Faculdade de Economia
Universidade de Coimbra

ESTRUTURAS E BASES DE DADOS

Estruturas de dados

Bases de dados

Utilização de dBase IV

Pedro Lopes Ferreira

1992/93

1. Estruturas de dados

1.1 Introdução

Com o aparecimento dos computadores digitais foi possível aumentar-se a velocidade de execução e a complexidade dos algoritmos utilizados. É hoje possível processar uma maior quantidade de informação criando verdadeiros modelos e abstracções do mundo real com as características e as propriedades periféricas dos objectos reais por vezes ignoradas. Num ficheiro de pessoal, cada funcionário pode ser representado abstractamente por um conjunto de dados onde se inclui, por exemplo, o nome e o salário do funcionário, entre outros. No entanto, não incluirá dados tais como a cor dos olhos ou do cabelo. O conjunto dos dados usados constitui portanto, uma abstracção da realidade.

O problema que nos interessa aqui estudar é como escolher uma notação para a descrição de algoritmos e dados. Pretendemos usar uma notação que nos seja familiar, baseada em estruturas matemáticas tais como números, conjuntos e sequências e que ao mesmo tempo possa ser interpretada por uma linguagem de programação.

A linguagem Pascal será a usada neste curso para ilustrar tais estruturas. Nesta linguagem, é usado o conceito de tipo de dados que permite determinar qual o conjunto de valores a que pertence uma constante, uma variável ou uma expressão ou que possa ser gerado por um operador ou uma função. O número de valores distintos que pertencem a um determinado tipo T é denominado cardinal de T.

Na noção de estrutura de dados incluímos não só o conjunto dos elementos como também a maneira como estes estão relacionados, isto é, as operações que podem ser realizadas.

Um dos métodos expeditos para definir quais os valores c_i que são associados a cada tipo T é o da enumeração descrito da forma:

type T = (c_1 , c_2 , …, c_n)

Exemplos:

```
type forma = (rectângulo, quadrado, elipse, círculo)
type cor = (vermelho, amarelo, verde)
type sexo = (feminino, masculino)
type booleano = (falso, verdadeiro)
type destino = (inferno, purgatório, céu)
type veículo = (comboio, automóvel, barco, avião)
type estrutura = (ficheiro, matriz, registo, conjunto)
```

Como em qualquer outra linguagem de programação, a linguagem Pascal aceita os chamados tipos primitivos, englobando os números, os valores lógicos e o conjunto dos caracteres. São normalmente representados pelos identificadores

integer , *real* , *boolean* , *char*

O tipo *integer* compreende o subconjunto dos números inteiros associados a operações exactas. Em cada implementação da linguagem Pascal existe um inteiro *maxint* de tal modo que qualquer inteiro N só pode ser representado no computador se, aproximadamente,

$$-maxint \leq N \leq maxint$$

Os operadores associados aos tipos inteiros são

+ (adição) **-** (subtração) ***** (multiplicação) **DIV** (divisão inteira) **MOD** (módulo)

Usando estes operadores, por exemplo,

```
2 + 3 * 4   =  14
(2 + 3) * 4   =  20
2  DIV  3   =  0
14  MOD  3   =  2
```

Os valores de tipo *real* são usados da mesma maneira que os números reais nas aplicações matemáticas. Como qualquer computador, devido a limitações de equipamento, só pode representar um subconjunto finito de números reais, as operações aritméticas de tipo real dependem do respectivo grau de precisão e de arredondamento.

Os operadores que podem ser usados com operandos reais são

```
+   -   *   /
```

que representam, respectivamente, adição, subtracção, multiplicação e divisão.

Os valores do tipo booleano são os identificadores *true* e*false* associados às operações lógicas de conjunção, união e negação. Além dos operadores booleanos, há também a considerar os operadores relacionais ou de comparação

```
<       menor do que
≤       menor do que ou igual a
=       igual a
≠       diferente de
≥       maior do que ou igual a
>       maior do que
```

Finalmente, o tipo *char* compreende o conjunto dos caracteres definido pela *International Standards Organization* (ISO) e, em especial, pela versão americana ASCII *(American Standard Code for Information Interchange)*, incluindo 95 caracteres de impressão (26 letras, 10 dígitos e 59 caracteres gráficos) e 33 caracteres de controlo.

Os caracteres podem ser ordenados segundo o seu valor ordinal e comparados através dos operadores

$$< \quad \leq \quad = \quad \neq \quad \geq \quad >$$

Assim, c1 < c2 é equivalente a ord(c1) < ord(c2)

Por vezes acontece também que uma variável de um certo tipo apenas toma valores dentro de um determinado intervalo. Nesse caso temos uma variável denominada de tipo limitado de acordo com o formato

```
type T = min .. max
```

onde min e max são os limites do intervalo.

São exemplos de tipos limitados, os seguintes:

```
type ano = 1900 .. 1999
type letra = 'A' .. 'Z'
type dígito = '0' ..'9'
```

No primeiro exemplo é posível fazermos a atribuição a:=1973, o mesmo não acontecendo a a:=1291.

Além do método de enumeração e dos tipos primitivos, a linguagem de programação Pascal utiliza essencialmente quatro tipos principais de estruturas. São eles as matrizes, os registos, os

conjuntos e os ficheiros sequenciais, todos eles denominados tipos estáticos em oposição aos tipos dinâmicos que são gerados durante a execução do programa e que podem mudar de tamanho e de forma. Estes tipos dinâmicos incluem, entre outras, as estruturas listas e árvores.

1.2. Estruturas estáticas de dados

Nesta secção iremos falar das estruturas matriz, registo, conjunto e ficheiro sequencial.

1.2.1 Matrizes

Esta é a estrutura de dados mais vulgar em linguagens de programação. Trata-se de uma estrutura homogénea com todos os componentes do mesmo tipo base T_O e igualmente acessíveis através de um índice I

```
type T = array [I] of T_O
```

Exemplos:

```
type linha = array [1 .. 5] of real

type alfa = array [1 .. 10] of char
```

Sendo dada uma variável x do tipo matriz *(array)*, podemos seleccionar um qualquer elemento desta variável especificando o correspondente índice x[i]. Ainda, como exemplo, consideremos as duas partes de programa que se seguem, ambas destinadas a determinar o menor índice i de uma componente com valor x.

```
var a: array [1..N] of T
i:=0
repeat i:=i+1 until (a[i]=x) or (i=N);
if a[i] ≠ x then "tal elemento não existe em a"
```

```
var a: array [1..N+1] of T;
i:=0; a[N+1]:=x;
repeat i:=i+1 until a[i]=x;
if i > N then "tal elemento não existe em a"
```

Os componentes dos tipos matriz podem também ser estruturados. Uma variável matriz real M cujos componentes são acedidos através de dois índices, um para a linha (i) e outro para a coluna (j), tem o selector M[i,j] e pode ter a declaração

```
M: array[1..10,1..5] of real
```

Programa para calcular o produto c de duas matrizes quadradas a e b de tamanho N:

```
for  i := 1  to  N  do
  for  j := 1  to  N  do
    begin
      c[i,j] := 0;
      for  k := 1  to  N  do
        c[i,j] := c[i,j] + a[i,k] * b[k,j]
    end
```

1.2.2 Registos

O processo mais geral para se obter um tipo estruturado é agrupar os elementos sem quaisquer limitações de tipo, quer sejam de tipos primitivos ou compostos. Um exemplo deste tipo de estrutura é a dos números complexos composta por dois números reais.

Em termos matemáticos, esta noção corresponde à de produto cartesiano. No entanto, como este conceito tem sido usado essencialmente em base de dados para representar, por exemplo, a estrutura de dados do pessoal de uma empresa, recebeu o nome de registo.

Um registo é definido por

```
type T = record
           S1: T1;
           S2: T2;
           ...
           Sn: Tn
         end
```

Vejamos como um registo pode estruturar os dados das variáveis de tipo complexo, data e pessoa.

```
type complexo = record
                  re: real;
                  im: real
                end

type data = record
              dia: 1..31;
              mês: 1..12;
              ano: 1..2000
            end
```

```
type pessoa = record
                apelido: alfa;
                nome: alfa;
                aniversário: data;
                sexo:(masculino, feminino);
                estadocivil:(solteiro,casado,viúvo,divorciado)
              end
```

Rotina para calcular a diferença entre complexos:

```
procedure csub(var a,b,c: complexo);
begin
   c.re := a.re - b.re;
   c.im := a.im - b.im
end
```

Sendo dadas as variáveis z, d e p, respectivamente dos tipos complexo, data e pessoa, os seguintes selectores podem aceder às suas componentes individuais.

```
z . im
d . mês
p . nome
p . aniversário
p . aniversário . dia
```

Os excertos de programa que se seguem pretendem ilustrar o uso de variáveis de tipo registo, determinando o número das pessoas solteiras de sexo feminino existentes na matriz a.

```
var a: array[1..N] of pessoa;
    cont: integer;
cont:=0;
for i:=1 to N do
  if (a[i].sexo = feminino) and (a[i].estadocivil = solteiro)
             then cont:=cont + 1
```

```
var  a: array[1..N] of pessoa;
     cont: integer;
cont:=0;
for i:=1 to N do
    with a[i] do
        if (sexo = feminino) and (estadocivil = solteiro) then
            cont:=cont + 1
```

Por vezes, e para podermos ser mais precisos na representação da realidade, há necessidade em se considerar dois novos tipos como variantes do mesmo tipo

```
type coordenada = record case tipo: (cartesiano, polar) of
                              cartesiano: (x,y: real);
                              polar: (r: real; j: real)
                  end
```

Outras vezes acontece que pretendemos representar dois tipos com componentes parcialmente idênticas

```
type pessoa = record
              apelido: alfa;
              nome: alfa;
              aniversário: data;
              estadocivil: (solteiro, casado, viúvo,divorciado);
              case sexo: (masculino,feminino) of
                    masculino: (peso: real; barba: boolean),
                    feminino: (tamanho: array[1..3] of integer)
              end
```

1.2.3 Conjuntos

A terceira estrutura fundamental de dados aqui apresentada é a estrutura conjunto que consiste numa colecção de objectos do mesmo tipo definida por uma declaração do tipo

```
type T = set of To
```

Exemplos:

```
type conjint = set of 0..30

type conjcar = set of char

var  is: conjint
     cs: conjcar

is: [1,4,9,16,25]
cs: ['+','-','*','/']
```

Os operadores que podem ser definidos entre variáveis deste tipo são * (intersecção ou multiplicação), + (união ou adição), - (diferença) e **in** (pertença).

Vejamos alguns exemplos:

```
type primária = (vermelho, amarelo,azul);
     cor = set of primária;
var c1,c2: cor;

c1 := [vermelho];
c2 := [ ];
c2 := c2 + [succ(vermelho)]
```

Os conjuntos são relativamente rápidos e eficazes e podem ser usados para eliminar testes mais complexos.

Um teste mais simples para

```
if (ch='a')or(ch='b')or(ch='c')or(ch='d')or(ch='z') then S
```

poderá ser

```
if ch in ['a'..'d','z'] then S
```

1.2.4 Ficheiros sequenciais

Em analogia com as estruturas precedentes, o ficheiro sequencial composto por zero ou mais componentes do tipo T_o, pode ser definido pela forma

```
type T = file of To
```

Apenas uma componente de um ficheiro sequencial pode ser acedida de cada vez. Esta é especificada à custa de um indicador de posição. Através de operadores apropriados, podemos fazer corresponder esta posição ao registo seguinte ou ao primeiro de todos.

Um exemplo de uma declaração é

```
type texto = file of char
```

Um ficheiro sequencial é constituído pela justaposição de componentes, podendo encontrar-se num de dois estados: a ser construído (escrita) ou a ser analisado (leitura).

Para se escrever e ler um ficheiro sequencial, utilizam-se instruções semelhantes às seguintes:

```
rewrite(x)
while p do
begin R(v)
    write(x,v)
end
```

```
reset(x)
while ¬ eof do
    begin read(x,v); S(v)
end
```

1.3 Estruturas dinâmicas de dados

Na secção anterior introduzimos (revimos) as estruturas matriz, registo e conjunto. São denominadas estruturas fundamentais pois constituem os alicerces principais nos quais outras estruturas mais complexas se baseiam. São também chamadas estruturas estáticas pois o seu enquadramento de memória é fixado no início e não pode ser alterado.

No entanto, há muitos programas que envolvem estruturas de informação bem mais complexas e alteráveis durante a computação. São as chamadas estruturas dinâmicas que irão ser tratadas nesta secção.

1.3.1 Tipos de dados recursivos

Uma das propriedades mais interessantes dos procedimentos em Pascal, é sem dúvida, a recursividade. Os valores de um tipo de dados recursivo contêm um ou mais componentes pertencentes ao mesmo tipo que ele próprio. Tal como nos procedimentos, a recursividade dos tipos de dados pode ser directa ou indirecta.

