

Programação Declarativa

Paradigmas de Programação Avançados

Técnicas de Programação

Dicionários

Árvores Binárias de Pesquisa

Estruturas Incompletas

Dicionários

Encontrar coisas em Prolog (sem falhar)

Digamos que queremos representar coisas que tenham uma **chave** (K) e um **valor** (V). Podemos representar estes **pares** de várias maneiras, p/ex:

1. `K(V)`
2. `xpto(K, V)` % `xpto` pode ser substituído por qualquer outro
3. `K=V` % um caso particular de (2)
4. `[K, V]`

Exemplos, para a chave **nome** e o valor **[jose, antunes]**, podem ser:

1. `nome([jose, antunes])`
2. `xpto(nome, [jose, antunes])`
3. `nome=[jose, antunes]`
4. `[nome, [jose, antunes]]`

Vamos escolher a opção (2) com o functor `=/2`, ou seja, a opção (3).

Dicionários funcionais

Podemos usar uma **lista de pares** para fazer um dicionário.

Optámos por representar os pares **(K,V)** na forma **K=V**.

Um dicionário será entendido como uma estrutura de dados que é modificada por algumas operações, resultando numa nova instância.

As operações (a “API”) do dicionário (dito *funcional*), é esta:

1. **fdic(D)** sucede se D for um dicionário
2. **flookup(D, K, V)** sucede se o par (K,V) estiver em D
3. **finsert(DI, K, V, DO)** sucede se inserindo (K,V) em DI der DO
4. **fremove(DI, K, DO)** sucede se removendo (K,_) de DI der DO

A operação 4 (remove) é opcional.

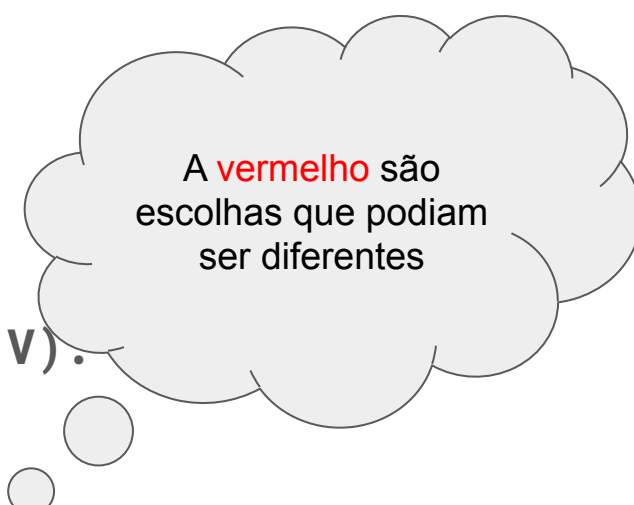
Dicionários funcionais: implementação

```
fdic([]).  
fdic([_=_|D]) :- fdic(D).
```

```
flookup([K=V|_], K, V).  
flookup([_|D], K, V) :- flookup(D, K, V).
```

```
fininsert([], K, V, [K=V]).  
fininsert([K=_|_], K, _, _) :- !, fail.  
fininsert([KV|DI], K, V, [KV|DO]) :- fininsert(DI, K, V, DO).
```

```
fremove([], _, []).  
fremove([K=_|D], K, D) :- !.  
fremove([KV|DI], K, [KV|DO]) :- fremove(DI, K, DO).
```



A **vermelho** são
escolhas que podiam
ser diferentes

Dicionários funcionais: exemplo

```
| ?- finsert([], n, xpto, D1),  
      finsert(D1, m, foo, D2),  
      finsert(D2, zz, bar, D3),  
      fremove(D3, m, D4),  
      flookup(D3, m, VM).
```

D1 = [n=xpto]

D2 = [n=xpto,m=foo]

D3 = [n=xpto,m=foo,zz=bar]

D4 = [n=xpto,zz=bar]

VM = foo ?

yes

```
| ?- finsert([a=b], a, c, D).
```

no

```
| ?- finsert([a=b], aa, c, D).
```

D = [a=b,aa=c]

yes

```
| ?-
```

Dicionários lógicos

Podemos considerar a lista como uma estrutura dita incompleta, ou seja, que termina com uma variável livre em vez da lista vazia. Assim podemos rever os predicados de dicionário para terem um só parâmetro D, que será modificado.

Assim, ficamos com:

1. **ldic(D)** sucede se D for um dicionário lógico
2. **llookup(D, K, V)** sucede se o par (K,V) estiver em D
3. **linsert(D, K, V)** sucede se inserirmos (K,V) em D
4. **lremove(D, K)** sucede se removermos (K,_) de D

A operação 4 (remove) é mais difícil de realizar.

