

1 Predicados de Inspeção de Estrutura

Predicados de inspeção de estrutura nos passam informações sobre um termo específico. Por exemplo, será um termo atômico, numérico, constante, variável? Ou será um funtor composto? Se for, qual será seu funtor principal, qual sua aridade e quais seus argumentos? É esse tipo de questão que os predicados de inspeção de estrutura respondem.

1.1 Predicados de tipos

Alguns dos, assim chamados, predicados de tipo são:

- `integer/1`;
- `real/1`;
- `atom/1`;
- `compound/1`;

Cada um deles pode ser interpretado como uma lista infinita de átomos. Por exemplo, `integer/1` pode ser interpretado como: `integer(0)`. `integer(1)`. `integer(2)`. `integer(3)`,

A partir desses predicados podemos criar outros. Por exemplo, podemos fazer `numero(X) :- integer(X); real(X)`, ou `constante(X) :- numero(X); atom(X)`.

1.2 Acesso de termos compostos

Queremos ser capaz, além de lidar com tipos, lidar com funtores. Para acessar o funtor principal, temos o predicado `functor/3`. `functor(Termo, F, Aridade)?` tem sucesso se o funtor principal de `Termo` tem aridade `Aridade` e nome `F`. Assim, por exemplo, `functor(f(x1, ..., Xn), f, n)` tem sucesso, já que o funtor principal é `f/n`.

Pode-se usar esse predicado para, entre outras coisas, decomposição e criação de termos:

1. `functor(tio(a,b), X, Y)?` tem como solução `{X = tio, Y = 2}`;
2. `functor(F, tio, 2)` tem como solução `{F = tio}`.

Note que, no Exemplo 2, se o goal fosse `functor(F, tio, N)?`, teríamos erro, já que o interpretador não consegue adivinhar a aridade de um funtor a partir do nome. Analogamente, `functor(F, tio(a, b), N)?` resultaria em

erro, já que **functor** espera um átomo como segundo argumento, não um functor composto. Mas, **functor(tio, X, N)?** teria sucesso, com $\{X = \text{tio}, N = 0\}$ (átomos são considerados funtores de aridade 0).

Similar, a **functor/3** é o **arg/3**: **arg(N, F(X_1, \dots, X_n), Q)?** tem sucesso se o N-ésimo argumento de F é o Q. Assim como **functor**, **arg/3** é comumente usado para decomposição e criação de termos:

- Para decompor um termo, **arg/3** acha um argumento particular de um termo composto;
- Para criar um termo, ele instancia um argumento variável de um termo.

Por exemplo, **arg(1, tio(a,b), X)?** tem sucesso com $\{X = a\}$, enquanto que **arg(1, tio(X,b), a)?** tem sucesso com $\{X = a\}$.

Exemplos um pouco mais interessantes são os seguintes:

Código 1: Sub Termo

```
% subtermo(Sub, termo) :-
%   Sub eh um subtermo de termo
%
```

```
subtermo(Termo, Termo).
subtermo(Sub, Termo) :-
    compound(Termo),
    functor(Termo, _, N),
    subtermo(N, Sub, Termo).
```

```
subtermo(N, Sub, Termo) :-
    arg(N, Termo, Arg),
    subtermo(Sub, Arg).
subtermo(N, Sub, Termo) :-
    N > 1,
    N1 is N - 1,
    subtermo(N1, Sub, Termo).
```

Código 2: Substituto

```
% substituto(Velho, Novo, Velt, Novt) :-
%   Novt eh o resultado de trocar
%   todas as ocorrencias de Velho no termo Velt por Novo
%
```

```
substituto(Ve, No, Ve, No).
substituto(Ve, _, Vet, Vet) :-
    constante(Vet),
    Vet \= Ve.
```



```

Goal =.. [P|Xs] ,
Goal.

map( _ , [] , [] ).
map(P , [X|Xs] , [Y|Ys]) :-
    list(X) ,
    flatten([X|Y] , Z) ,
    apply(P , Z) ,
    map(P , Xs , Ys).

map(P , [X|Xs] , [Y|Ys]) :-
    apply(P , [X , Y]) ,
    map(P , Xs , Ys).

```

1.3 Predicados de Meta-Programação

O predicado de meta-programação básico é o `var/1`: `varT?` tem sucesso se `T` é uma variável não instanciada, e falha caso contrário. Sua irmã, `nonvar/1`, funciona de maneira análoga. Por exemplo, `var(a)?` e `var([X|Xs])?` falham, enquanto que `var(X)?` tem sucesso, se `X` é uma variável não instanciada.

Esse tipo de predicado é chamado meta-predicado: eles são “predicados sobre predicados”. Como tais, podem prover grande flexibilidade a um programa lógico.

1.4 Operações Aritméticas

Antes de continuarmos, será útil introduzirmos operações aritméticas. Primeiro, é importante notar que a atribuição de uma operação aritmética, no Prolog, funciona de maneira diferente da unificação. Ela é do tipo `X is A Op B.`, onde `Op` é uma operação com valores aritméticos (que resulta em um valor aritmético) do tipo `a` que estamos acostumados: `1 + 2`, `3 * 8`, `(1 - 5)/4 + 33475`, etc, e `A` e `B` são valores. É importante notar que operações aritméticas só podem ser feitas com números: `A Op B`, onde `Op` é uma operação aritmética, resulta em erro se `A` ou `B` for uma variável ou uma constante não numérica. Vale notar que `is/2` não é simétrico: `5 + 4 is A` resulta em erro.

Assim, vemos que o operador de ou exclusivo (o `xor`), por exemplo, não age da maneira como estamos acostumados: gostaríamos que `1 is A xor 0` seja equivalente a `A is 1`. Veremos mais tarde outras formas de contornarmos isso, mas, por ora, podemos lidar com isso por meio de predicados de meta-programação:

Código 4: Ou Exclusivo

```
% ou_exclusivo(X, Y, Z) :-  
%   se ao menos dois dos argumentos nao sao variaveis ,  
%   o resultado eh o esperado  
%  
  
ou_exclusivo(X, Y, Z) :- nonvar(X), nonvar(Y), Z is X xor Y.  
ou_exclusivo(X, Y, Z) :- nonvar(Z), nonvar(Y), X is Z xor Y.  
ou_exclusivo(X, Y, Z) :- nonvar(X), nonvar(Z), Y is Z xor X.
```