```
[paraguai] = •(paraguai, [])
```

Esse exemplo mostra como o princípio geral para a estruturação de objetos Prolog também se aplica a listas de qualquer tamanho. Como o exemplo também mostra, a notação direta com o uso do functor "•" pode produzir expressões bastante confusas. Por essa razão o sistema Prolog oferece uma notação simplificada para as listas, permitindo que as mesmas sejam escritas como seqüências de itens separados por vírgulas e incluídos entre colchetes. O programador pode empregar qualquer notação, entretanto, a que utiliza colchetes é normalmente preferida. Segundo tal notação, um termo da forma [H|T] é tratado como uma lista de cabeça H e corpo T. Listas do tipo [H|T] são estruturas muito comuns em programação não-numérica. Deve-se recordar que o corpo de uma lista é sempre outra lista, mesmo que seja vazia. Os seguintes exemplos devem servir para demonstrar tais idéias:

`[X | Y]` ou `[X | [Y | Z]]` unificam com `[a, b, c, d]`

`[X, Y, Z]` não unifica com `[a, b, c, d]`

```
[a, b, c] == [a | [b | [c]]] == [a | [b, c]] == [a, b | [c]] == [a, b, c | []]
```

As consultas abaixo também são elucidativas:

```
?-[X | Y] = [a, b, c].
X=a Y=[b, c]

?-[X, Y, Z] = [a, b, c, d].
não

?-[X | [Y | Z]] = [a, b, c, d].
X=a Y=b Z=[c, d]
```

5.2 OPERAÇÕES SOBRE LISTAS

Estruturas em lista podem ser definidas e transformadas em Prolog de diversas maneiras diferentes. Na presente seção procura-se, através de uma variedade de exemplos, mostrar a flexibilidade das listas na representação de situações complexas. Emprega-se, para maior clareza, de agora em diante a notação:

```
simbolo_predicativo/aridade
```

para a identificação de predicados. Por exemplo gráfico/3 denota uma relação denominada gráfico com três argumentos. Esse detalhamento é às vezes importante. Nome e aridade são os elementos necessários e suficientes para a perfeita identificação de um predicado.

5.2.1 CONSTRUÇÃO DE LISTAS

A primeira necessidade para a manipulação de listas é ser capaz de construí-las a partir de seus elementos básicos: uma cabeça e um corpo. Tal relação pode ser escrita em um único fato:

```
cons(X, Y, [X | Y]).
```

Por exemplo:

```
?-cons(a, b, Z).
Z=[a | b]
```

Durante a unificação a variável X é instanciada com a, Y com b e Z com [X|Y], que por sua vez é instanciada com [a|b], devido aos valores de X e Y. Se X for um elemento e Y uma lista, então [X|Y] é uma nova lista com X como primeiro elemento. Por exemplo:

```
?-cons(a, [b, c], Z).
Z=[a, b, c]

?-cons(a, [], Z).
Z=[a]
```

A generalidade da unificação permite a definição de um resultado implícito:

```
?-cons(a, X, [a, b, c]).
X=[b, c]
```

Neste último exemplo as propriedades de simetria dos argumentos, lembram um solucionador de equações: um X é encontrado tal que [a|X] = [a, b, c]. Entretanto, se o primeiro argumento for uma lista com, digamos, três elementos e o segundo uma lista com dois, o resultado não será uma lista com cinco elementos:

```
?-cons([a, b, c], [d, e], Z).
Z=[[a, b, c], d, e]
```

de modo que o predicado cons/3 não resolve o problema de concatenar duas listas em uma terceira. Mais adiante será estudado o predicado conc/3 que realiza tal função.

5.2.2 OCORRÊNCIA DE ELEMENTOS EM UMA LISTA

Vamos implementar um tipo de relação de ocorrência que estabelece se determinado objeto é membro de uma lista, como em:

```
membro(X, L)
```

onde X é um objeto e L uma lista. O objetivo membro(X, L) é verdadeiro se X ocorre em L. Por exemplo, são verdadeiros:

```
membro(b, [a, b, c])
membro([b,c], [a, [b, c], d])
```

mas a declaração

```
membro(b, [a, [b, c]])
```

é falsa. O programa que define a relação membro/2 baseia-se na seguinte afirmação:

```
X é membro de L se
       (1) X é a cabeça de L, ou
       (2) X é membro do corpo de L.
```

que pode ser representada em Prolog por meio de duas cláusulas. A primeira, um fato, estabelece a primeira condição: X é membro de L, se X é a cabeça de L. A segunda, uma regra que será empregada quando X não é cabeça de L, é uma chamada recursiva que diz que X ainda pode ser membro de L, desde que seja membro do corpo de L. Em Prolog:

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

Note-se que o corpo da lista na primeira cláusula é sempre um resultado sem qualquer interesse, o mesmo ocorrendo com a cabeça da lista na segunda. É possível então empregar variáveis anônimas e escrever o predicado de forma mais elegante:

```
membro(X, [X | _]).
membro(X, [_ | C]) :-
      membro(X, C).
```

5.2.3 CONCATENAÇÃO DE LISTAS

Para a concatenação de duas listas quaisquer, resultando em uma terceira, se definirá a relação:

```
conc(L1, L3, L3)
```

onde L1 e L2 são duas listas e L3 é a concatenação resultante. Por exemplo:

```
conc([a, b], [c, d], [a, b, c, d])
```

Novamente, dois casos devem ser considerados para a definição de conc/3, dependendo do primeiro argumento L1:

(1) Se o primeiro argumento é uma lista vazia, então o segundo e o terceiro argumentos devem ser

a mesma lista. Chamando tal lista de L, essa situação pode ser representada pelo seguinte fato Prolog:

```
conc([], L, L).
```

