par(1, par(2, par(3, nada)))
- [1, 2, 3]. ○ ○



Listas... veremos mais tarde

Programa em Lógica

Coleção de fórmulas: *cláusulas*.

Clausula:

- Cabeça
 - Literal
 - Funtor principal: indica o **predicado**.
- Corpo
 - Um goal
 - Conjunção de literais (o “and” é a vírgula: “,”)
 - Pode levar parêntesis
 - Pode usar disjunção (“;”) e negação por falha (“\+”)

Predicados

Um predicado é uma coleção de cláusulas com o mesmo funtor principal.

As cláusulas enumeram todos os casos possíveis para esse predicado.

Atenção: a aridade importa!

`p(a, 1, 2).`
`p(b, 2, 3).`
`p(c, 3, 4).`

p/3

`p(a, xpto).`
`p(b, foo).`
`p(c, bar).`

p/2

Notação: **NOME/ARIDADE** para designar o predicado com funtor principal NOME e ARIDADE argumentos.

Por exemplo, **morada/3** para
morada(RUA, NUM, ANDAR)

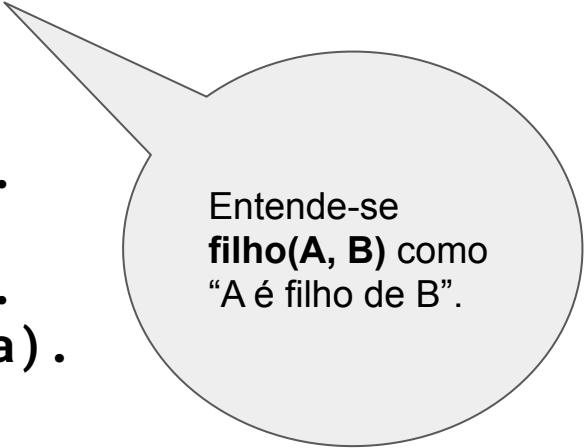
Predicados “base de dados”

Todas as cláusulas têm **corpo vazio**.

Exemplo:

```
% filho(FILHO, PAI)
```

```
filho(tomas, francisco).  
filho(tomas, ana).  
filho(mariana, francisco).  
filho(mariana, ana).  
filho(francisco, antonio).  
filho(francisco, francisca).
```



Entende-se
filho(A, B) como
“A é filho de B”.

Predicados “base de dados”

Podemos completar este programa com uma definição

```
homem(francisco).  
homem(tomas).  
homem(antonio).
```

Que podemos completar, por exemplo, com:

```
mulher(ana).  
mulher(mariana).  
mulher(francisca).
```

Clausulas com corpo não vazio

Alternativamente, podemos dizer que quem não for homem é mulher:

```
mulher(X) :- \+ homem(X).
```

Assim ficámos com menos duas cláusulas (para mulher/1).

Crítica:

- Pró: escusamos de enunciar todos os casos (mais compacto)
- Con: podemos deixar passar coisas a mais (perigoso)

Mais segurança

Exemplo de situação errada:

```
| ?- mulher(alberto).
```

Sucedee, quando a intenção seria provavelmente que falhasse.

Podemos resolver o problema com um predicado auxiliar, p/ ex

```
mulher(X) :- pessoa(X), \+ homem(X).
```

Em que **pessoa/1** produz uma solução antes do teste.

Predicados derivados

Definição de filho/2 entendida no sentido lato, i.e.

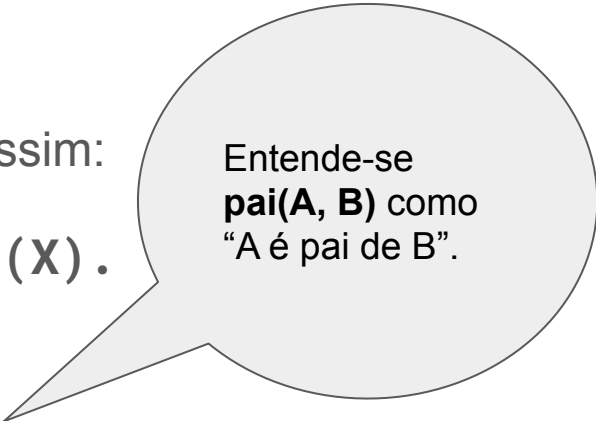
- Tanto pode designar filho como filha
- Não discrimina entre pai e mãe

Para encontrar o pai de Y, podemos fazer assim:

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

De igual modo:

```
mae(X,Y) :- filho(Y,X), mulher(X).
```



Entende-se
pai(A, B) como
“A é pai de B”.

Variáveis existencialmente quantificadas

Como dizer “A é irmão de B se tiverem o *mesmo pai* e a *mesma mãe*”?

