1 Listas

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

Definição 1.1. *Usaremos a seguinte definição formal de 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, []))` *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 argumentos);*
- `[A, B, C, ...]` *é o mesmo que* `.(A, .(B, .(C, ...)))`*;*
- `[A|B]` *indica que A é o primeiro elemento da lista (também chamado de* ***cabeça da lista****) e B é o resto da lista, também chamado de* ***corpo da lista*** *.* **cabeça da lista** **corpo da 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 estritamente 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, estritamente 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).
```

1.0.1 Operadores aritméticos

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 A for uma constante numérica e B for outra constantes, ocorre falha. Em outras ocasiões, ocorre erro. Disso segue que is/2 não é simétrico: por exemplo, `A is 5` resulta em A = 5, mas `5 is A.`, onde A é uma variável não instanciada, 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 pode não ter o resultado intuitivamente esperado: **is**

Código 3: Length

```
length([X|Xs], N) :-
  length(Xs, K),
  N = 1 + K.
length([], 0).
```

Intuitivamente, esperaríamos que o N fosse unificado, por um processo recursivo, com o número correspondente ao tamanho da lista. Isso não ocorre, porque =/2 tem um efeito puramente simbólico e não realiza operações aritméticas: o que teríamos em N seria algo como uma *string* de símbolos *1+(1+(1+0))*, ao invés da avaliação dessa *string*, ou seja, 3.

Outro exemplo é o do operador ou exclusivo (o *xor*), que também 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. Por ora, podemos lidar com isso por meio de predicados de meta-programação (os quais serão melhor explicados no Capítulo 6):

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

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.
```

Temos à nossa disposição também *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, de "menor que";
- =</2, de "menor ou igual que";

Voltando ao código 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`, uma vez que, como dito anteriormente, `N is K`, onde *N* é um número e *K* não, resulta em erro.

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

Código 5: Length

```
length([X|Xs], N) :-
  N > 0,
  K is N -1,
   length(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 procedural, a leitura declarativa 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.

1.0.2 Flattening

Uma característica importante da lista, que lhe dá a flexibilidade necessária para poder representar muitos tipos de dados diferentes é que ela é "fechada sob a relação de instanciação", isto é, não existe problema em fazer uma lista de listas. Assim, uma lista perfeitamente válida é `[[[a,b],c],[]]`. Algumas vezes, entretanto, será útil assumirmos que uma lista L só contenha "não-listas" como elementos. Para tanto, podemos fazer uso do funtor flatten/1, que pode ser implementado como se segue:

Código 6: Flatten

```
flatten([X|Xs], Ys) :-
  flatten(X, Ys1),
  flatten(Xs, Ys2),
  append(Ys1, Ys2, Ys).
flatten(X, X) :-
  atomic(X),
  X \= [].
flatten([], []).
```

O atomic/1 usado é avaliado como sucesso se seu argumento for uma constante (`atomic(A)?` resulta em sucesso se A for um funtor de aridade zero).

1.0.3 Lista Completa

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 $L\iota$ 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]` (menos popularmente conhecida como `.(a, .(b,Xs))`), não. Isso porque Xs não tem, a princípio, obrigação de ser uma lista.

1.1 Listas de diferença

Estruturas de dados incompletas, no geral, podem ser bem importantes e úteis. Um exemplo interessante são as **listas de diferença**, uma estrutura de dados que pode simplificar e aumentar a eficiência de programas que lidam com listas.

listas de diferença

Listas de diferença tem esse nome porque são criadas como a diferença de duas listas. Por exemplo, dizemos que a diferença entre as listas `[a,b,c]` e `[c]` é a lista `[a,b]`. A diferença entre duas listas incompletas `[a,b|Xs]` e `Xs` é equivalente à lista `[a,b]` e, mais no geral, a diferença entre duas listas incompletas `[`x_0`, ...,`x_i`|Xs]` e `Xs` é equivalente a `[`x_0`, ...,`x_i`]`. A diferença entra as listas *Ys* e *Xs*, onde `Ys = [Zs|Xs]`, é denotada `Ys\Xs`, onde *Ys* é dita a *cabeça* e *Xs* a cauda. Na prática, poderíamos definir um funtor tal como lista_diff/2, o que seria potencialmente mais eficiente, mas a notação anterior será mais conveniente pelo momento. Se eficiência for uma preocupação, termos como `Ys\Xs` poderiam ser substituídos automaticamente por outro funtor apropriado.

É importante notar que qualquer lista *L* pode ser trivialmente representada na forma de lista de diferença como `L\[]`. Fazer a concatenação de uma lista de diferença `Ys\Xs` com uma `Zs\Ws` só é possível quando *Xs* seja unificavel com *Zs*, sendo, nessa ocasião, ditas **listas compatíveis** e, nesse caso, resulta na lista de diferença `Zs\Xs`. Esse fato é capturado no seguinte código:

listas compatíveis

Código 7: Append DL