(2) Se o primeiro argumento de conc/3 for uma lista não-vazia, então é porque ela possui uma cabeça e um corpo e pode ser denotada por [X|L1]. A concatenação de [X|L1] com uma segunda lista L2, produzirá uma terceira lista com a mesma cabeça X da primeira e um corpo L3 que é a concatenação do corpo da primeira lista, L1, com toda a segunda, L2. Isso pode ser visto na figura 5.2, e se representa em Prolog por meio da regra:

```
conc([X | L1], L2, [X | L3]) :-
      conc(L1, L2, L3).
```

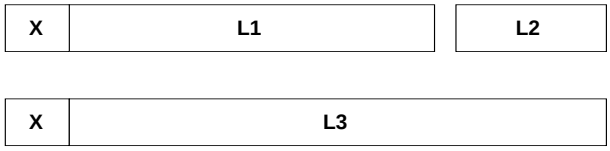

Figura 5.2 Concatenação de duas listas

O programa completo para a concatenação de listas, descrevendo o predicado conc/3 é apresentado a seguir:

```
conc([], L, L).
conc([X | L1], L2, [X | L3]) :-
      conc(L1, L2, L3).
```

Exemplos simples de utilização de tal programa são:

```
?-conc([a, b, c], [1, 2, 3], L).
L=[a, b, c, 1, 2, 3]

?-conc([a, [b, c], d], [a, [], b], L).
L=[a, [b, c], d, a, [], b]

?-conc([a, b], [c | R], L).
L=[a, b, c | R]
```

O programa conc/3, apesar de muito simples, é também muito flexível e pode ser usado em inúmeras aplicações. Por exemplo, ele pode ser usado no sentido inverso ao que foi originalmente projetado para decompor uma lista em duas partes:

```
?- conc(L1, L2, [a, b, c]).
L1=[] L2=[a, b, c];
L1=[a] L2=[b, c];
L1=[a, b] l2=[c];
L1=[a, b, c] L2=[];
não
```

Esse resultado mostra que é sempre possível decompor uma lista de n elementos em n+1 modos, todos eles obtidos pelo programa através de backtracking. Podemos também usar o programa para procurar por um determinado padrão em uma lista. Por exemplo, podemos encontrar os meses antes e depois de um determinado mes:

```
?-M=[jan,fev,mar,abr,mai,jun,jul,ago,set,out,nov,dez], conc(Antes, [mai | Depois], M).
Antes=[jan,fev,mar,abr] Depois=[jun,jul,ago,set,out,nov, dez]
```

e também achar o sucessor e o predecessor imediatos (os vizinhos) de um determinado item da lista:

```
?-conc(_, [X, g, Y | _], [a, b, c, d, e, f, g, h]).
X=f Y=h
```

É possível ainda apagar de uma lista todos os elementos que se seguem a um determinado padrão. No exemplo abaixo, retira-se da lista dos dias da semana a sexta-feira e todos os dias que a seguem.

```
?-conc(Trab, [sex | _], [seg,ter,qua,qui,sex,sab,dom]).
Trab=[seg,ter,qua,qui]
```

A própria relação de ocorrência, membro/2, vista na seção anterior pode ser reprogramada em função de conc/3:

```
membro1(X, L) :-
      conc(_, [X | _], L).
```

Essa cláusula nos diz que X é membro de uma lista L se L pode ser decomposta em duas outras listas onde a cabeça da segunda é X. Na verdade, membro1/2 define a mesma relação que membro/2, apenas adotou-se um nome diferente para estabelecer uma distinção entre ambas.

5.2.4 REMOÇÃO DE ELEMENTOS DE UMA LISTA

A remoção de um elemento X de uma lista L pode ser programada através da relação:

```
remover(X, L, L1)
```

onde L1 é a mesma lista L com o elemento X removido. A relação remover/3 pode ser definida de maneira similar à relação de ocorrência. Novamente são dois casos a estudar:

(1) Se X é a cabeça da lista L, então L1 será o seu corpo;

(2) Se X está no corpo de L, então L1 é obtida removendo X desse corpo.

Em Prolog, isso é escrito da seguinte maneira:

```
remover(X, [X | C], C).
remover(X, [Y | C], [Y | D]) :-
      remover(X, C, D).
```

Assim como a relação membro/2, remover/3 é também não-determinística por natureza. Se há diversas ocorrências de X em L, a relação remove/3 é capaz de retirar cada uma delas através do mecanismo de backtracking do Prolog. Evidentemente, em cada execução do programa remove/3 retiramos somente uma das ocorrências de X, deixando as demais intocáveis. Por exemplo:

```
?-remover(a, [a, b, a, a], L).
L=[b, a, a];
L=[a, b, a];
L=[a, b, a];
não
```

remover/3 irá falhar se a lista L não contiver nenhuma ocorrência do elemento X. Essa relação pode ser ainda usada no sentido inverso para inserir um novo item em qualquer lugar da lista. Por exemplo, pode-se formular a questão: "Qual é a lista L, da qual retirando-se 'a' , obtem-se a lista [b, c, d]?"

```
?-remover(a, L, [b, c, d]).
L=[a, b, c, d];
L=[b, a, c, d];
L=[b, c, a, d];
L=[b, c, d, a];
não
```

De modo geral, pode-se inserir um elemento X em algum lugar de uma lista L, resultando em uma nova lista L1, com o elemento X inserido na posição desejada, por meio da cláusula:

```
inserir(X, L, L1) :-
      remover(X, L1, L).
```

Em membro1/2 foi obtida uma forma alternativa para a relação de ocorrência, utilizando o predicado conc/3. Pode-se obter a mesma relação por meio de remover/3:

```
membro2(X, L) :-
      remover(X, L, _).
```

5.2.5 INVERSÃO DE LISTAS