Dizendo que:

- existe um X que é pai de A e pai de B
- existe um Y que é mãe de A e mãe de B

Não estamos particularmente interessados em saber **quais** são X e Y...

irmao(A,B) :- pai(X,A), pai(X,B), mae(Y,A), mae(Y,B).

Para melhor exprimir a relação de "irmão" (ou irmã) temos de acrescentar a indicação de que “A é diferente de B”, ou seja:

**irmao(A,B) :- pai(X,A), pai(X,B), mae(Y,A), mae(Y,B),
A \= B.**

Variáveis existencialmente quantificadas

De igual modo, como dizer que “A e B são primos (direitos)”?

R: se um seu progenitor (pai ou mãe) for irmão dum do outro...

```
primo(A, B) :- progenitor(X,A),  
               progenitor(Y,B),  
               irmao(X,Y).
```

Mais uma vez, não estamos interessados em saber **quais** os valores de X e Y.

Precisamos definir um predicado auxiliar:

```
progenitor(X,A) :- pai(X,A).  
progenitor(X,A) :- mae(X,A).
```

Estrutura dum programa

Os dados, i.e. os termos usados, constituem a base do programa.

Prever todos os casos.

Permitir expansão por via de variáveis inicialmente livres.

No caso da “família” os termos são triviais: **átomos** que representam uma pessoa.

Relações ancestrais

Para exprimirmos a relação de “avô” ou “avô” vamos definir o predicado **avo/2**, como segue:

A é **avo** de B, se A for **pai** dum X que seja **pai** de B, **OU**:

A é **avo** de B, se A for **pai** dum X que seja **mãe** de B, **OU**:

A é **avo** de B, se A for **mãe** dum X que seja **pai** de B, **OU**:

A é **avo** de B, se A for **mãe** dum X que seja **mãe** de B... e... basta!

Ou seja, trocando isto por Prologs:

```
avo(A,B) :- pai(A,X), pai(X,B).
```

```
avo(A,B) :- mae(A,X), pai(X,B).
```

```
avo(A,B) :- pai(A,X), mae(X,B).
```

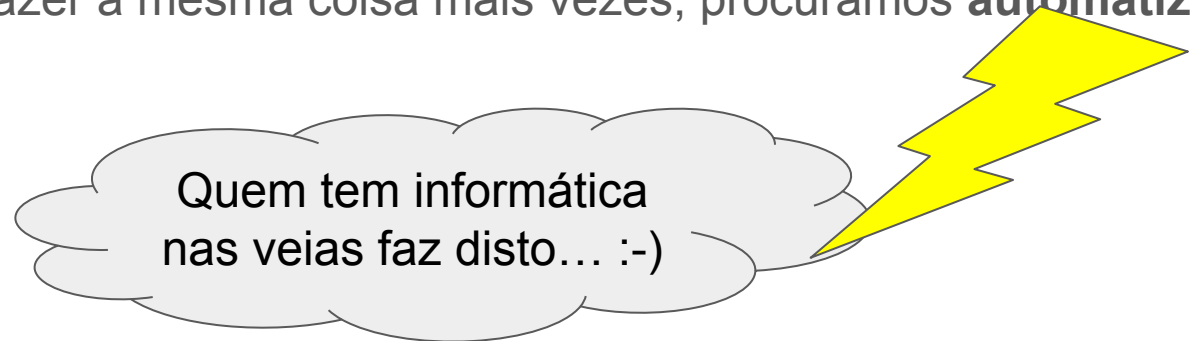
```
avo(A,B) :- mae(A,X), mae(X,B).
```

Arrumar a casa...

Quando fazemos uma coisa uma vez, ok.

Quando fazemos a mesma coisa **duas** vezes, começamos a pensar...

Quando queremos fazer a mesma coisa mais vezes, procuramos **automatizar!**



Neste caso, temos o padrão “pai ou mãe” que podemos substituir por “progenitor”, assim ficamos com:

```
avo(A,B) :- progenitor(A,X), progenitor(X,B).
```

Relações iteradas (ou recursivas)

Se quisermos falar dum pai, avô, bisavó, etc. estamos perante uma relação (“é **antepassado de**”) cuja definição é recursiva.

Dizemos que A é antepassado de B se:

A for progenitor de B, **OU:**

Se existir um X progenitor de A, que seja antepassado de B.

Traduzindo para Prolog:

```
antepassado(A,B) :- progenitor(A,B).
```

```
antepassado(A,B) :- progenitor(A,X), antepassado(X,B).
```

Hmmm... porque não este?

```
antepassado(A,B) :- progenitor(A,B).
```

```
antepassado(A,B) :- antepassado(A,X), progenitor(X,B).
```


Fecho transitivo

À relação **antepassado/2** (a primeira definição) chama-se **fecho transitivo** da relação **progenitor/2**.

