Na nossa implementação de listas generalizadas, elas são representadas por variáveis do tipo ponteiro, declaradas no próprio texto do programa. Estas variáveis, se declaradas dentro de rotinas, são criadas quando a rotina recebe o controle e destruídas tão logo a rotina termine de executar. Além disto, todos os nodos usados pela estrutura são alocados dinamicamente.

Vimos que áreas de memória alocadas de forma dinâmica isto é, com o comando **new**, não são liberadas automaticamente quando saem do escopo no qual foram criadas. Assim, se uma lista é acessível apenas através de uma variável interna a uma rotina, quando esta termina de executar, a variável ponteiro é destruída automaticamente, deixando inacessível toda a estrutura apontada.

Não é difícil perceber a necessidade de uma rotina capaz de liberar todos os nodos de uma lista generalizada que não é mais desejada! Ao destruir uma lista L, temos três casos a considerar:

- L tem valor nulo, representando uma lista vazia;
- L aponta um nodo terminal, representando um átomo;
- L aponta um nodo não-terminal, representando uma lista generalizada.

Os dois primeiros casos permitem solução trivial: no primeiro, como a lista já está vazia, não há nada a fazer; no segundo, basta liberar o nodo terminal apontado por L. No terceiro caso, como o ponteiro L aponta uma lista generalizada, podemos "dividir" esta lista em cabeça e cauda e destruir estas sub-listas recursivamente:

```
procedure Kill(var L:LstGen);
begin
     if not Null(L) then
        begin
           if Atom(L) then
              begin
                 dispose(L);
                 L := nil;
              end
           else
              begin
                 Kill(L^.cabeca);
                 Kill(L^.cauda);
                 dispose(L);
                 L := nil;
              end;
        end;
end;
```

Garantimos que o algoritmo termina porque, cada vez que seguimos o ponteiro para cabeça, descemos um nível na estrutura e certamente, em algum instante, encontraremos um ponteiro nulo ou um nodo terminal, o que fará a recursão terminar. Analogamente, a cada chamada recursiva que é feita para destruir a cauda, caminhamos para a frente e, conseqüentemente, em algum momento chegaremos a um ponteiro nulo que fará a recursão terminar.

Vamos agora implementar uma rotina capaz de duplicar uma lista generalizada. A rotina irá receber um ponteiro para uma lista e construir uma nova cópia dela na memória. No final do processo, a rotina deverá retornar o endereço do nodo inicial da estrutura construída.

Novamente, teremos os mesmos casos básicos vistos anteriormente. Na verdade, estes casos não dependem do algoritmo que manipula a estrutura, mas da própria forma segundo a qual ela foi definida. Logo, qualquer que seja o algoritmo a ser desenvolvido para manipular uma lista generalizada, devemos sempre considerar estes mesmos casos básicos apresentados.

Caso L seja uma lista vazia, não há o que duplicar e retornamos nulo. Caso L aponte um nodo terminal, duplicamos este nodo e retornamos o seu endereço. Há ainda a possibilidade de L apontar um nodo não-terminal e, considerando que Head(L) e Tail(L) são listas que podem ser duplicadas recursivamente pelo próprio algoritmo, neste último caso, basta construir uma nova lista cuja cabeça seja uma cópia da cabeça de L e cuja cauda seja uma cópia da cauda de L:

```
function Dup(L:LstGen): LstGen;
begin
     if Null(L) then
        Dup := nil
     else
        if Atom(L) then
           Dup:= Crat(L^.obj)
        else
           Dup:= Cons( Dup(Head(L), Dup(Tail(L) );
end;
```

Para finalizar, vamos apresentar um algoritmo capaz de comparar duas listas generalizadas, informando se elas são iguais ou não. A igualdade para listas generalizadas verifica-se quando as duas listas armazenam os mesmos elementos nas mesmas posições; ou seja, as duas listas devem ter a mesma forma e armazenar as mesmas informações para que sejam consideradas iguais:

```
function Equal(L,M:Lista): boolean;
begin
  if Null(L) and Null(M) then
    Equal := true
  else
    if Atom(L) and Atom(M) then
      Equal := (L^.obj = M^.obj)
    else
      if Null(L) or Null(M) then
        Equal := false
      else
        if Atom(L) or Atom(M) then
          Equal := false
        else
           equal := ( equal(Head(L),Head(M)) and
                      equal(Tail(L),Tail(M));
end;
```

Para que duas listas L e M sejam iguais é necessário que ambas sejam vazias, ou então que ambas sejam átomos cujos valores são iguais. Caso estas condições não se verifiquem, o algoritmo testa se acontece de termos uma lista vazia e a outra não, ou então uma "lista" atômica e a outra não e se isto ocorre, então as listas não podem ser iguais. Se não podemos dizer com certeza se as listas são iguais ou não pelos testes já realizados, então podemos lançar mão da solução recursiva, que verifica se a cabeça de uma lista é igual à cabeça da outra e se a cauda de uma lista é igual à cauda da outra...

Observe como o uso de recursão, aliado ao uso das rotinas de manipulação básicas já definidas, permite-nos escrever códigos claros, concisos e simples para manipular as listas generalizadas. Sem a recursão, estas rotinas seriam muito mais complicadas do que podem parecer!