Uma expressão pode ser um bom exemplo para a recursividade. De facto, por definição, uma expressão é um termo seguido de um operador, seguido de outro termo. Um termo, por outro lado, é uma variável ou, de novo, uma expressão entre parêntesis.

```
type expressão = record
                   op: operador;
                   opd1,opd2: term
                 end;
type termo = record
               if t then (id: alfa)
                  else (subexp: expressão)
             end
```

Um outro exemplo, agora de recursividade directa, é o caso da árvore genealógica de uma família. Cada nodo da árvore genealógica é definido pelo nome da pessoa e pela árvore genealógica dos pais, o que nos leva necessariamente a uma estrutura infinita. Na prática isto não acontece porque a informação começa a falhar à medida que recuamos no tempo.

```
type pessoa = record
                if conhecido then (nome: alfa;
                                   pai, mãe: pessoa )
              end
```

3.2 Ponteiros

Na impossibilidade de se pré-atribuir uma quantidade fixa de memória a estruturas recursivas, a técnica de memória dinâmica utiliza os chamados ponteiros. Todos os tipos de dados que discutimos até agora apenas continham dados. Os ponteiros contêm outro tipo de informação - um endereço. Diagramaticamente, o exemplo da árvore genealógica pode ser ilustrado do seguinte modo, onde F indica a ausência de mais componentes.

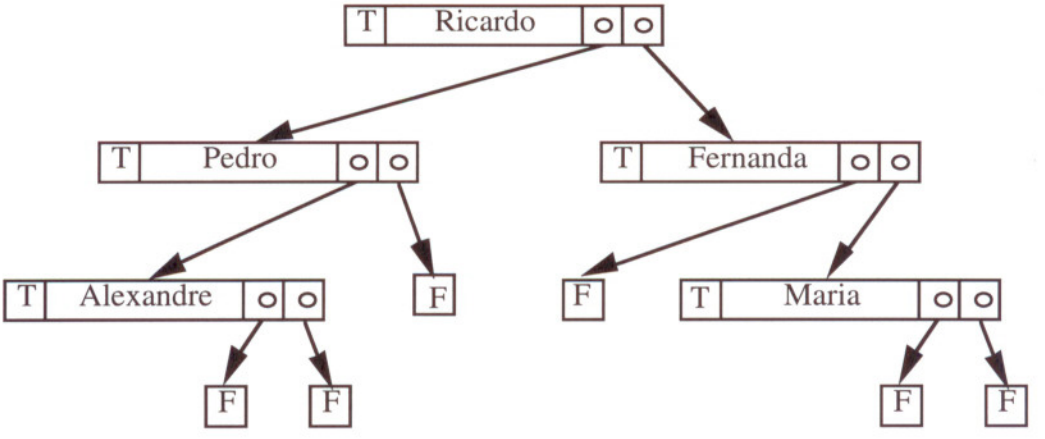


Figura 1.1 - Representação de uma árvore genealógica

Um ponteiro, em vez de conter dados, é uma variável que contém um endereço de memória onde os dados serão armazenados. Como se vê, é necessário distinguir-se os dados dos ponteiros (para outros dados). A notação que usamos para esse fim é a seguinte

```
type Tp = ^ T
```

Esta declaração especifica que os valores do tipo T_p são ponteiros de dados do tipo T. Para se especificar um endereço, e por conseguinte, para colocar dinamicamente um dado em memória, utilizamos a instrução

```
new (p)
```

Sendo dada uma variável ponteiro p do tipo T_p , esta instrução gera um ponteiro do tipo T_p referenciando esta nova variável e atribuindo este ponteiro à variável p. Por outro lado, a variável que é referenciada por p é denotada por p^.

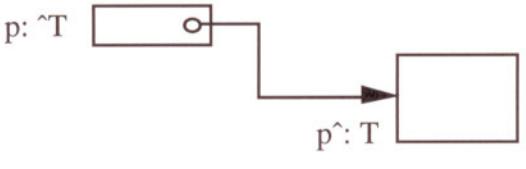


Figura 1.2 - Mecanismo de ponteiro

Um elemento que pertence a todos os tipos ponteiro é o símbolo **nil** que significa que esse elemento não aponta para qualquer outro. Os tipos de dados recursivos atrás mencionados poderão ser escritos do seguinte modo:

```
type expressão = record
                    op: operador;
                    opd1,opd2: ^term
                 end;

type termo = record
               if t then (id: alfa)
                    else (subexp: ^expressão)
             end

type pessoa = record
                nome: alfa;
                pai, mãe: ^pessoa
              end
```

A estrutura correspondente à árvore genealógica atrás apresentada poderia ser

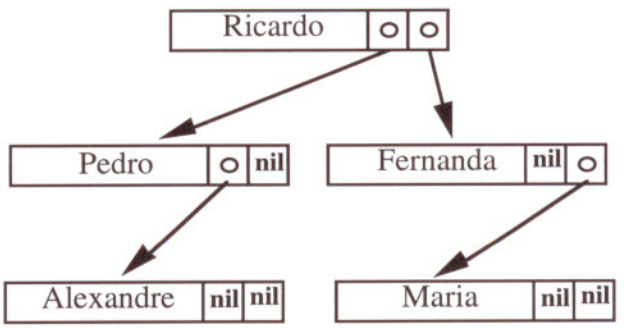


Figura 1.3 - Utilização de **nil** na representação de uma árvore genealógica

A parte restante desta secção é dedicada à geração e manipulação de estruturas de dados cujas componentes estão ligadas por ponteiros. São elas as pilhas e filas de espera, as listas ligadas e as árvores.

1.3.3 Listas lineares

O processo mais simples de ligar um conjunto de elementos é alinhá-los uns a seguir aos outros formando uma lista. Para cada novo elemento, apenas necessitamos de uma ligação ao elemento precedente. Podemos usar a declaração

```
type T = record
            chave: integer;
            seg: ^T;
            ...
         end
```

onde cada variável deste tipo é composta por três componentes: uma chave identificadora, um apontador para o seu sucessor e eventualmente outra informação qualquer.

Para que se possa aceder a esta estrutura de lista temos de criar um ponteiro p apontado para o primeiro elemento (cabeça) da lista.

Figura 1.4 - Lista linear

Os elementos novos são colocados à cabeça da lista, o que implica necessariamente a existência de uma lista vazia para iniciar todo o processo.

```
p := nil;
while n>0 do
  begin      new(q);  q^.seg := p;  p:=q;
       q^.chave := n;  n := n-1
  end
```

Se pretendermos inserir o elemento designado pelo ponteiro q a seguir ao elemento designado pelo ponteiro p, procedemos às seguintes atribuições de valor:

```
q^.seg := p^.seg;  p^.seg := q
```

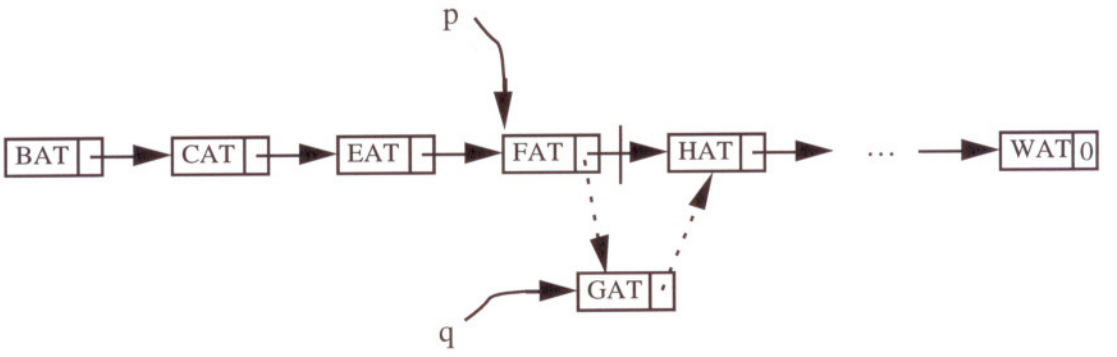


Figura 1.5 - Inserção de elemento numa lista linear

Para eliminar o elemento apontado por ^p, limitamo-nos a reatribuir ponteiros

```
r := ^p.seg;  ^p.seg := ^r.seg;
```

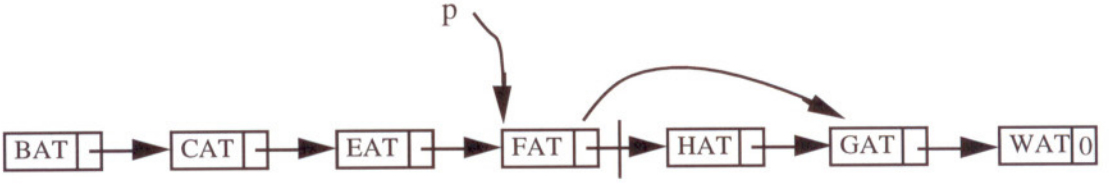


Figura 1.6 - Eliminação de elemento numa lista linear

Uma outra operação frequente é a pesquisa, na lista, de um elemento com uma determinada chave x. Tal como nos ficheiros, esta pesquisa é puramente sequencial, estando concluída quando o elemento for encontrado ou quando atingirmos o fim da lista. Partindo da hipótese de que a cabeça da lista é designada pelo ponteiro p, a parte de programa que se segue, pesquisa um elemento x.

```
while p ≠ nil do
  if ^p.chave = x then "encontrei"
                  else p := ^p.seg
```

1.3.3.1 Pilhas

Dentro da categoria das listas lineares, uma das mais importantes subclasses é a família das estruturas de pilha. Iremos estudar alguns conceitos, assim como os algoritmos de inserção e eliminação.

As pilhas são tão frequentemente usadas que são normalmente estudadas ao nível das estruturas mais simples. Como fazem parte da classe das listas ordenadas (e porque um vector pode ser considerado como uma lista ordenada) são muitas vezes representadas através de uma estrutura *array* .

Sendo dada uma pilha $S=(a_1,\ldots,a_n)$ chamamos a a_n o último elemento e a a_1 o primeiro elemento da lista. O elemento a_i está em cima do elemento a_{i+1}, para $1 < i \leq n$.

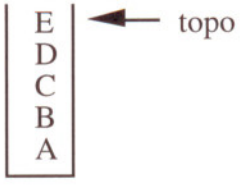


Figura 1.7 - Representação esquemática de uma pilha

As restrições numa pilha requerem que os elementos A, B, C, D e E sejam adicionados à pilha por esta ordem. O último elemento a ser inserido numa pilha será o primeiro a ser removido, sendo por esta razão as pilhas por vezes chamadas listas LIFO (last-in, first-out) ou LCFS (last-come, first-served). Como analogia ao conceito de pilha podemos considerar uma pilha de guardanapos de alguns restaurantes e um sistema de desvio de composições de caminho de ferro. Neste sistema, o último comboio a ser colocado na pilha é o primeiro a sair.

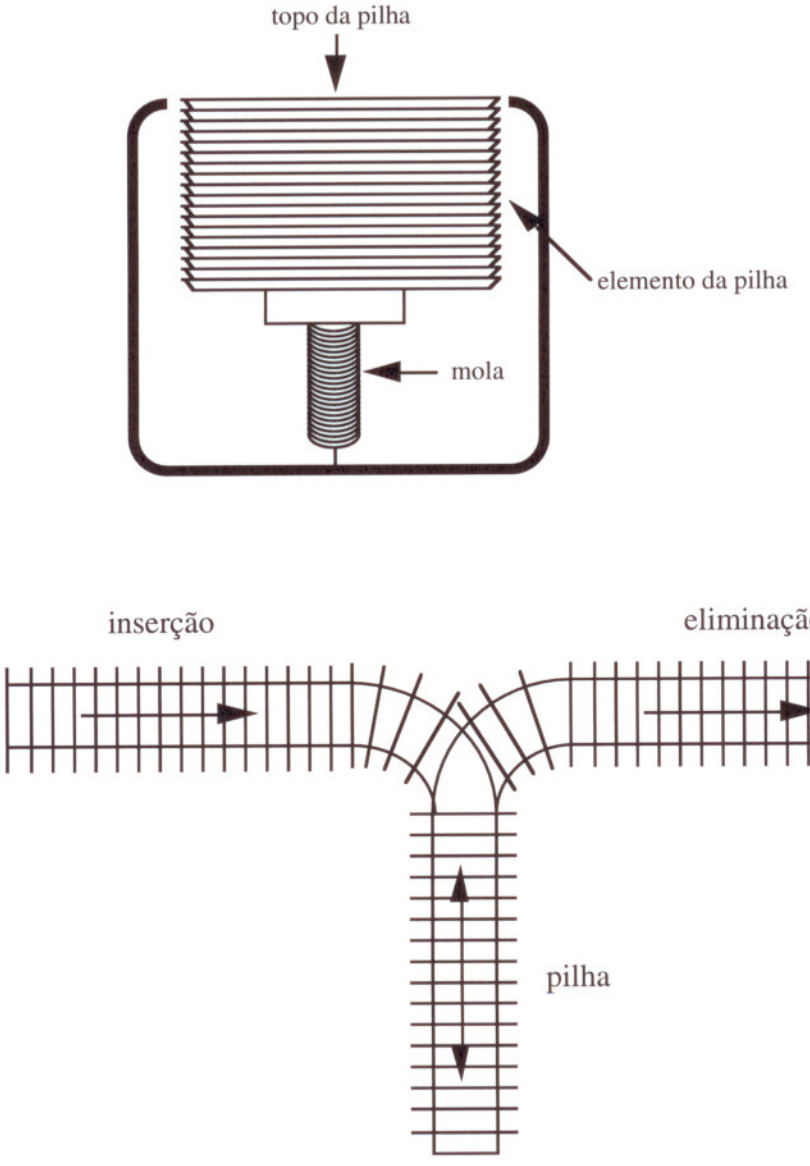


Figura 1.8 - Analogias físicas do conceito de pilha

O uso repetido de operações de inserção e de eliminação permite alterar a ordem dos elementos de dentro de uma pilha. Utilizando a terminologia das pilhas, a inserção e a eliminação de elementos são normalmente indicadas por *push* e *pop*, respectivamente.