Frequentemente quando se quer falar duma relação que representa um **grafo**, denotamos os **arcos** com o nome da relação e os (2) argumentos com o nome dos **nós**.

Assim, se tivermos um “mapa de estradas” com nós que representam cidades, podemos representar isso com uma relação Prolog simples...

Exemplo de mapa

```
e(lisboa, santarem).  
e(santarem, coimbra).  
e(santarem, caldas).  
e(caldas, lisboa).  
e(coimbra, porto).  
e(lisboa, evora).  
e(evora, beja).  
e(lisboa, faro).  
e(beja, faro).
```

O predicado **e/2** indica que há uma estrada (entendida como ligação direta) entre os seus dois argumentos.

Fecho transitivo

Se quisermos significar que existe um caminho de A para B, podemos fazê-lo como para a relação “antepassado”:

$\text{cam}(A, B) :- e(A, B).$

$\text{cam}(A, B) :- e(A, C), \text{cam}(C, B).$

Está tudo bem, mas temos um problema... mesmo mais... dois problemas:

1. Se quisermos que haja automaticamente um caminho de B para A se houver um de A para B (i.e. **estradas bidirecionais**), como é que fazemos?
2. E... se houver um caminho que **regresse ao ponto de partida!?!**

Fecho transitivo com ciclos

Ponto 1:

- Em vez de **e/2** usemos um predicado **a/2**, definido assim:

a(X, Y) :- e(X, Y).

a(X, Y) :- e(Y, X).

- Ficou bidirecional, mas ficámos também com um grave inconveniente: introduzimos **ciclos**!

Fecho transitivo com ciclos

Ponto 2:

Temos de **evitar** ciclos, para isso precisamos de **memória**...

Podemos registar o *caminho já efetuado* num **argumento suplementar**, que será uma espécie de **lista**, com a coleção dos sítios por onde passámos.

Dispondo desta informação, resta-nos assegurar que o “próximo passo” **não conste** da lista dos sítios já vistos.

```
cam(A, B) :- cam(A, B, A).
```

```
cam(A, B, X) :- a(A, B), nao_figura(B, X).
```

```
cam(A, B, X) :- a(A, C), nao_figura(C, X), cam(C, B, c(C, X)).
```

Fecho transitivo com ciclos

Temos de definir o predicado **nao_figura/2**, para isso diremos que **nao_figura(X,K)** se o nó X **não figurar** no caminho K.

Sendo o “caminho” a tal “espécie de **lista**”, que vai tomar a forma:

Um termo K é um caminho se:

K for um nó, **OU se**

K for da forma **c(N, KK)** em que:

N é um nó e **KK** é outro caminho.

Ou seja, dizendo isto com sotaque Prolog, define-se um caminho como:

caminho(K) :- no(K).

caminho(c(N, KK)) :- no(N), caminho(KK).

Fecho transitivo com ciclos

Já conseguimos definir o predicado `nao_figura/2`:

```
nao_figura(N,K) :- \+ figura(N,K).
```

Com a definição auxiliar:

```
figura(N, N).  
figura(N, c(N,_)).  
figura(N, c(_,K)) :- figura(N, K).
```

Tipo de dados: **lista**

Listas em Prolog são parentes das **listas ligadas** das linguagens imperativas.

Uma lista é um termo, que pode tomar os seguintes aspetos:

- A lista vazia, i.e. o átomo **[]**
- Um “nó interior”, i.e. o termo composto **.(A, B)** em que B é uma lista e A um termo qualquer, normalmente denotada **[A | B]**

Escreve-se com os elementos separados por vírgulas, o tudo entre parêntesis retos: **[e]**.

A lista **[1, 2, 3]** pode ser expressa como **.(1, .(2, .(3, [])))** ou **[1|[2|[3|[]]]]** ou ainda **[1, 2, 3|[]]**.

A lista **[um, [2, 3], quatro]** como **.(um, .(. (2, .(3, [])), .(quatro, [])))**.

Exemplo de uso de listas

O predicado “figura/2” que definimos atrás pode bem ser realizado com listas em vez dos termos compostos para “caminho”, e uns predicados pré-definidos:

member/2

memberchk/2

Por exemplo, **member(X, [1,2,3])** sucede se X for um elemento da lista (encarada como um conjunto)

Alguns predicados built-in (de sistema)

call/1: **call(G)** corresponde a interpretar o termo **G** como um novo goal, e tentar interpretá-lo, i.e. **call(G)** é verdade se **G** também for.

=/2: igualdade (unicidade): **X=Y** sucede se X e Y **forem unificáveis**. É como se estivesse definido como:

X=X.

\+/1: negação por falha, i.e. **\+ G** sucede se **G** falhar e vice-versa.