1 Listas

Antes de prosseguirmos em outros aspectos da programação lógica, ocasionalmente será util sabermos trabalhar com **listas**:

Definição 1.1. *Usaremos a seguinte definição formal de lista:* **Lista**

- `.()` *(o "funtor .") é uma lista*[1] *(o que chamamos "lista vazia", é o mesmo que "[]" das seguintes convenções);*
- `.(A, B)` *é uma lista se B é uma lista*

Como é chato ficar escrevendo coisas como `.(A, .(B, C))` *usaremos as seguinte convenções:*

- `[A,B]` *é o mesmo que* `.(A, .(B, .()))`*;*
- `[A]` *é o mesmo que* `.(A, .())` *(o funtor . é de aridade 2, a não ser quando não recebe arguentos);*
- `[A, B, C, ...]` *é o mesmo que* `.(A, .(B, .(C, ...)))`*;*
- `[A|B]` *indica que A é o primeiro elemento da lista e B é o resto da lista (B é uma lista).*

Dado isso, podemos escrever um programa para adicionar um elemento na lista como o seguinte:

Código 1: Append

```
append([], A, A).
append([X|Y], A, [X|C]) :- append(Y, A, C).
```

Esse é um programa clássico e, por isso, escolhemos manter seu nome clássico, que, daqui para frente será usado sem itálico. Os dois elementos iniciais de append são listas e o final é o resultado de se "juntar as duas listas": cada elemento da primeira lista está, ordenadamente, antes da cada elemento da segunda. Para exercitar, pense em qual seria o resultado do goal `append([cafe, queijo], [goiabada], L)?`.

Para uma melhor compreensão, será instrutivo analisarmos o seguinte programa:

[1] Na verdade, isso não é extritamente padrão e implementações diferentes podem usar funtores diferentes. Essa é outra razão para não usarmos essa definição na prática, mas sim a convenção seguinte. Apesar disso, é importante se lembrar que a lista não é, extritamente falando, diferente de um funtor.

Código 2: Length

```
% length(Xs, N) :-
%   N     a quantidade de elementos na lista Xs
%

lenght([X|Xs], N) :-
  length(Xs, K),
  N is 1 + K.

length([], 0).
```

Mas antes, precisamos entender o que significa `N is M + 1.`. Esse **is** é o operador de atribuição aritmético e ele não funciona como os demais predicados: para algo como `A is B`, se B for uma constante numérica e A é uma variável, o comportamento é o esperado (A assume o valor de B); se não, ocorre erro. Disso segue que is/2 não é simétrico: por exemplo, `A is 5` resulta em A = 5, mas `5 = A.`, onde A é uma variável, resulta em erro. Ademais, quando algum operador aritmético[2] op é usado com is/2, o operador realiza a operação esperada: `A is 2 + 3` e `A is 10 - 8` resultam em A = 5, por exemplo. Vale notar que esse comportamento é diferente do =/2, por exemplo: o seguinte programa não tem o resultado esperado:

is

Código 3: Length

```
lenght([X|Xs], N) :-
  length(Xs, K),
  N = 1 + K.

length([], 0).
```

Isso porque =/2 tem um efeito puramente simbólico. Em particular, =/2 não realiza operações aritméticas. Além do is/2, temos **operadores de comparação** (também chamados de operadores relacionais), que funcionam de maneira semelhante e também possibilitam o uso de operadores aritméticos à direita do símbolo, na maneira usual. Eles serão de alguma importância:

- =:= /2, de igualdade;
- = / = /2, de desigualdade;
- > /2, de "maior que";
- >= /2, de "maior ou igual que";

[2]Operadores aritméticos que temos à disposição são: $+/2, -/2, */2, //2, \wedge/2, -/1, abs/1, sin/1, cos/1, max/2, sqrt/1/ \ll /1, \gg /1$. O funcionamento de muitos deles é claro pelo símbolo, o dos demais será explicado na medida que forem usados.

- < /2, de "menor que";
- =< /2, de "menor ou igual que";

Voltando ao programa 1. O goal `length(Xs, N)?`, onde Xs é uma lista e N uma variável, resulta em N tomando o valor da quantidade de elementos de Xs (o seu "tamanho"). Mas, o goal `length(Xs, N)?`, onde N é um número natural positivo e Xs uma variável resulta em erro ao chegar no trecho `N is K + 1`, já que, como dito anteriormente, `N is K`, onde N é um número, resulta em erro.

Para o seguinte programa, esse já não é o caso.

Código 4: Length

```
length([X|Xs], N) :-
  N > 0,
  K is N -1,
  len(Xs, K).

length([], 0).
```

O goal `length(Xs, N)?`, para esse programa, resulta em erro se N não for um número. Entretanto, se for um natural positivo, `length(Xs, N)?` resulta em sucesso e Xs se torna uma lista de N elementos. Se Xs for uma lista e N um natural positivo, `length(Xs, N)?` resulta em falha se Xs tem uma quantidade de elementos diferente de N e sucesso se Xs tem uma quantidade de elementos igual a N.

Nota-se que, apesar dessa diferença, a leitura do programa é essencialmente a mesma. A diferença decorre da maneira como os operadores aritméticos funcionam em Prolog e leva a outras situações parecidas, o que eventualmente se tornará um incoveniente grande demais. Veremos como lidar com esse tipo de incoveniente de maneiras diferentes no capítulo de inspeção de estruturas e, depois, no de restrições lógicas.

Agora, voltando rapidamente a um assunto discutido no capítulo passado, temos a definição de **lista completa**:

lista completa

Definição 1.2. *Uma lista L é completa se toda instância Lι satisfaz a definição de lista dada. Se existem instâncias que não a satisfazem, ela é dita incompleta.*

Por exemplo, a lista `[a,b,c]` (menos popularmente conhecida como `.(a,.(b,.(c,.())))`) é completa: a `[a,b|Xs]` (novamente, menos popularmente conhecida como `.(a,.(b,Xs))`), não.

Se o primeiro argumento do length/2, do código 3 acima, for uma lista completa, a computação termina, enquanto que, se for uma lista incompleta, não.