Uma das aplicações mais vulgares das pilhas é a recursividade, já anteriormente mencionada. Lembremo-nos para isso que a função factorial n! é definida do seguinte modo:

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n\,(n-1) & \text{se } n > 0 \end{cases}$$

A seguinte função calcula a função factorial:

```
function fact (n: integer): integer;
begin
  if n=0 then fact := 1 else fact := n * fact(n-1)
end
```

O cálculo de factorial de 4, por exemplo, exige as seguintes chamadas recursivas à função fact:

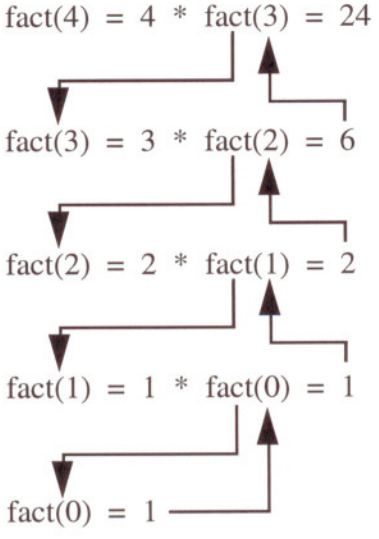


Figura 1.9 - Execução da função factorial

Na execução desta função, usa-se uma pilha S para armazenar um registo associado a cada chamada recorrente. Cada registo temporário contém o valor n (PARAM) e o endereço de retorno (ENDEREÇO). De início, PARAM contém o valor de n e ENDEREÇO contém o endereço da instrução **end** da função. O algoritmo utilizado é o seguinte:

```
1. push(PARAM,ENDEREÇO)
2. se n=0 então fact = 1;  vai para passo4
          senão PARAM = n-1; ENDEREÇO = passo3; vai para passo1
3. fact = n * fact
4.  pop(PARAM,ENDEREÇO);  vai para ENDEREÇO
```

1.3.3.2 Filas de espera

Uma subclasse importante das listas lineares é a família das filas de espera com a característica comum de permitirem a eliminação de elementos do início da lista, sendo as inserções realizadas no fim da lista. Se tivermos a fila de espera

Figura 1.10 - Representação esquemática de uma fila de espera

A será o primeiro elemento a ser removido sendo, por essa razão, as filas de espera normalmente chamadas listas FIFO (first-in, first-out) ou FCFS (first-come, first-served).

Exemplos de filas são as listas de espera de programas num sistema operativo em multiprogramação ou as filas nos bancos ou cruzamentos.

A estrutura fila de espera pode ser representada através de um *array* ou de uma lista linear.

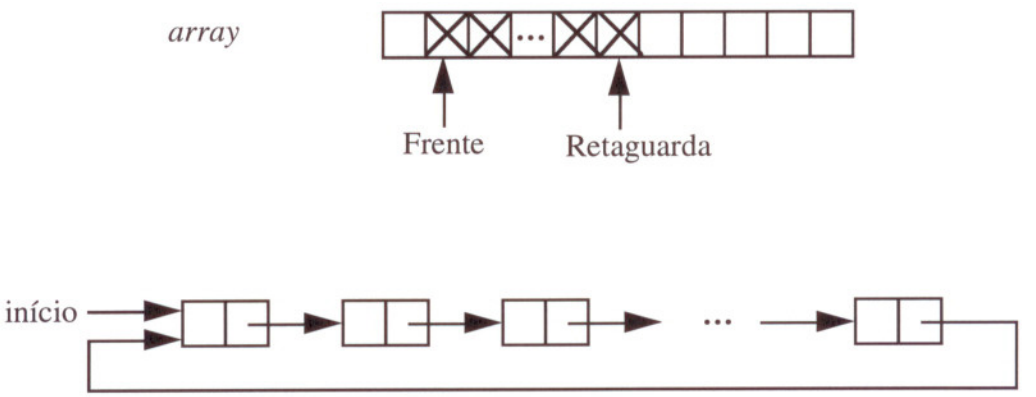


Figura 1.11 - Representação de uma fila de espera em termos de um *array* e de uma lista linear

Nesta última situação, os procedimentos de inserção e eliminação são os comuns a qualquer lista. No caso da representação através de um *array* Q, os procedimentos a seguir destinam-se a inserir e a eliminar elementos. Notar que o elemento Q[1] segue o elemento Q[N]. F e R representam os ponteiros de frente e de retaguarda, tendo uma fila vazia ambos os valores iguais a zero.

```
procedure inserção_lista_circular (var Q: vector; N: integer;
                                   var F,R: integer; X: integer);
begin
  if  R = N  then  R := 1  else  R := R + 1;      ajustar R se necessário
  if  F = R  then  call Q_over;                   condição de overflow
  Q(R) := X;
  if  F = 0  then  F := 1                         ajustar ponteiro F
end
```

```
procedure elimina_lista_circular (var Q: vector; N: integer;
                                  var F,R, X: integer);
begin
  if  F = N  then  call Q_under;                    condição de underflow
  X := Q(F);                                        elimina elem. da cabeça
  if  F = R  then  begin  F := 0;  R := 0  end      verificar se fila vazia
              else  begin                           incrementar pont.frente
                      F := F + 1;
                      if  F > N  then  F := 1
                    end
end
```

1.3.4 Árvores

Até agora temos limitado o nosso estudo a estruturas de dados lineares. Nesta secção examinaremos alguns detalhes associados a estruturas de dados em árvore. Uma árvore é uma estrutura de dados composta por registos ligados por ponteiros na qual cada registo pode conter ponteiros para outros registos. Intuitivamente, uma estrutura em árvore significa que os dados estão organizados de tal modo que os itens de informação estão relacionados.

Formalmente, uma árvore é um conjunto finito de um ou mais nodos de tal modo que:

(i) existe um nodo específico chamado raiz;

(ii) os restantes nodos são repartidos em $n \geq 0$ conjuntos disjuntos $T_1, \ldots T_n$ onde cada um destes conjuntos é uma árvore. $T_1, \ldots, T_n$ são chamadas subárvores da raiz.

Alguns termos devem ser definidos. Um nodo da árvore compreende um item de informação mais os ramos para outros itens. O número de subárvores de um nodo é chamado o seu grau.

Na figura seguinte temos uma representação de uma árvore cuja raiz é A, sendo os nodos D, E, F, G, I, J e K, os nodos terminais ou folhas da árvore. D, E e F são filhos de B e, por exemplo, C é pai de G e de H.

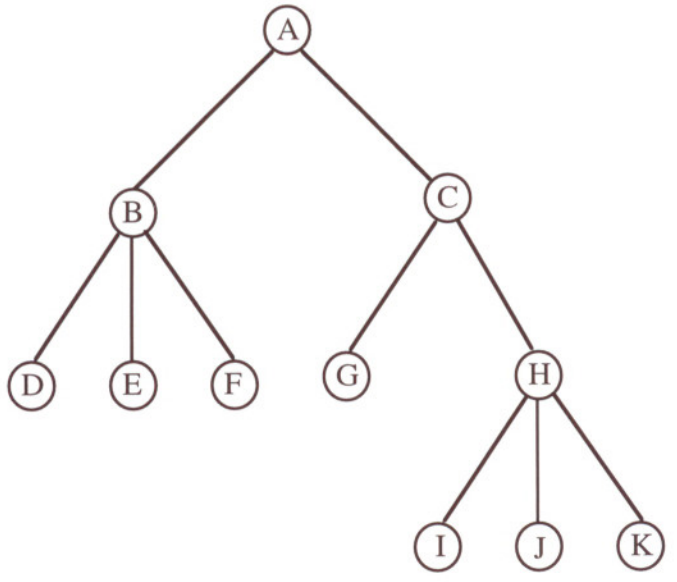


Figura 1.12 - Representação esquemática de uma árvore

Existem outras formas de desenhar uma árvore. Uma delas é a lista, podendo a árvore da figura anterior ser escrita do seguinte modo:

(A (B (D . E . F) . C . (G . H (I . J . K))))

A informação sobre o nodo raiz vem antes da informação referente às subárvores saídas desse nodo. Diagramaticamente, temos

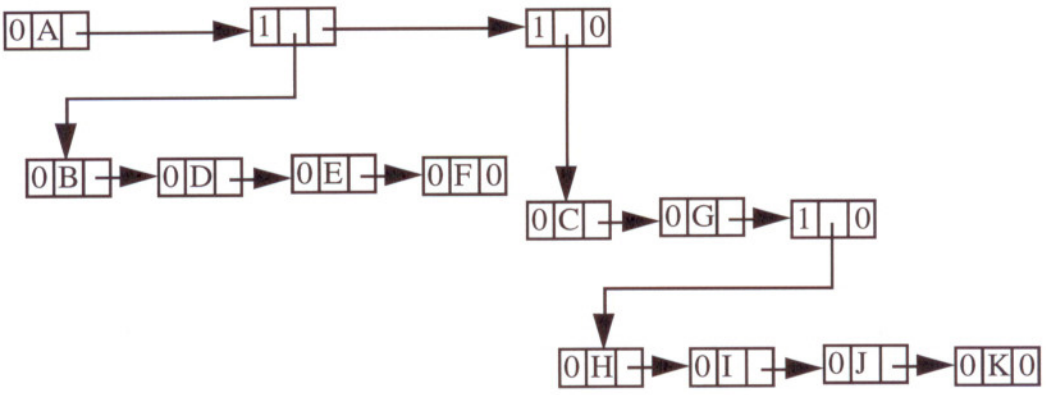


Figura 1.13 - Representação de uma árvore através de listas

As árvores podem também ser encaradas de uma forma recursiva. De facto, uma árvore pode ser vazia ou um nodo contendo apontadores para árvores disjuntas. Portanto, uma árvore binária (com, no máximo, dois ramos partindo de cada nodo)

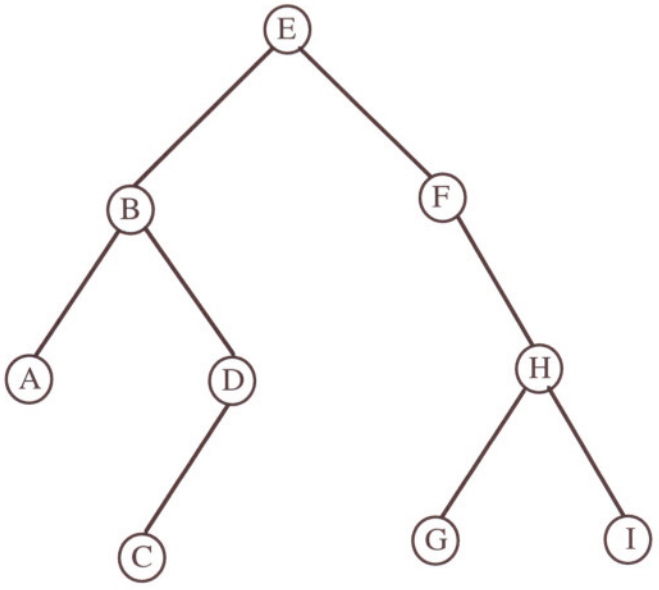


Figura 1.14 - Árvore binária

pode ser representada em Pascal pelo registo

```
type
  ligação := ^nodo;
  nodo = record
           esquerda, direita: ligação;
           dados: tipodedados
         end
```

De entre os procedimentos possíveis com uma estrutura em árvore, um dos mais comuns é a sua travessia, isto é, a visita segundo uma ordem previamente especificada. Há essencialmente três ordens que aparecem naturalmente de uma estrutura em árvore. Usando como referência a árvore binária

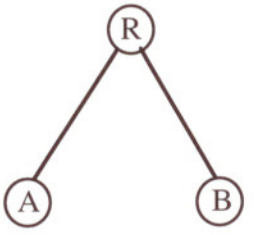


Figura 1.15 - Árvore binária simples

R indica a raiz e A e B representam as subárvores filhas, respectivamente, da esquerda e da direita. As três ordens são as seguintes: (1) preordem (R,A,B); (2) inordem (A,R,B); e (3) postordem (A,B,R).

De uma maneira informal podemos dizer que na preordem visitamos um nodo, atravessamos para a esquerda e continuamos. Quando não podermos continuar mais, movemos para a direita e começamos de novo ou andamos para trás até nos podermos mover para a direita e continuar.

Na inordem movemos para baixo na árvore na direcção da esquerda até não podermos ir mais. Visitamos então o nodo, movemos um nodo para a direita e continuamos. No caso de já não podermos mover mais para a direita, recuamos um nodo.

Finalmente na postordem visitamos um nodo, atravessamos para a direita e procedemos de uma maneira equivalente às anteriores.

Para exemplificar, consideremos a árvore que representa a expressão (a+b/c)*(d-e*f).