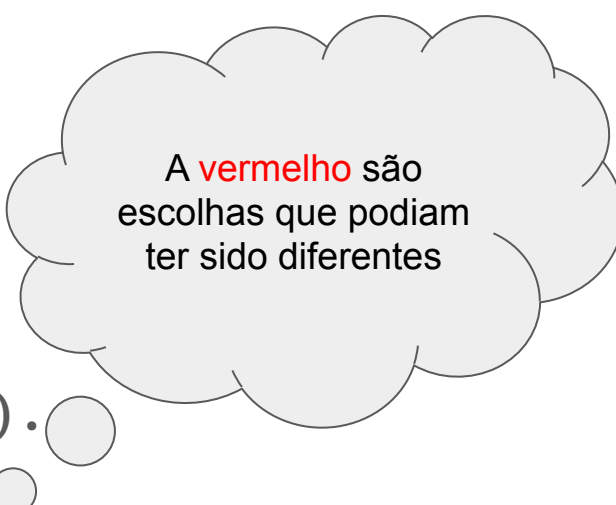
Dicionários lógicos: implementação

```
ldic(D) :- var(D), !.  
ldic([_=_|D]) :- ldic(D).
```

```
llookup(D, _, _) :- var(D), !, fail.  
llookup([K=V|_], K, V).  
llookup([_|D], K, V) :- lookup(D, K, V).
```

```
insert(D, K, V) :- var(D), !, D=[K=V|_].  
insert([K=_|_], K, _) :- !, fail.  
insert([KV|D], K, V) :- insert(D, K, V).
```

% remove/3 não foi implementado aqui



A **vermelho** são
escolhas que podiam
ter sido diferentes

Subtermo

Vamos definir um predicado `subterm/2` em que `subterm(ST, T)` sucede se `ST` for um subtermo de `T`, i.e. se ocorrer no interior de `T`.

Uma implementação usando os builtins `functor/3` e `arg/3`:

```
subterm(T, T).  
subterm(ST, T) :-  
    compound(T),  
    functor(T, _, N),  
    subterm(N, ST, T).  
  
subterm(N, ST, T) :-  
    N > 1, N1 is N-1,  
    subterm(N1, ST, T).  
subterm(N, ST, T) :-  
    arg(N, T, A),  
    subterm(ST, A).
```

Subterm como lookup...

Podemos “desviar” subterm/2 para fazer as vezes do flookup/3:

```
| ?- subterm(X, [a=b]).  
X = [a=b] ? ;  
X = (a=b) ? ;  
X = a ? ;  
X = b ? ;  
X = [] ?  
(1 ms) yes  
| ?- subterm(K=V, [a=b, b=c]).  
K = a  
V = b ? ;  
K = b  
V = c ? ;  
(1 ms) no  
| ?-
```

Encontrar coisas em Prolog (podendo falhar)

Uma maneira fácil de memorizar coisas, para lhes aceder mais tarde é usar os predicados ditos de “**base de dados**” (`asserta/1`, `assertz/1`, `retract/1`).

Por exemplo, para guardar e recuperar valores – designados por *V* (*value*), indexados por um termo (ground), digamos que lhe chamamos *K* (*key*), podemos fazer:

```
insert(K, V) :- retract(xpto(K, _)), fail.
```

```
insert(K, V) :- asserta(xpto(K, V)).
```

```
lookup(K, V) :- xpto(K, V).
```

Embora simples, este método pode ser menos eficaz se o usarmos muito intensamente (porque o `assert` é caro).

Encontrar (muitas) coisas em Prolog

Se tivermos muitas chamadas a fazer a **insert/2** e **lookup/2**, será melhor procurar uma implementação mais eficaz.

Solução: usar as ***variáveis globais***.

No caso do GNU Prolog, são *persistentes*, no caso do SWI (e YAP, que partilha a base de código), serão contextuais ao goal em que aparecem.

GNU Prolog:

g_assign(CHAVE, VALOR) e **g_read(CHAVE, VALOR)**

SWI Prolog:

b_setval(CHAVE, VALOR) e **b_getval(CHAVE, VALOR)**

Árvores binárias de pesquisa

ABPs e nós

Quando queremos ter estruturas de dados mais eficazes em termos das operações de dicionário do que as listas (complexidade $O(N)$ para todas as operações), temos algumas escolhas.

A árvore binária de pesquisa é uma delas; tem complexidade $O(\log N)$ para as operações de dicionário, portanto é preferível.

Vamos representar uma árvore binária de pesquisa (ABP) como um termo da forma:

- **no(KV, L, R)**
- **nil**

Em que KV é um par $K=V$, como anteriormente, e L e R são termos que denotam novas ABPs.

Só ABPs agora

% abp(A) se A for binaria, dois casos: no(XXX, L, R) e nil

abp(nil).

abp(no(_, L, R)) :- abp(L), abp(R).

% alookup(ABP, K, V) se (K,V) for membro de ABP

alookup(no(K=V, _, _), K, V).

alookup(no(X=_, L, _), K, V) :- K @< X, !, alookup(L, K, V).

alookup(no(X=_, _, R), K, V) :- K @> X, !, alookup(R, K, V).

% ainsert(IN, K, V, OUT)

ainsert(nil, K, V, no(K=V, nil, nil)).

ainsert(no(X=VX, L, R), K, V, no(X=VX, LL, R)) :-

K@<X, ainsert(L, K, V, LL).

ainsert(no(X=VX, L, R), K, V, no(X=VX, L, RR)) :-

K@>X, ainsert(R, K, V, RR).

Exemplo de uso de ABPs

```
| ?- ainsert(nil, c,1, A),  
      ainsert(A, b,2, B), ainsert(B, d,4, D).
```

```
A = no(c=1,nil,nil)
```

```
B = no(c=1,no(b=2,nil,nil),nil)
```

```
D = no(c=1,no(b=2,nil,nil),no(d=4,nil,nil)) ?
```

```
(1 ms) yes
```

```
| ?-
```