```
append_dl(Xs\Zs, Ys\Xs, Ys\Zs).
```

Perceba que, enquanto no código 1 a concatenação realiza uma quantidade de operações linear no tamanho da lista, no código 1.1 a concatenação é realizada em uma quantidade constante de operações.

Outro exemplo de programa que poderia ser melhorado com o uso de listas de diferença é o flatten/2. Se queremos realizar o *flatten* de uma lista Xs e temos um flatten_dl/2 que realiza o flatten em uma lista diferença, sabemos que `flatten(Xs,Ys)` é o mesmo que `flatten_dl(Xs\[], Ys\[])`. Um flatten_dl/2 pode ser como o seguinte:

Código 8: Flatten DL

```
flatten_dl([X|Xs], Ys\Zs) :-
  flatten_dl(X, As\Bs), flatten_dl(Xs, Cs\Ds),
  append_dl(As\Bs, Cs\Ds, Ys\Zs).
flatten_dl(X,[X|Xs]\Xs) :-
  atomic(X),
  X \= [].
flatten_dl([],Xs\Xs).
```

Perceba agora que o passo em que usamos append_dl/2, no código 1.1, pode ser feito de maneira implícita, resultando no seguinte

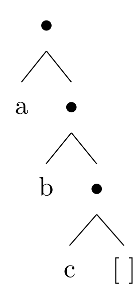
Código 9: Flatten DL o retorno

```
flatten_dl([X|Xs], Ys\Zs) :-
  flatten_dl(X, Ys\Bs), flatten_dl(Xs, Bs\Zs).
flatten_dl(X,[X|Xs]\Xs) :-
  atomic(X),
  X \= [].
flatten_dl([],Xs\Xs).
```

que parece melhor do que nosso flatten/2 original. Essa mudança poderia ser obtida automaticamente com uma aplicação de um *unfolding*. *Unfolding* é um tipo de "transformação programática" que consiste, em termos gerais, na substituição de um goal por sua definição e é o contrário de *folding*, que consiste na substituição do corpo de uma cláusula por sua cabeça. Essas transformações são úteis na otimização de código e para outras coisas, que fogem do nosso escopo atual.

Vistos os exemplos de listas de diferença dados, é justo dizer que a ideia de estruturas de diferença parecem boas e nos perguntar se ela não é generalizavel para tipos de dados diferentes de listas. Para tanto, precisamos desenvolver uma representação um pouco melhor de o que a lista é e como ela funciona. Uma lista, como a definimos acima, é uma forma de representar uma árvore. Por exemplo, a lista `[a,b,c]` se parece com:

Árvore 1: Lista simples

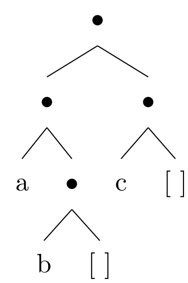


onde o • representa o funtor ./2 da lista. Na verdade, funtores em geral são árvores, não só os de lista, mas funtores de lista tem essa "cara especial". Uma lista como `[[a,b],c]` seria como:

O que flatten/2 faz é uma transformação em árvores como essa, transformando uma lista aninhada como a 2 em uma simples, como a 1. No caso, o resultado de flatten na lista 2 (que é uma árvore) seria a lista 1.

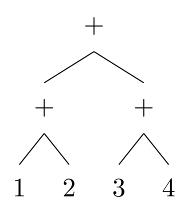
Agora, para vermos como uma estrutura de diferença pode ser útil em outras ocasiões, considere o seguinte exemplo. Em Prolog, a operação de soma é associativo a esquerda, o que significa que a operação `1 + 1 + 1 + 1` é tomada como `(((1 + 1) + 1) + 1)`. Por razões técnicas, podemos querer que ela seja

Árvore 2: Lista aninhada



normalizada como associativa à direita. Ou seja, se temos algo como `(1 + 2) + (3 + 4)`, dado pela árvore

Árvore 3: Soma



queremos que isso se torne `(1 + (2 + (3 + 4)))`, dado pela árvore 4.

O que precisamos é de uma forma de normalizar essa operação. Para tanto, precisamos de um novo funtor (já que o "+" já está em uso). Definiremos o funtor ++/2 como um operador infixo, da seguinte forma:

```
:- op(500, xfy, ++).
```

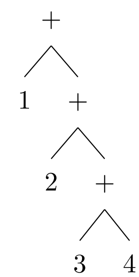
Resumidamente, essa linha nos diz que o funtor ++/2 é um operador binário (podemos usá-lo na forma `A ++ B`), de prioridade 500 (quando maior o número, menor a prioridade de avaliação, sendo a menor prioridade possível dada por 1200) e associativo à direita (yfx seria associativo à esquerda e xfx seria não-associativo).

Com esse funtor em mãos, definimos a "soma de diferença" de maneira análoga à lista de diferença, isto é, como `S1++S2`, onde *S1* e *S2* são somas normalizadas incompletas. Nesse contexto, o número 0 faz o papel da lista vazia e `S1++0` é equivalente a *S1*. Assim, podemos definir o seguinte código:

Código 10: Normalize

```
normalize(Exp, Norm) :- normalize_ds(Exp, Norm++0).
```

Árvore 4: Soma normalizada




```
normalize_ds(A+B, Norm++Space) :-
   normalize_ds(A, Norm++NormB),
   normalize_ds(B, NormB++Space).

normalize_ds(A, (A+Space)++Space) :-
   atomic(A).
```

O goal `Normalize(Exp, Norm)` tem sucesso se *Norm* é a versão normalizada da expressão *Exp*. Perceba a semelhança entre esse normalizador e o flatten/2: a transformação feita na árvore é essencialmente a mesma. De uma expressão `A+B`, é como se tivessemos normalizado A, normalizado B e, então, concatenado o resultado (como seria uma operação de concatenação de "somas de diferença"?).

Fica como exercício a seguinte questão: qual seria o comportamento esperado nas operações usuais de listas de diferença `Xs\Zs` quando $Xs \subset Zs$ (isto é, quando os elementos de *Xs* pertencem a *Zs* mas alguns de *Zs* podem não pertencer a *Xs*)?

Leituras Adicionais

A pesquisa e prática em programação lógica frequentemente é muito próxima à de linguagens funcionais e esse é o caso na área de transformações de programas. As transformações de *folding/unfolding* foram primeiro adaptadas para a programação lógica por Tamaki e Sato [1]. Mais informações podem ser encontradas em [2].

[1] Tamaki, H. and Sato, T., "Unfold/Fold Transformations of Logic Programs", Proc. Second International Conference on Logic Programming, pp. 127-138, Uppsala, Sweden, 1984.

[2] Roychoudhury, A. and Kumar Narayan K. and Ramakrishnan C.R. and Ramakrishnan I.V., "Beyond Tamaki-Sato Style Unfold/Fold Transformations

for Normal Logic Programs", International Journal of Foundations of Computer Science, World Scientific Publishing Company