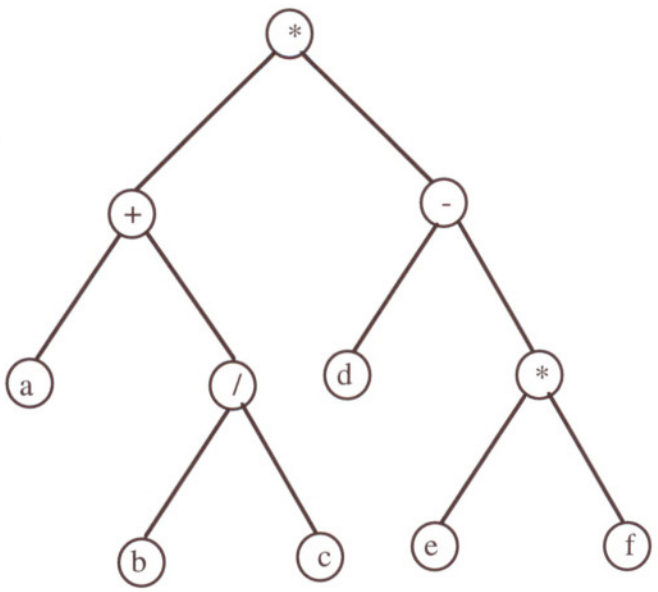


Figura 1.16 - Árvore correspondente à expressão (a+b/c)*(d-e*f)

Dependendo da ordem usada, a esta árvore corresponderão as seguintes ordens:

1. Preordem: * + a / b c - d * e f
2. Inordem: a + b / c * d - e * f
3. Postordem: a b c / + d e f * - *

Se considerarmos a árvore binária T onde cada nodo tem três campos (FILHO_ESQ, DADOS e FILHO_DIR), os procedimentos seguintes percorrem a árvore segundo as três ordens.

```
procedure PREORDEM(T)
if  T ≠ 0  then  print (DADOS(T))
                 PREORDEM(FILHO_ESQ(T))
                 PREORDEM(FILHO_DIR(T))
end PREORDEM
```

```
procedure INORDEM(T)
if  T ≠ 0  then  INORDEM(FILHO_ESQ(T))
                 print (DADOS(T))
                 INORDEM(FILHO_DIR(T))
end INORDEM
```

```
procedure POSTORDEM(T)
if  T ≠ 0  then  POSTORDEM(FILHO_ESQ(T))
                 POSTORDEM(FILHO_DIR(T))
                 print (DADOS(T))
end POSTORDEM
```

O algoritmo para inserir um elemento numa árvore inordem (ordenada lexicograficamente) é o seguinte:

```
1 Se a árvore não contém nodos, associar o novo nodo como nodo raiz na
  árvore e sair.
2 Comparar o novo nome com o nome do nodo raiz.
  Se o novo nome é lexicograficamente menor do que o nome do nodo raiz,
        então se a subárvore esquerda é não vazia,
                então repetir passo2 na subárvore esquerda;
              senão, associar o novo nome como uma folha esquerda à
                     árvore presente e sair.
         senão, se a subárvore direita é não vazia,
                     então repetir passo2 na subárvore direita
                     senão, associar o novo nome como folha direita.
```

Exemplo: Inserir os elementos N , R , J , B , L , P , K , M.

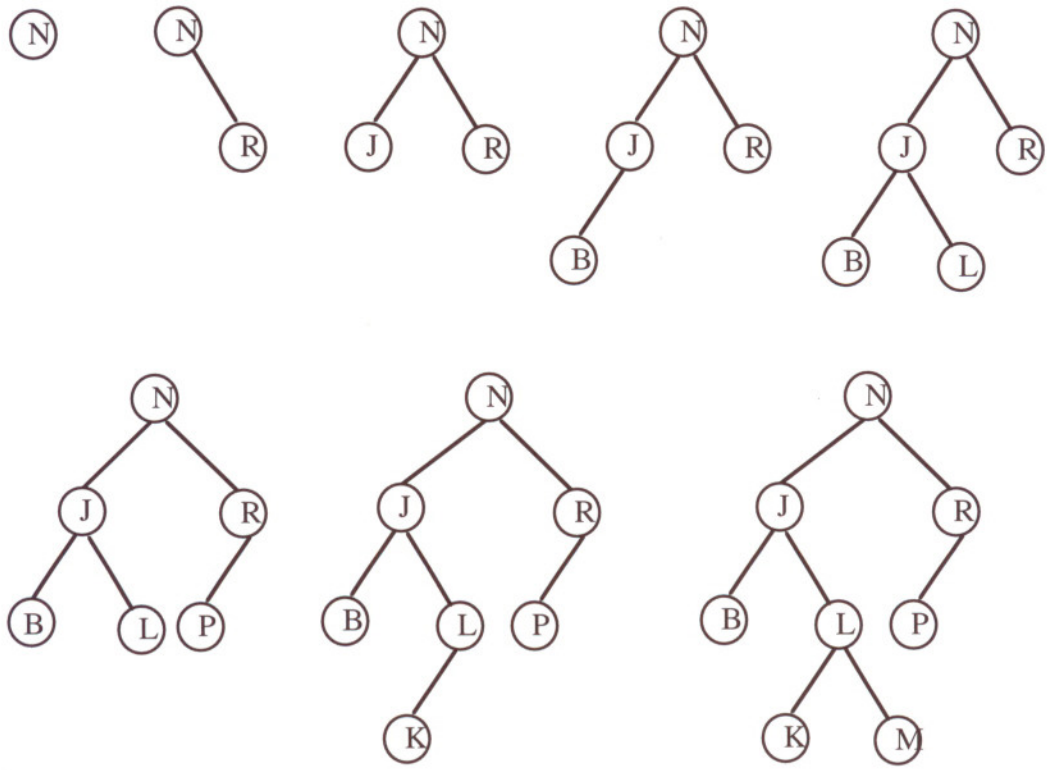
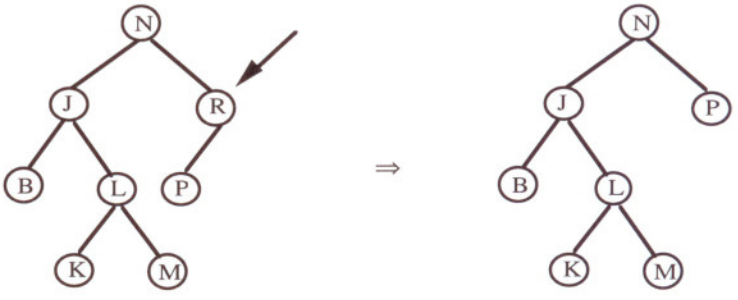


Figura 1.17 - Inserção de elementos numa árvore

O algoritmo para eliminar um elemento numa árvore inordem (ordenada lexicograficamente) é o seguinte:

```
1 Determinar o nodo pai do nodo a apagar, caso exista
  (só não existe quando estamos a apagar a raiz).
2 Se o nodo que acaba de ser apagado tem uma subárvore esquerda ou
  direita, então associar esta subárvore não vazia ao seu nodo avô (o
  nodo encontrado no passo1) e sair.
3 Obter o sucessor inordem do nodo a ser apagado.
  Associar a subárvore direita deste nodo sucessor ao seu avô.
  Substituir o nodo a ser apagado, pelo seu sucessor inordem.
        Isto é acompanhado associando as subárvores direitas e
        esquerdas (com o nodo  sucessor) do nodo marcado para
        eliminação, ao nodo sucessor.
        O nodo sucessor é também associado ao pai do nodo apagado (isto
        é, o nodo obtido no passo1).
```

Exemplo: Eliminar o elemento R.



Exemplo: Eliminar o elemento K.

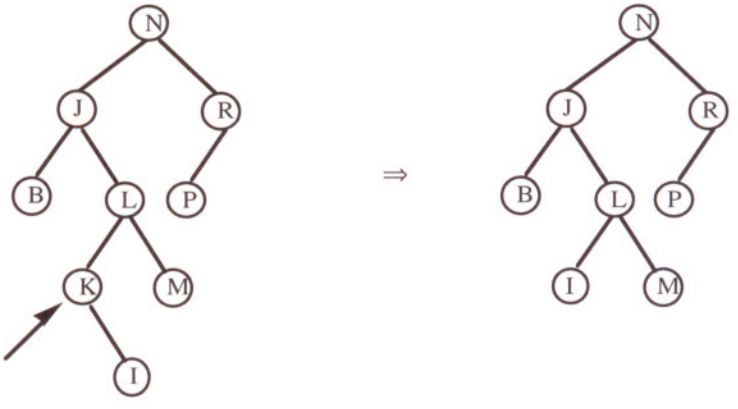


Figura 1.18 - Eliminação de elementos numa árvore

2. Base de dados

2.1 Conceitos básicos

2.1.1 Introdução

As bases de dados e a tecnologia de base de dados têm produzido um grande impacto no cada vez maior domínio dos computadores no nosso dia a dia. O termo "base de dados" é tão frequentemente usado que será melhor começarmos por defini-lo. Uma base de dados é uma colecção de dados logicamente coerentes e com um significado. Representando aspectos do mundo real, qualquer base de dados é desenhada, construída e povoada por registos com objectivos determinados.

Encontramos, na realidade, diversas categorias de informação a que damos o nome de entidades. Uma entidade pode ser uma pessoa, local, coisa, acontecimento ou conceito acerca do qual a informação é armazenada. A cada entidade, no entanto, podemos fazer corresponder várias propriedades, ou atributos, sendo qualquer associação entre entidades denominada uma relação.

As bases de dados podem ter vários tamanhos e vários tipos de complexidade. Podem também ser geradas e mantidas manualmente ou por intermédio de uma máquina, sendo apenas as informatizadas o objecto do nosso estudo.

Chamamos sistema de gestão de base de dados (SGBD) a uma colecção de programas que permite aos seus utilizadores criar e manter uma base de dados. Definir uma base de dados envolve a especificação dos tipos de dados a armazenar numa base de dados, assim como uma descrição detalhada de cada tipo de dados. Construir uma base de dados é o processo de armazenar os próprios dados num determinado suporte de dados controlado pelo SGBD, Manipular uma base de dados inclui funções tais como pesquisar a base de dados para obter um determinado dado, actualizar a base de dados para reflectir actualizações na vida real, e gerar relatórios a partir dos dados.

Ao conjunto formado pela base de dados com o software damos o nome de sistema de base de dados (Figura 2.1).

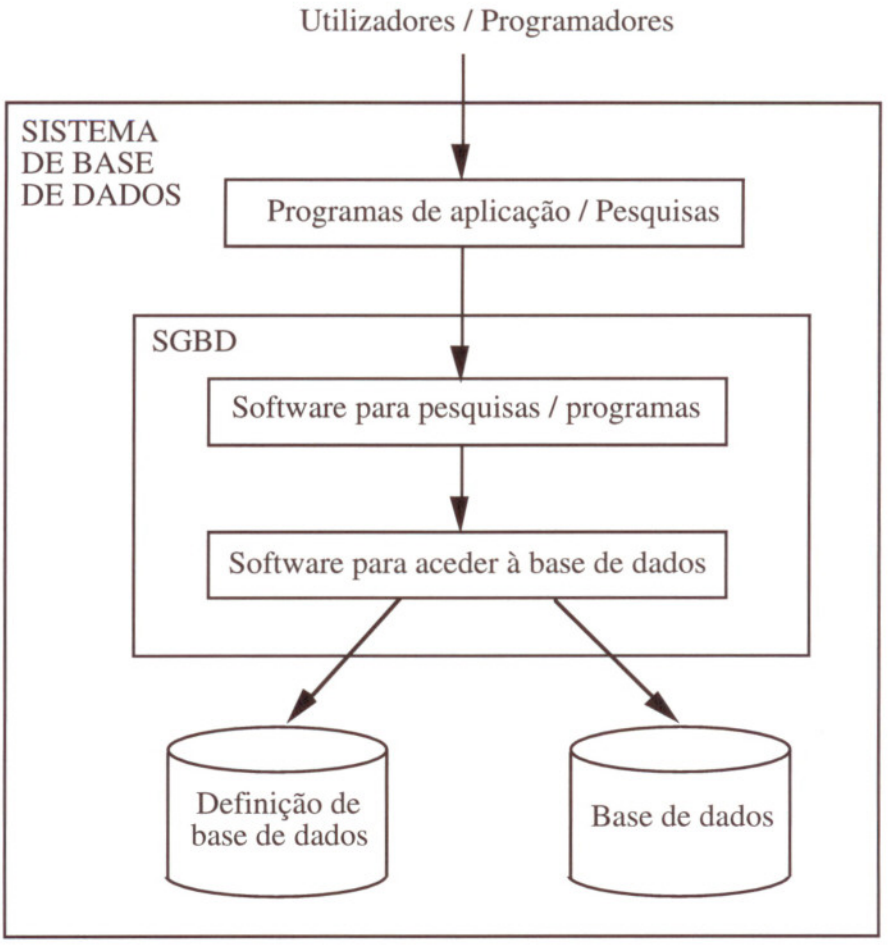


Figura 2.1 - Sistema de Gestão de Base de Dados

2.1.2 Base de dados vs. processamento de ficheiros

É possível distinguir-se a abordagem de base de dados da abordagem tradicional do processamento de ficheiros. Num processamento de ficheiros, o utilizador define e implementa os ficheiros necessários a uma determinada aplicação. Há uma consequente redundância em termos de definição e armazenamento de dados, desperdiçando espaço em memória e esforço para manter os dados actualizados. Ainda mais importante, é que os ficheiros que representam

os mesmos dados se podem tornar inconsistentes, sempre que uma actualização se processe num ficheiro e não noutros.

