1 Predicados de Inspeção de Estrutura

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

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. O goal `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)`, onde os X_i são variáveis, tem sucesso, já que o funtor principal é f/n.

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

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

Note que, no item 2 acima, 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 funtor composto. Mas, `functor(tio, X, N)?` teria sucesso, com {X = tio, N = 0}.

Similar a functor/3 é o arg/3: `arg(N, F(`X_1`, ..., `X_n`), Q)?` tem sucesso se o Enésimo 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 $\{X = \text{a}\}$, enquanto que `arg(1, tio(X,b), a)?` tem sucesso com $\{X = \text{a}\}$.

Exemplos um pouco mais interessantes são os seguintes:

```
% subtermo(Sub, termo) :-
%   Sub eh um termo que aparece em 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).

% 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 (por razões históricas obscuras), *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 variável 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 esquerdo 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*:

```
% map(P, Xs, Ys) :-
%   Ys eh o resultado de se aplicar P a cada elemento de Xs
%   se P for 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
%

apply(P, Xs) :-
  Goal =.. [P|Xs],
```

[1] Predicados para acesso e construção de termos têm origem na família Prolog de Edimburgo (que tem se tornado o padrão de fato) e alguns dos nomes usados vêm de lá. Em particular, 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]`).

```
  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

Digamos que você queira fazer um interpretador de Prolog em Prolog (algumas possíveis aplicações disso são *debuggers*). O mais simples possível é dado a seguir:

```
solve(Goal) :- Goal.
```

Esse interpretador, sozinho, é de pouca utilidade. Nos dá a possibilidade de acessar quase nada do programa. Se quisermos fazer melhor, precisaremos de predicados que nos digam mais sobre predicados (ou seja, meta-predicados, predicados de meta-programação).

Um predicado de meta-programação básico é o clause/2. O goal `clause(Head, Body)?` é verdadeiro se *Head* for unificável com a cabeça de uma cláusula e *Body* com seu respectivo corpo (se *Head* for um fato, *Body* é unificável com *true*).

Com isso, podemos fazer um meta-interpretador, algo mais elaborado:

```
solve(true).

solve((A,B)) :-
  solve(A), solve(B).
solve(A) :-
  clause(A,B), solve(B).
```

Precisamos dos parênteses a mais em `solve((A,B))` por razões técnicas (não queremos confundir o compilador). Esse interpretador teria que ser estendido se quisermos que faça tudo o que é esperado de um interpretador Prolog (ele não lida apropriadamente com cortes, por exemplo, ou com entradas pelo teclado). Isso poderia ser corrigido sem grandes dificuldades.

O Prolog nos possibilita não só facilmente escrever interpretadores para a própria linguagem (o que pode ser útil, por exemplo, na construção de *debuggers*, de sistemas especializados, de programas autocorretores, de programas

que "se explicam" em termos de porquês e comos entre outros), mas também nos possibilita facilmente escrever interpretadores para outras linguagens e para dialeto dessas. Por exemplo, alguém poderia argumentar que o Prolog, sendo uma linguagem de programação lógica, não lida bem com situações em que a incerteza é uma parte importante. Mas, com ele, podemos facilmente criar nossa própria linguagem para fazer isso. Para tanto, suponhamos, por exemplo, que cláusulas com uma certeza (a probabilidade de estar correta) C sejam representadas pelo funtor clause_c/3, em que `clause_c(Head,Body,C)?` é verdade quando *Head* unifica com uma cabeça de cláusula cujo corpo unifica com *Body* e cujo "fator de certeza" unifica com C, e considere o seguinte:

```
solve(true, 1, Limit) :- !.

solve((G1, G2), C, Limit) :-
  !, solve(G1, C1, Limit), solve(G2, C2, Limit),
  C is min(C1, C2).

solve(G, C, Limit) :-
  clause_cf(G, B, C1), C1 > Limit,
  Limit1 is Limit/C1, solve(B, C2, Limit1),
  C is C1 * C2.
```

Um goal `solve(Goal,C,Limit)` submetido a esse interpretador resulta em sucesso se a "confiança" C de *Goal* for maior do que o limite inferior *Limit*. Alguns pontos válidos de se notar nesse programa são que, numa conjunção `A,B`, a nossa confiança na conjunção é tomada como o mínimo entre a de A e a de B e que se, para provar um goal *Goal*, é preciso passar por uma cláusula com fator de certeza *C1*, a prova do corpo da cláusula terá um fator de certeza menor, dado por `Limit/C1`. Isto porque queremos que nossa "confiança" (pensando de forma probabilística) *C1* na cláusula vezes a confiança *C2* (pense na probabilidade de eventos independentes) na resolução do corpo da cláusula seja maior que *Limit*.

1.3.1 Meta-predicados de variáveis

Outro predicado de meta-programação básico é o var/1: `var(T)?` tem sucesso se T é uma variável não instanciada e falha caso contrário. Sua irmã, novar/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 predicado nos permite fazer, por exemplo, programas como o seguinte:

```
length(Xs, N) :- nonvar(Xs), length1(Xs, N).
length(Xs, N) :- var(Xs), nonvar(N), length2(Xs, N).
```

em que length1/2 e length2/2 são dados pelos length no (Capítulo 4).