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(`x_1`, ..., `X_n`), 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. Análogamente, `functor(F, tio(a, b), N)?` resultaria em

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

Similar a functor/3 é o arg/3: `arg(N, F(`X_1`, ..., `X_n`), Q)?` tem sucesso se o Nésimo 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 (o símbolo de percentagem denota um comentário no código, enquanto que o underline denota uma variável anônima):

É uma variável a que não se dá nome. Assim, não se pode chamá-la depois: cada variável anônima é diferente.

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: Substitutu

```
% 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.

substituto(Ve, No, Vet, Not) :-
  compound(Vet),
  functor(Vet, T, N),
  functor(Not, T, N),
  substituto(N, Ve, No, Vet, Not).

substituto(N, Ve, No, Vet, Not) :-
  N > 0,
  arg(N, Vet, Arg),
  substituto(Ve, No, Arg, Arg1),
  arg(N, Not, Arg1),
  N1 is N - 1,
  substituto(N1, Ve, No, Vet, Not).
substituto(0, _, _, _, _) :- !.
```

Outro predicado de inspeção de estrutura é o, assim chamado, *univ*, escrito como `=../2`[1]. Um exemplo de seu uso é `Termo =.. [f,a,b]?`, com o resultado {Termo = f(a, b)}. O *univ* pode ser usado de essencialmente duas formas diferentes:

1. Com um funtor ao lado esquerdo e uma váriavel no direito: a variável é unificada com `[f|v]`, onde f é o funtor principal e V a lista de seus argumentos;
2. Com uma variável ao lado esq analisam os predicados."uerdo e uma lista no direito: se a lista é `[a,` b_1`, ..., ` b_n`]`, a variável é unificada com `a(`b_1`, ..., ` b_n`)`.

O seguinte programa mostra um exemplo da utilidade de *univ*:

Código 3: Map

```
% map(P, Xs, Ys) :-
%   Ys eh o resultado de se aplicar P a cada elemento de Xs
%   se P for uma P/n para n > 1, Xs precisa ser uma lista de listas
%
% map/3 usa apply/2 como auxiliar
%
% apply(P, [X1, ..., Xn]) :-
%   P(X1, ..., Xn)? tem sucesso
```

[1]Predicados para acesso e construção de termos têm origem na família Prolog de Edinburgo (que tem se tornado o padrão de facto) e alguns dos nomes usados vêm de lá. A forma "=.." para univ vem do Prolog-10, onde era usado ",.." no lugar de "—" em listas (`[a, b,.. Xs]` no lugar de `[a, b|Xs]`).

```
%

apply(P, Xs) :-
  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 instnanciada, 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 Op.`, onde `Op` é uma operação com valores, do tipo a que estamos acostumados: 1 + 2, 3 * 8, (1 - 5)/4 + 33475, etc. 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. Além disso, is/2 não é simétrico: `5 + 4 is A` resulta em erro.

Assim, vemos que o operador de adição, por exemplo, não aje da maneira como estamos acostumados: gostaríamos que `5 is A + 4` seja equivalente a `A is 5 - 4`. Veremos mais tarde outras formas de contornarmos isso, mas, por ora, podemos lidar com isso por meio de predicados de meta-programação:

Código 4: Map

```
% mais(X, Y, Z) :-
%    se ao menos duas das letras nao sao variaveis,
%    o resultado eh o desejado
%

mais(X, Y, Z) :- nonvar(X), nonvar(Y), Z is X+Y.
mais(X, Y, Z) :- nonvar(Z), nonvar(Y), X is Z-Y.
mais(X, Y, Z) :- nonvar(X), nonvar(Z), Y is Z-X.
```