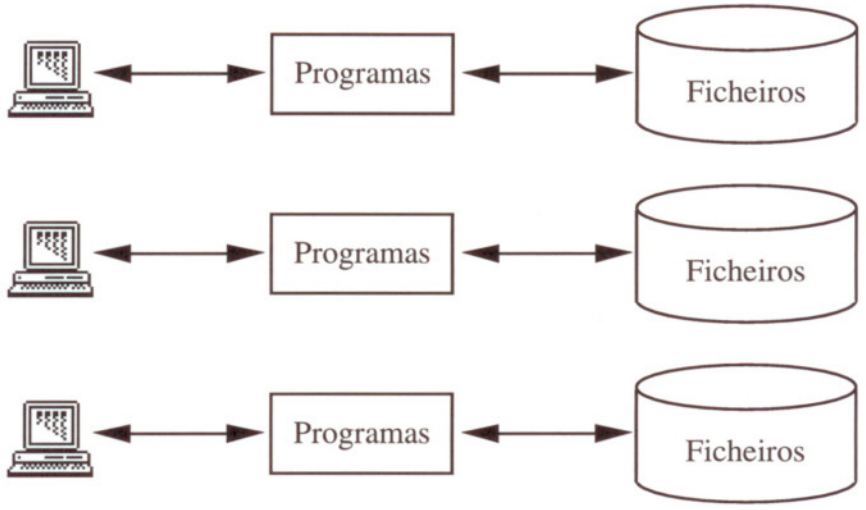


Figura 2.2 - Abordagem de Processamento de Ficheiros

Na abordagem de base de dados, os dados são definidos uma só vez e acedidos por vários utilizadores.

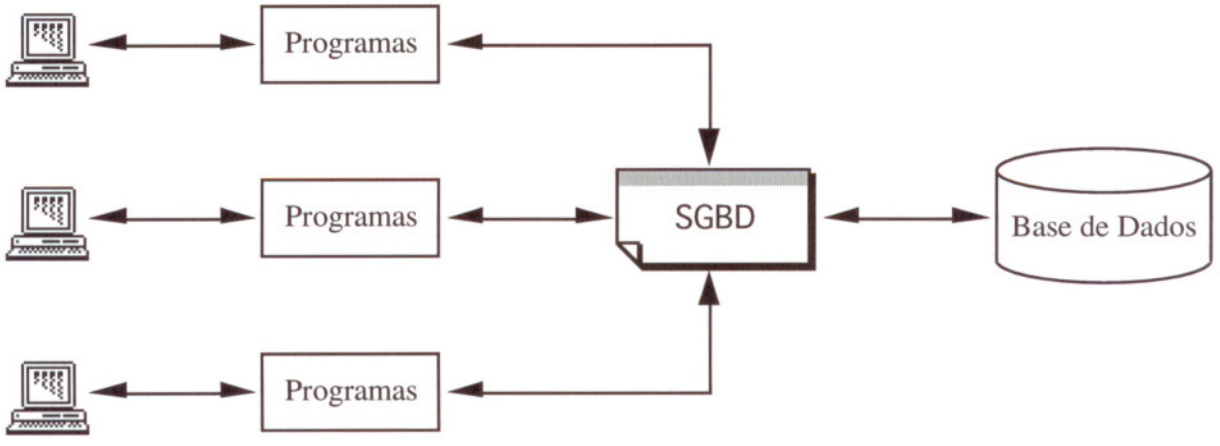


Figura 2.3 - Abordagem de Base de Dados

Vejamos, de seguida, algumas características das bases de dados que as distinguem do processamento tradicional de ficheiros.

- *Natureza do sistema de base de dados*

  Uma característica fundamental da abordagem de base de dados é que o sistema de base de dados contem não apenas a própria base de dados mas também uma definição ou descrição completa da própria base de dados. A informação referente à estrutura de cada ficheiro, ao tipo e ao formato de armazenamento de cada item de dados a às várias restrições impostas nos dados está inserida no chamado catálogo ou dicionário de dados do sistema. Um dicionário de dados é um repositório central de informação relativo às entidades: os dados que representam as entidades, as relações entre as entidades, as suas origens, significados, usos e formatos de representação.

  O sistema de gestão de base de dados não é escrito para nenhuma aplicação em particular e deve funcionar igualmente para qualquer aplicação da base de dados. No processamento tradicional de ficheiros, a definição dos dados faz parte dos programas de aplicações e estes programas são escritos para trabalhar com conjuntos de dados previamente especificados.

- *Independência entre programas e dados*

  No processamento tradicional de ficheiros, a estrutura dos ficheiros de dados está embebida nos programas de tal modo que uma alteração na estrutura de um ficheiro exige alterações em todos os programas que acedem a esse ficheiro.

  Por outro lado, os programas que acedem a um SGBD são escritos independentemente dos ficheiros de dados. A estrutura destes é armazenada no SGBD independentemente dos programas de acesso, sendo assim obtida a chamada independência programa-dados.

- *Abstracção dos dados*

  Num SGBD possuímos sempre uma representação conceptual dos dados. Chamamos modelo de dados a uma abstracção de tipo de dados usada para fornecer esta representação conceptual. O modelo de dados utiliza conceitos lógicos tais como objectos, propriedades e inter-relações, tornando-se assim mais fácil de entender a estrutura de dados, pois os detalhes de armazenamento físico são ignorados. O modelo conceptual representa as entidades e as suas relações.

  Numa base de dados, os detalhes de estrutura e organização de cada ficheiro são armazenados no catálogo. Vários tipos de modelos de dados serão objecto de estudo mais adiante.

- *Várias perspectivas dos dados*

  Uma base de dados normalmente tem vários utilizadores, possuindo cada um deles uma perspectiva eventualmente diferente dos dados, com variáveis provenientes da base de dados e outras externas. Para gerir todas estas perspectivas diferentes, o SGBD tem de conter software de controlo de concorrência para garantir que os vários utilizadores actualizam os mesmos dados controladamente.

- *Procedimentos de salvaguarda e recuperação*

  Falhas de hardware e de software são uma ameaça para as bases de dados. Assim, um SGBD contem normalmente um subsistema de salvaguarda (backup) e recuperação (recovery). Se o sistema informático falha no meio da execução de uma programa de actualização, o subsistema de segurança garante-nos que podemos restaurar o sistema para a situação anterior à actualização. Por outro lado, o subsistema de recuperação garante-nos que o programa volta ao ponto onde foi interrompido.

Além destas, há ainda outras vantagens das bases de dados.

- *Estabelecimento de eventuais standards*

  A abordagem de base de dados permite aos responsáveis pelo sistema criar critérios comuns aos vários utilizadores, o que permite facilitar a comunicação e a cooperação entre os vários componentes da organização. Estes critérios comuns podem incidir, entre outros, no nome e no formato dos dados, na forma de apresentação dos dados, e nas estruturas dos relatórios.

- *Flexibilidade*

  Por vezes, há necessidade em se alterar a estrutura de uma base de dados. Isto deve ser possível efectuar-se sem afectar os programas de aplicações já existentes.

- *Tempo para desenvolvimento*

  Uma das características mais importantes da abordagem de base de dados é estar associada à não necessidade de dispendermos muito tempo no desenvolvimento de novas aplicações. Usando um SGBD, o tempo para desenvolver uma nova aplicação é estimado em cerca de 1/6 a 1/4 do tempo necessário quando se utiliza um sistema tradicional de ficheiros.

- *Disponibilidade de informação actualizada.*

  Logo que um dado é actualizado por um utilizador, torna-se imediatamente disponível a qualquer outro utilizador.

- *Economia de escala*

  Qualquer SGBD permite uma consolidação entre dados e aplicações, reduzindo a sobreposição entre actividades de processamento de dados e portanto, uma maior rentabilidade das actividades a processar numa organização.

2.1.3 Pessoas envolvidas numa base de dados

Em bases de dados de grandes dimensões, há várias pessoas envolvidas no seu desenho, uso e manutenção e com tarefas especialmente definidas. De entre elas há a destacar as seguintes:

- *Administrador de base de dados*

  É responsável pela definição do acesso à base de dados e pela coordenação e monitorização da sua utilização, preocupando-se também com problemas como a segurança e o tempo de resposta. Um SGBD contem software de segurança e autorização usado pelo administrador para criar números de conta e definir restrições. Cada número de conta está normalmente associado a um conjunto de privilégios que condicionam o acesso do utilizador aos dados.

- *Desenhador de base de dados*

  Tem a responsabilidade de identificar os dados a serem armazenados na base de dados e de escolher quais as estruturas mais apropriadas para esses dados. É também responsável pelo diálogo com os futuros utilizadores de modo a estruturar os dados de uma maneira mais apropriada.

- *Utilizadores finais*

  Aqueles que acedem à base de dados para pesquisar, actualizar e gerar relatórios.

- *Analista de sistemas e programador de aplicações*

  Os analistas de sistemas determinam as necessidades dos utilizadores finais e criam as especificações necessárias para as transacções satisfazerem tais necessidades.

  Os programadores de aplicações implementam estas especificações escrevendo, testando, documentando e mantendo programas.

2.2 Modelos de dados

Os dados necessitam ser estruturados e organizados para que se possam tornar úteis. Os modelos de dados são usados para mostrar como é que as várias partes da base de dados estão relacionadas. Essencialmente, há três tipos de bases de dados logicamente estruturadas: hierárquico, rede e relacional.

2.2.1 Modelo hierárquico

Uma estrutura de dados hierárquica implica que uma entidade não possa ter mais do que uma entidade-pai. É, portanto, uma estrutura construída à custa de associações do tipo 1:M (um para muitos - um pai, vários filhos) ou 1:1 (um para um - um pai, um filho). Outras associações tais como M:1 ou M:N não são permitidas. Estas estruturas hierárquicas são por vezes denominadas árvores devido à semelhança existente entre as ligações entre as entidades e os ramos de uma árvore (Figura 2.4).

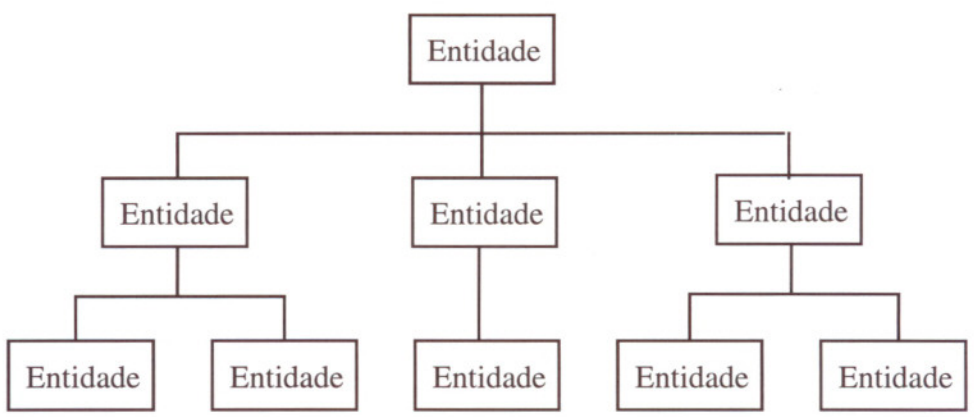


Figura 2.4 - Estrutura hierárquica de dados

A Figura 2.5 mostra um exemplo de base de dados para os empregados de uma firma. Neste exemplo, o conjunto dos empregados e respectivos departamentos está organizado como uma árvore. Uma base de dados hierárquica é uma colecção de tais árvores.

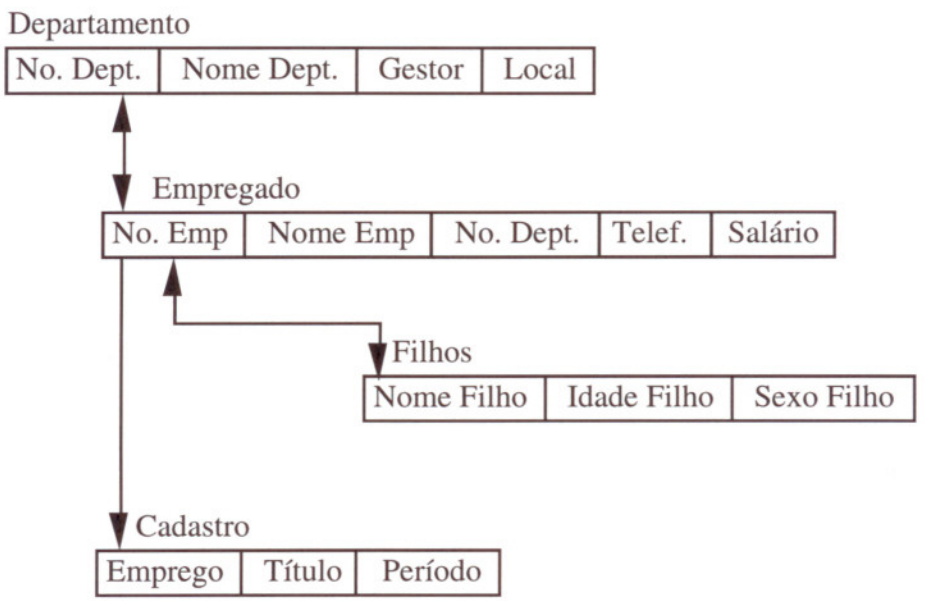


Figura 2.5 - Esquema para um ficheiro hierárquico

Nesta estrutura, os dados têm de estar estritamente encaixados, isto é, cada nodo tem apenas um pai. No entanto, no mundo real, há por vezes necessidade de um mesmo dado se repetir em vários locais da base de dados. Usando o modelo hierárquico, há para este caso, necessidade de se criarem registos virtuais. Em vez do dado, um registo virtual contém um ponteiro para o registo físico. A mesma técnica pode ser usada se necessitarmos de registos repetidos.

As bases de dados hierárquicas exibem pouca flexibilidade, embora tenham normalmente bons desempenhos em casos de aplicações pré-concebidas.

2.2 Modelo em rede

A estrutura em rede permite a qualquer entidade ter qualquer número de pais ou filhos (ver figura 2.6). As entidades são unidas através de ligações de rede que constituem itens de dados comuns a ambas as entidades ligadas.

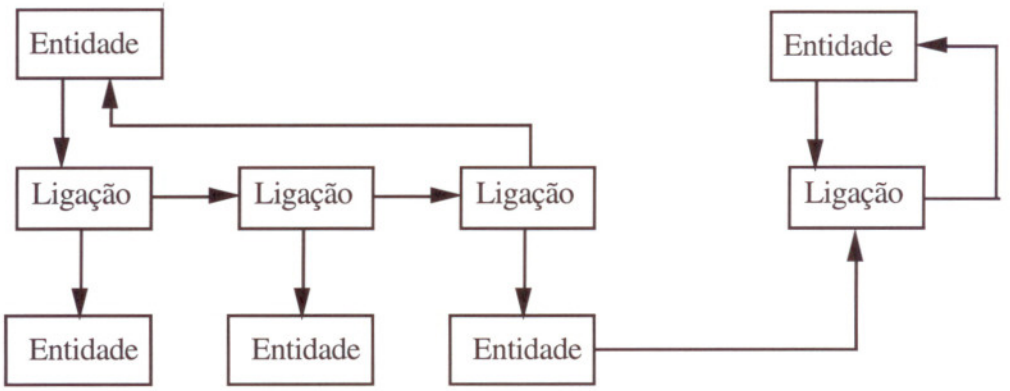


Figura 2.6 - Estrutura de dados em rede

Alguns dos problemas inerentes às estruturas hierárquicas podem ser ultrapassados pela estrutura em rede, embora esta seja mais complexa. As ligações, juntamente com os registos apontados, são referidos pela linguagem de definição dos dados, como conjuntos. A Figura 2.7 mostra o esquema de uma base de dados de empregados.

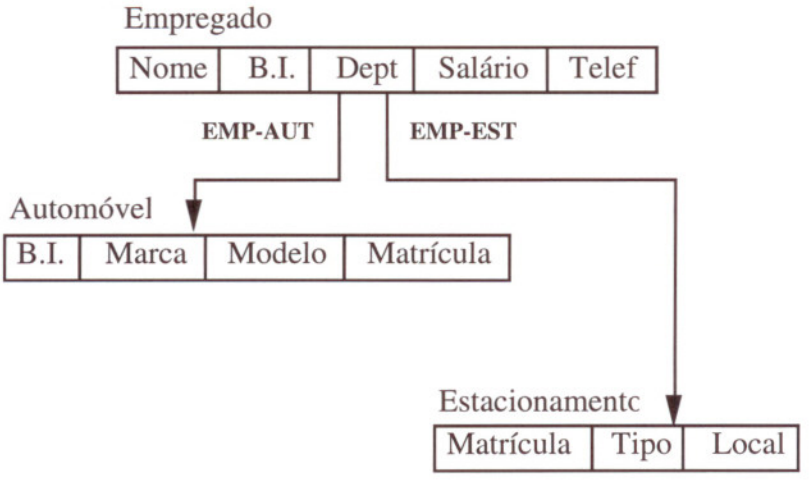


Figura 2.7 - Exemplo de uma base de dados em rede

2.2.3 Modelo relacional

Uma estrutura relacional consiste em uma ou mais tabelas chamadas relações. As linhas da tabela representam registos e as coluna, atributos. A figura 2.8 apresenta um pequeno exemplo de uma base de dados relacional.

Empregado

Nome	Idade	No. Dept
...	...	...
...	...	...
...	...	...
...	...	...
...	...	...
...	...	...
...	...	...
...	...	...

Departamento

No. Dept	Nome Dept
...	...
...	...
...	...
...	...
...	...

Figura 2.8 - Exemplo de tabelas de uma base de dados relacional

Manter as tabelas numa estrutura relacional é normalmente muito simples quando comparado com a manutenção de estruturas hierárquicas e ou em rede. Uma das grandes vantagens de uma estrutura relacional é que as pesquisas ad-hoc são manipuladas de uma maneira eficiente.

De uma maneira conceptual, as operações realizadas numa base de dados relacional limitam-se a cortes e colagens de porções de tabelas de modo a criar as tabelas.

2.3 Normalização

2.3.1 Introdução

A normalização permite-nos analisar o desenho de uma base de dados relacional, isto é, fornece-nos um método para identificar a existência de eventuais problemas. Este processo envolve vários tipos de formas normais: a primeira forma normal (1NF), a segunda forma normal (2NF) e a terceira forma normal (3NF). Estas formas normais constituem uma progressão na medida em que uma tabela que esteja na 1NF é melhor do que uma tabela que não esteja na 1NF; uma tabela na 2NF ainda é melhor; e assim por diante.

Iniciando o nosso trabalho com uma tabela ou com uma colecção de tabelas, o objectivo deste processo é produzir uma nova colecção de tabelas equivalente à original, mas isenta de problemas, ou seja, uma nova colecção de tabelas 3NF.

No entanto, antes de continuarmos, vamos definir dois conceitos importantes para a compreensão daquilo que se entende por normalização: dependência funcional e chave. E para isso vamos usar as relações representadas na Figura 2.9. Nesta figura podemos ver que existem três vendedores cujos números são 3, 6 e 12. O nome do vendedor 3 é Maria João que mora na Avenida 25 de Abril, 123, em Lisboa e o total das sua comissões é de 2150 contos à taxa de 5%.

VENDEDOR

NUMVEND	NOMEVEND	ENDVEND	TOTCOMM	PERCCOMM
3	MARIA JOÃO	AV 25 ABRIL, 123, LISBOA	2150.00	.05
6	MANUEL COSTA	R SACRAMENTO, 21, PORTO	4912.50	.07
12	FRANCISCO SOUSA	R CARLOS SEIXAS, 245, COIMBRA	2150.00	.05

CLIENTE

NUMCL	NOMECL	ENDEREÇO	SALDO	LIMCRED	NUMVEND
124	ILDA FERREIRA	AV BOAVISTA, 200, PORTO	418.75	500	3
256	ANTÓNIO PEREIRA	R JOSÉ ALMEIDA, 12, COIMBRA	10.75	800	6
311	BERTA MARTINS	EST S MARCOS, CACÉM	200.10	300	12
315	CARLOS DUARTE	R CARLOS SEIXAS, 17, COIMBRA	320.75	300	6
405	ERNESTO SANTOS	R VISC SETÚBAL, 140, PORTO	210.75	800	12
412	ILDA FERREIRA	AV 5 OUTUBRO, 102, LISBOA	908.75	1000	3
522	FERNANDO DIAS	R SEQUEIRA, 100, LISBOA	49.50	800	12
567	GRAÇA RIBEIRO	R CAM CAST BRANCO, COIMBRA	201.20	300	6
587	MARIA SILVA	AV COMB G GUERRA, 96, LEIRIA	57.75	500	6
622	CARLOS COSTA	AV CAL GULBENK, 98, COIMBRA	575.50	500	3

ENCOMENDAS

NUMENC	DATAENC	NUMCL
12489	20991	124
12491	20991	311
12494	40991	315
12495	40991	256
12498	50991	522
12500	50991	124
12504	50991	522

LINHAENC

NUMENC	NUMPEÇA	QUANT	PREÇO
12489	AX12	11	1 495
12491	BT04	1	40 299
12491	BZ66	1	31 195
12494	CB03	4	17 500
12495	CX11	2	5 795
12498	AZ52	2	2 295
12498	BA74	4	495
12500	BT04	1	40 299
12504	CZ81	2	10 899

PEÇA

NUMPEÇA	DESCPEÇA	STOCK	CLASSE	ARMAZÉM	PREÇOUNIT
AX12	FERRO	104	CASA	3	1 795
AZ52	SKATES	20	DESP	2	2 495
BA74	FUTEBOL	40	DESP	1	495
BH22	TORRADEIRA	95	CASA	3	3 495
BT04	FORNO	11	ELEC	2	40 299
BZ66	MAQLAVAR	52	ELEC	3	31 195
CA14	FRIGIDEIRA	2	CASA	3	1 995
CB03	BICICLETA	44	DESP	1	18 750
CX11	MISTURADORA	112	CASA	3	5 795
CZ81	PESOS	208	DESP	2	10 899

Figura 2.9 - Tabelas de uma base de dados relacional

Vemos também que há 10 clientes, numerados 124, 256, 311, 315, 405, 412, 522, 567, 587 e 622. O nome do cliente 124 é Ilda Ferreira que mora na Avenida da Boavista, 200, no Porto. O seu saldo actual é de 418.750$00 com um limite de crédito de 500 contos e é cliente do vendedor número 3 (Maria João).

Na tabela PEÇA vemos que há 10 peças numeradas AX12, AZ52, BA74, BH22, BT04, BZ66, CA14, CB03, CX11 e CZ81. A peça AX12 é um ferro e a empresa tem 104 unidades em stock. Estas pertencem à classe CASA , estão armazenadas no armazém 3 e custam 1.795$00.

Voltando à tabela ENCOMENDAS, vemos que há sete encomendas numeradas 12489, 12491, 12494, 12495, 12498, 12500 e 12504. A encomenda 12489 foi feita no dia 2 de Setembro de 1991 pelo cliente 124 (Ilda Ferreira).

A tabela LINHAENC contem uma descrição das encomendas. Por exemplo, a encomenda 12498 tem duas linhas de encomenda: a primeira para dois AZ52's a 2295$00 cada; a outra para quatro BA74's a 495$00 cada.

Um atributo B é funcionalmente dependente de outro atributo A se o valor de A determinar sempre um valor para B. De outro modo, se conhecermos o valor de A, será que conhecemos o valor de B? Em caso afirmativo, B é funcionalmente dependente de A (A-->B) e A funcionalmente determina B.

Por exemplo, na tabela CLIENTE, NOMECL é funcionalmente dependente de NUMCL. Sendo dado o número 124, apenas encontramos um nome. No entanto, ENDEREÇO não é funcionalmente dependente de NOMECL. Sabendo que o nome é Ilda Ferreira, não é possível determinar o seu endereço.

Na tabela LINHAENC, o atributo QUANT não é funcionalmente dependente de NUMENC pois não fornece informação suficiente. Também o não é de NUMPEÇA. No entanto, se considerarmos a combinação de NUMENC com NUMPEÇA, já podemos dizer que QUANT é funcionalmente dependente desta combinação.

O segundo conceito subjacente ao de normalização é o de chave primária. Um atributo A (ou um conjunto de atributos) é uma chave primária de uma relação (tabela) R se (i) todos os atributos de R forem funcionalmente dependentes de A; e se (ii) não existir outra subcolecção de atributos em A que satisfaça (1).

Na figura 2.9 podemos ver que NOMECL não pode ser uma chave primária da tabela CLIENTE, podê-lo no entanto ser o atributo NUMCL. Do mesmo modo NUMENC não pode ser chave primária da tabela LINHAENC, mas uma combinação formada pelos atributos NUMENC e NUMPEÇA já pode ser chave primária, pois todos os atributos podem ser determinados por esta combinação. Finalmente, e embora a combinação de NUMPEÇA com DESCPEÇA possuam esta última propriedade, esta combinação não é chave primária, pois o atributo NUMPEÇA sozinho também a possui.

2.3.2 Primeira forma normal

A primeira coisa que fazemos quando analisamos uma tabela relacional é tentarmos encontrar grupos repetidos. Uma relação (tabela) está na primeira forma normal (1NF) se não contiver grupos repetidos.

Consideremos a seguinte tabela

ENCOMENDAS (NUMENC, DATAENC, $\overline{\text{(NUMPEÇA,QUANT)}}$

ENCOMENDAS

NUMENC	DATAENC	NUMPEÇA	QUANT
12489	20991	AX12	11
12491	20991	BT04	1
		BZ66	1
12494	40991	CB03	4
12495	40991	CX11	2
12498	50991	AZ52	2
		BA74	4
12500	50991	BT04	1
12504	50991	CZ81	2

Figura 2.10 - Tabela não normalizada

Nesta tabela possuímos uma chave NUMENC, um atributo DATAENC e um grupo repetido contendo dois atributos NUMPEÇA e QUANT. Removendo o grupo repetido, obtemos

ENCOMENDAS (NUMENC, DATAENC, NUMPEÇA, QUANT)

ENCOMENDAS

NUMENC	DATAENC	NUMPEÇA	QUANT
12489	20991	AX12	11
12491	20991	BT04	1
12491	20991	BZ66	1
12494	40991	CB03	4
12495	40991	CX11	2
12498	50991	AZ52	2
12498	50991	BA74	4
12500	50991	BT04	1
12504	50991	CZ81	2

Figura 2.11 - Tabela normalizada 1NF

Notar que, por exemplo, a segunda linha da tabela da Figura 2.10 é substituída, na Figura 2.11, por duas linhas. A chave primária agora é uma combinação de NUMENC (chave original) com NUMPEÇA (chave do grupo repetido).

3.3 Segunda forma normal

A fase seguinte é preocuparmo-nos com dependências funcionais. A tabela que se segue

ENCOMENDAS (NUMENC, DATAENC, NUMPEÇA, DESCPEÇA, QUANT, PREÇO)

tem as dependências funcionais

NUMENC --> DATAENC

NUMPEÇA --> DESCPEÇA

NUMENC, NUMPEÇA --> QUANT, PREÇO

Portanto, NUMENC determina DATAENC, NUMPEÇA determina DESCPEÇA e a concatenação de NUMENC e NUMPEÇA determina QUANT e PREÇO. Consideremos o exemplo ilustrado pela Figura 2.12.

ENCOMENDAS

NUMENC	DATAENC	NUMPEÇA	DESCPEÇA	QUANT	PREÇO
12489	20991	AX12	FERRO	11	1 495
12491	20991	BT04	FORNO	1	40 299
12491	20991	BZ66	MAQLAVAR	1	31 195
12494	40991	CB03	BICICLETA	4	17 500
12495	40991	CX11	MISTURADORA	2	5 795
12498	50991	AZ52	SKATES	2	2 295
12498	50991	BA74	FUTEBOL	4	495
12500	50991	BT04	FORNO	1	40 299
12504	50991	CZ81	PESOS	2	10 899

Figura 2.12 - Tabela de encomendas

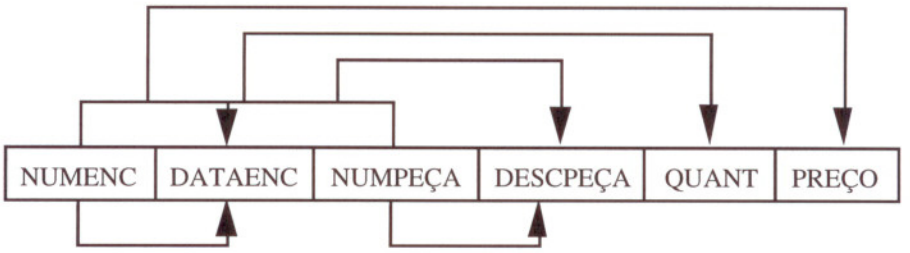
Como se pode ver nesta tabela, a descrição de uma determinada peça, BT04 por exemplo, ocorre várias vezes. Esta redundância provoca alguns problemas e ocupa espaço desnecessário. No entanto, outros problemas são bem mais perigosos. Entre estes há a destacar o problema da actualização, exigindo que todas as ocorrências sejam alteradas, de modo a evitar inconsistências entre os dados.

Problemas existem também quando pretendemos adicionar ou eliminar uma peça da tabela. Como a chave primária desta tabela é composta por NUMENC e NUMPEÇA, só podemos

adicionar uma peça nova quando tivermos uma encomenda para ela, mesmo sendo uma encomenda fictícia. Por outro lado, se eliminarmos, por exemplo, a encomenda 12489 da base de dados perdemos também a informação referente ao nome FERRO da peça AX12.

Estes problemas acontecem porque a nossa tabela tem um atributo, DESCPEÇA que não é dependente de toda a chave, sendo apenas do atributo NUMPEÇA. Por isso, chamamos atributo não chave a qualquer atributo que não faça parte da chave primária.

Uma relação (tabela) está na segunda forma normal (2NF) se estiver na 1NF e se não existir nenhum atributo não chave que seja dependente de parte da chave. Usando um diagrama de dependências, podemos representar as dependências funcionais existentes numa tabela. No nosso caso,



As setas de cima indicam as dependências que há a esperar; as de baixo são as que fazem com que a tabela não esteja na 2NF. Para resolver esta situação, procedemos do seguinte modo:

i Para cada subconjunto de atributos que compõem a chave primária, criamos uma tabela com esse subconjunto como chave:

(NUMENC
(NUMPEÇA
(NUMENC, NUMPEÇA

ii De seguida, colocamos cada um dos restantes atributos com a chave primária apropriada de que depende:

(NUMENC, DATAENC)
(NUMPEÇA, DESCPEÇA)
(NUMENC, NUMPEÇA, QUANT, PREÇO)

Isto leva-nos a construir as tabelas da Figura 2.13.

ENCOMENDAS

NUMENC	DATAENC
12489	20991
12491	20991
12494	40991
12495	40991
12498	50991
12500	50991
12504	50991

PEÇAS

NUMPEÇA	DESCPEÇA
AX12	FERRO
AZ52	SKATES
BA74	FUTEBOL
BH22	TORRADEIRA
BT04	FORNO
BZ66	MAQLAVAR
CA14	FRIGIDEIRA
CB03	BICICLETA
CX11	MISTURADORA
CZ81	PESOS

LINHAENC

NUMENC	NUMPEÇA	QUANT	PREÇO
12489	AX12	11	1 495
12491	BT04	1	40 299
12491	BZ66	1	31 195
12494	CB03	4	17 500
12495	CX11	2	5 795
12498	AZ52	2	2 295
12498	BA74	4	495
12500	BT04	1	40 299
12504	CZ81	2	10 899

Figura 2.13 - Conversão para 2NF

Notar que os problemas existentes anteriormente já foram eliminados, não havendo redundâncias pois a descrição apenas aparece uma vez. Para adicionar uma peça nova e a respectiva descrição, basta acrescentar uma linha à tabela PEÇA. Do mesmo modo, apagar a encomenda 12489 não implica a eliminação da peça AX12 da tabela PEÇA.

2.3.4 Terceira forma normal

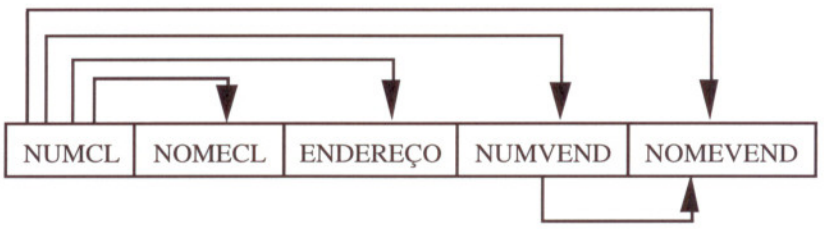
Uma tabela na 2NF ainda pode ter problemas. Vejamos a tabela CLIENTE

CLIENTE (NUMCL, NOMECL, ENDEREÇO, NUMVEND, NOMEVEND)

com as dependências funcionais

NUMCL --> NOMECL, ENDEREÇO, NUMVEND, NOMEVEND

NUMVEND --> NOMEVEND



CLIENTE

NUMCL	NOMECL	ENDEREÇO	NUMVEND	NOMEVEND
124	ILDA FERREIRA	AV BOAVISTA, 200, PORTO	3	MARIA JOÃO
256	ANTÓNIO PEREIRA	R JOSÉ ALMEIDA, 12, COIMBRA	6	MANUEL COSTA
311	BERTA MARTINS	EST S MARCOS, CACÉM	12	FRANCISCO SOUSA
315	CARLOS DUARTE	R CARLOS SEIXAS, 17, COIMBRA	6	MANUEL COSTA
405	ERNESTO SANTOS	R VISC SETÚBAL, 140, PORTO	12	FRANCISCO SOUSA
412	ILDA FERREIRA	AV 5 OUTUBRO, 102, LISBOA	3	MARIA JOÃO
522	FERNANDO DIAS	R SEQUEIRA, 100, LISBOA	12	FRANCISCO SOUSA
567	GRAÇA RIBEIRO	R CAM CAST BRANCO, COIMBRA	6	MANUEL COSTA
587	MARIA SILVA	AV COMB G GUERRA, 96, LEIRIA	6	MANUEL COSTA
622	CARLOS COSTA	AV CAL GULBENK, 98, COIMBRA	3	MARIA JOÃO

Figura 2.14 - Tabela CLIENTE

Esta tabela está na 2NF pois a chave primária é composta apenas por uma coluna. No entanto, persistem alguns problemas. É o caso do nome do vendedor 12 (Francisco Sousa) que aparece várias vezes. Esta redundância cria o mesmo conjunto de problemas já descrito anteriormente. Por exemplo, a alteração do nome de um vendedor exige a alteração de várias linhas da tabela. Se quisermos acrescentar o vendedor número 47, cujo nome é Maria Daniela por exemplo, teremos de criar pelo menos um cliente seu. Tudo isto acontece porque o atributo NUMVEND não é uma chave primária, aparecendo várias vezes.

Antes de definirmos 3NF, consideremos como atributo (ou colecção de atributos) determinante qualquer atributo que determina outro atributo. NUMVEND é determinante pois determina NOMEVEND. Chamemos também chave candidata qualquer atributo (ou colecção de atributos) que podia ter funcionado como chave primária. Assim, uma relação (tabela) está na terceira forma normal (3NF) se estiver na 2NF e se os únicos atributos determinantes que possuir foram chaves candidatas.

Para colocar esta tabela na 3NF, procedemos do seguinte modo:

i Para cada determinante que não seja chave candidata, removemos da tabela os atributos que dele dependem.

ii Criamos uma nova tabela contendo todos os atributos da tabela original que dependem deste determinante.

iii Colocar o determinante como chave principal desta tabela.

A tabela 2NF é substituída por

CLIENTE (<u>NUMCL</u>, NOMECL, ENDEREÇO, NUMVEND)

e VENDEDOR (NUMVEND, NOMEVEND)

Os nomes dos vendedores aparecem apenas uma vez, evitando assim a redundância e facilitando a alteração do nome do vendedor. Para criar um novo vendedor, basta acrescentar uma linha à tabela VENDEDOR.

CLIENTE

NUMCL	NOMECL	ENDEREÇO	NUMVEND
124	ILDA FERREIRA	AV BOAVISTA, 200, PORTO	3
256	ANTÓNIO PEREIRA	R JOSÉ ALMEIDA, 12, COIMBRA	6
311	BERTA MARTINS	EST S MARCOS, CACÉM	12
315	CARLOS DUARTE	R CARLOS SEIXAS, 17, COIMBRA	6
405	ERNESTO SANTOS	R VISC SETÚBAL, 140, PORTO	12
412	ILDA FERREIRA	AV 5 OUTUBRO, 102, LISBOA	3
522	FERNANDO DIAS	R SEQUEIRA, 100, LISBOA	12
567	GRAÇA RIBEIRO	R CAM CAST BRANCO, COIMBRA	6
587	MARIA SILVA	AV COMB G GUERRA, 96, LEIRIA	6
622	CARLOS COSTA	AV CAL GULBENK, 98, COIMBRA	3

VENDEDOR

NUMVEND	NOMEVEND
3	MARIA JOÃO
6	MANUEL COSTA
12	FRANCISCO SOUSA

Figura 2.15 - Conversão para 3NF

3 Utilização de dBase IV

3.1 Conceitos básicos

3.1.1 Introdução

dBase IV é um produto da Ashon-Tate lançado em 1990 e pode ser usado em dois modos: no chamado modo convencional dBase e no modo SQL, um standard da maioria das bases de dados relacionais actuais. Possui também uma linguagem de alto nível, permitindo aos utilizadores o desenvolvimento de programas de aplicação.

O acesso ao dBase é feito através do Centro de Controlo onde podemos controlar os objectos dBase: podemos manipular ficheiros (criar, modificar, abrir, acrescentar, editar, percorrer, fechar), criar perspectivas, definir formulários, criar ou gerar relatórios, correr programas e até usar o gerador de aplicações para criar novos programas.

3.1.2 Ficheiros, registos e bases de dados

O sistema de gestão de bases de dados utiliza ficheiros para armazenar informação. Estes ficheiros podem ser de dois tipos: ficheiros de programas e ficheiros de dados.

Os ficheiros de programas são um conjunto de instruções a serem processadas sobre os dados. Podem ser criados por qualquer editor, pelo editor interno do dBase IV, pelo gerador de aplicações ou por qualquer processador de texto em modo não-documento ou ASCII. Possuem extensões .PRG, .FRG ou .LGB. Antes de executar qualquer programa, o dBase compila-o e cria um ficheiro com a extensão .xxO.

Um ficheiro de dados é uma colecção de dados relacionados, ou registos, estando cada um destes registos dividido em campos. As extensões usadas para ficheiros de dados são .DBF para os ficheiros de base de dados e .NDX ou .MDX para ficheiros índice.

O dBase IV pode manter até 99 ficheiros abertos num determinado instante, podendo 10 deles serem bases de dados. Os ficheiros são abertos através do comando USE e são atribuídos a uma área de trabalho identificada por um número de 1 a 10. O comando SELECT permite mover de uma área para outra.

```
SELECT 1
USE conta                    USE conta IN 1
SELECT 2
USE lote                     USE lote IN 2
```

O processo mais simples para fechar todos os ficheiros é usar o comando CLEAR ALL, sendo libertadas as bases de dados, os ficheiros e as variáveis de memória utilizadas. Para se fechar todas as bases de dados sem alterar nem os ficheiros nem as variáveis, utiliza-se o comando CLOSE DATABASES. Abrir um ficheiro numa área de trabalho leva ao fecho do ficheiro que a ocupava anteriormente.

3.1.3 Tipos de dados, operadores e variáveis de memória

Em linguagem de programação, uma variável é uma etiqueta que correspondente a uma determinada posição de memória. Em dBase IV as variáveis podem ser de qualquer dos tipos representados na Figura 3.1.

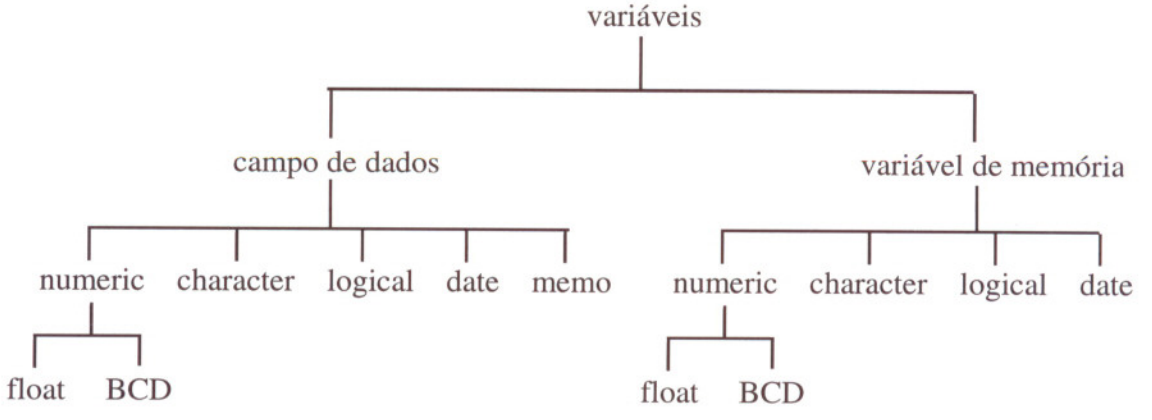


Figura 3.1 - Tipos de variáveis do dBase IV

As variáveis campo de dados fazem parte dos registos dos ficheiros e os seus valores podem ser armazenados através dos comandos REPLACE ou GET:

```
REPLACE preço WITH mpreço
```

As variáveis de memória residem na memória do computador e podem ser alteradas através do comando STORE ou do sinal de igual:

```
STORE 8 TO matr
STORE 0.00 TO mqte, mvalor, mcvho
STORE "Gráfico No. 1" TO título
RESP = .N.
```

Estas variáveis podem ser guardadas no disco ou restauradas de disco. O comando

```
SAVE TO temp
```

guarda todas as variáveis de memória no ficheiro TEMP.MEM. Se usarmos o ponto-de-interrogação para representar um carácter e o asterisco para representar um grupo de caracteres, podemos usar este comando de uma maneira mais eficiente:

```
SAVE TO temp ALL LIKE m*
SAVE TO temp ALL EXCEPT m???
```

O comando RESTORE pode ser usado para inicializar variáveis. No entanto, sempre que é usado, apaga as variáveis de memória existentes. Para se evitar isso, pode usar-se a componente ADDITIVE.

```
RESTORE FROM temp
RESTORE FROM temp ADDITIVE
```

Os dados numéricos podem ser do tipos fixo (BCD) ou flutuante (binário). O primeiro possui uma representação exacta, enquanto que o segundo tem uma representação aproximada. As variáveis de memória podem ser N(BCD) ou F(binário).

As variáveis de tipo character podem conter um ou mais caracteres alfanuméricos. As variáveis de tipo boolean podem ter os valores T, t, Y ou y para *true* ; e F, f, N, n para *false*. As variáveis de tipo date são armazenadas como números, embora possam ser introduzidas no computador ou escritas em formato de data. Para isso podemos usar as funções que convertem datas no formato desejável.

Como o dBase não possui declarações de variáveis, se escrevermos, por exemplo,

```
@ 5,1 SAY "Introduza o nome" GET mnome
READ
```

o programa não sabe qual o tipo de dados que deve esperar. O programa a seguir resolverá este problema:

```
STORE SPACE(20) TO mnome
STORE 0.00 TO mquant
@ 5,1 SAY "Introduza o nome" GET mnome
@ 6,1 SAY "Introduza o valor em dívida" GET mquant
READ
```

O dBase IV pode utilizar variáveis do tipo *array* com uma ou duas dimensões, tendo a característica de não exigir que todas as variáveis de memória sejam do mesmo tipo. Variáveis de memória do tipo *array* são úteis, por exemplo, sempre que se pretenda armazenar uma matriz de variáveis a serem escritas no monitor ou impressas num relatório. Para criar espaço em memória, necessitamos de usar o comando DECLARE. Só então podemos copiar o registo corrente para o *array* . O comando DECLARE serve também para limpar um *array* .

```
DECLARE item[15]
COPY TO ARRAY item
```

O comando DISPLAY MEMORY envia para o monitor os elementos do *array* . Estes elementos serão exactamente do mesmo tipo dos campos correspondentes do registo.

Notar que o comando COPY apenas copia o primeiro registo do ficheiro. Para se copiar um registo determinado, necessitamos de posicionar o ficheiro nesse registo e só então usar o comando

```
COPY NEXT 1 TO ARRAY item
```

Usando o comando SET FIELDS, podemos especificar um subconjunto de campos a serem copiados.

O programa que se segue é uma rotina que apresenta o nome e o conteúdo do primeiro registo de uma base de dados com quatro campos, permitindo que estes sejam editados. A variável de tipo *array* é usada como armazenagem temporária.

```
SET TALK OFF
USE temp
DECLARE buf[4]
COPY NEXT 1 TO ARRAY buf
nocampo = 1
DO WHILE nocampo <= 4
     @ 2,1 SAY FIELD(nocampo)
     @ 2,15 GET buf[nocampo]
     READ
     @ 2,0
     nocampo = nocampo + 1
ENDDO
nocampo = 1
DO WHILE nocampo <= 4
     nome = FIELD(nocampo)
     REPLACE &nome WITH buf[nocampo]
     nocampo = nocampo + 1
ENDDO
```

Para duas dimensões, os dados são representados numa matriz com linhas e colunas declaradas do seguinte modo:

```
DECLARE endereços[15,6]
```

3.1.4 Expressões, funções e programas

A menos que se use o programa SET PRINT ON, o resultado do comando ? é enviado para o monitor para a linha imediatamente a seguir. O comando ?? envia os resultados sem previamente saltar uma linha.

```
USE meufich
GO BOTTOM
? 'Há' + STR(RECNO(),3)
?? 'registos no ficheiro'
```

RECNO() é uma função que devolve o número de registo corrente. STR() converte um numérico em cadeia de caracteres.

Há certas convenções sempre que usamos variáveis do tipo date:

- Para se obter o ano com 4 dígitos temos de usar o comando SET CENTURY ON
- O domingo é considerado o primeiro dia da semana.
- Para se alterar o formato da data, usar o comando SET DATE
- Para contar o número de registos para um dado período de tempo, usamos a função LIKE()

```
COUNT FOR LIKE("06/??/91"DTOC(data)) TO junho
```

- Para indexar ou concatenar com outras cadeias de caracteres, temos de converter as datas para cadeias de caracteres.

```
? "Hoje é " + DTOC(DATE())
```

° Para se usar datas em funções aritméticas, estas têm de estar no formato data

```
STORE MIN(data1, data2) TO datamin
```

Em dBase IV podemos também criar funções definidas por nós. São as chamadas UDF (user-defined functions). O programa que se segue determina o valor da diagonal de um rectângulo.

```
comp = 0.00
larg = 0.0
@ 2.1 SAY "Comprimento: " GET comp
@ 3,1 SAY "Largura: " GET larg
READ
@ 5,1 SAY "A diagonal é " + STR(diag(comp,larg),5,2)
RETURN
FUNCTION diag
PARAMETERS x,y
RETURN sqrt(x**2+y**2)
```

As UDFs podem ser usadas, por exemplo, para controlar os valores introduzidos num programa, ou para encriptar dados. No entanto, não podemos usar funções para realizar acções; as funções têm de devolver um valor. São normalmente definidas como parte de um ficheiro .PRG e são compiladas para obter um ficheiro .DBO.

Algumas operações matemáticas são imediatamente postas à disposição. São elas os comandos COUNT, SUM, TOTAL, AVERAGE e CALCULATE.

O comando COUNT conta o número de registos na base de dados que satisfazem determinadas condições.

```
USE endereços
COUNT FOR cidade = 'Coimbra' TO x
? x
```

O comando SUM permite-nos somar variáveis de campo de tipo numérico. Se tivermos uma base de dados com os campos 'quantidade encomendada' e 'peço unitário', as instruções que se seguem permitir-nos-ão calcular o valor total de encomendas.

```
SUM quant * prunit TO x
? x
SUM quant * prunit TO x FOR cidade = 'Coimbra'
? x
```

O comando AVERAGE obtém a média dos valores de um campo, incluíndo os zeros.

```
AVERAGE notas  TO média
```

O comando TOTAL permite-nos compactar informação de um ficheiro. Por exemplo se tivermos o ficheiro vendedores

Registo No.	Cidade	NoVend
1	Lisboa	7
2	Porto	2
3	Porto	5
4	Coimbra	1
5	Coimbra	3

e correrms o programa

```
USE vendedores
INDEX ON cidade TO vendedores
TOTAL ON cidade TO resumo
```

o ficheiro resultante será

```
4          Coimbra          4
1          Lisboa           7
2          Porto            7
```

Os comandos COUNT, SUM, AVERAGE e TOTAL são raras vezes usados em programas por serem demasiado lentos. Com o comando CALCULATE podemos usar várias funções estatísticas

```
CALCULATE SUM(quant*custo),AVG(quant),STD(quant) TO custotal,média,desvpd
```

Falemos agora de programas. Um programa é uma sequência de instruções escritas e armazenadas num ficheiro a ser executado uma ou mais vezes. Uma aplicação é um conjunto de programas, bases de dados e outros ficheiros concebidos para realizar uma determinada tarefa.

Um programa pode ser executado através da zona *Applications* do *Control Center* ou através do comando DO. Os programas são então compilados e executados. Notar que não se trata de uma verdadeira compilação pois não é criado nenhum módulo executável; é apenas criado um ficheiro .DBO e, para o executar, necessitamos sempre de ter o dBase IV ou o módulo *Run Time* .

Para se criar ou editar um programa podemos usar o *Control Center* , um qualquer processador de textos em formato de não documento ou o comando

```
. MODIFY COMMAND xxx
```

Usando o Centro de Controlo, implicitamente actualizamos o catálogo de ficheiro e apagamos os ficheiros .DBO obsoletos. O comando MODIFY apenas apaga os ficheiros .DBO. A título de exercício, crie o seguinte programa TESTE. PRG

```
STORE 6 TO mquant1
STORE 1 TO mquant2
STORE mquant1 + mquant2 TO mquant
? mquant
```

De seguida escreva as instruções

```
DO teste
SET TALK OFF
DO teste
```

e veja o efeito de se usar o comando SET TALK.

Com o dBase IV, podemos também usar procedimentos. Ao contrário das funções, os procedimentos admitem valores de entrada e saída.

```
...
DO teste WITH mcampo, resposta
? resposta
...
RETURN

PROCEDURE teste
PARAMETERS nomecampo, resultado
IF &nomecampo > 0
   STORE .t. TO resultado
ELSE
   STORE .f. TO resultado
ENDIF
RETURN
```

Reparar no seguinte:

- existe sempre um RETURN no fim de cada procedimento e do programa
- utilização da macro &nomecampo
- a instrução PARAMETERS é colocada imediatamente a seguir à primeira instrução do procedimento
- os identificadores das variáveis no procedimento não necessitam ser os mesmos da chamada.

Normalmente, um programa é composto por quatro partes: o preâmbulo, a inicialização, o corpo do programa e a secção final. No preâmbulo, incluem-se normalmente, pelo menos, o seguinte:

- uma linha a descrever o objectivo do programa
- nome do autor
- nome do programa
- data da última revisão
- nomes dos ficheiros e índices abertos durante o programa
- variáveis de memória necessárias quando da chamada

Na área de inicialização são colocados todos os comandos SET, abertos os ficheiros e inicializadas as variáveis.

O corpo do programa contem as instruções que executam tarefas no programa.

A parte final do programa é usada para fechar ficheiros, fazer saltar uma última página da impressora e actualizar variáveis globais.

A detecção de erros é uma área extremamente importante em qualquer programa. Para isso há algumas regras a respeitar.

- As entradas de dados pelo teclado devem ser sempre validadas.
- Sempre que se usam os comandos FIND ou SEEK para pesquisar um registo, o programa deve controlar se o referido registo existe ou não. Para isso, podemos usar as funções EOF(), BOF() ou FOUND().
- Sempre que se usa o comando DO CASE deve existir a opção OTHERWISE para o caso de erro.

Geralmente, usa-se a última linha de baixo do monitor para emitir as mensagens de erro. Por vezes há necessidade, no caso de ocorrer erro, de dar ao utilizador a possibilidade de continuar ou de parar.

```
STORE '   ' TO continua
WAIT 'Carregue numa tecla ...' TO continua
```

O exemplo que se segue limpa a linha de mensagem, escreve a mensagem de erro, toca o beep e esperar um certo tempo para prosseguir.

```
STORE 1 to xx
@ 23,1
@ 23,1 SAY 'mensagem de erro'
?? CHR(7)
DO WHILE xx < 35
   STORE xx+1 TO xx
ENDDO
```

O dBase possui também a opção ERROR para o comando @ ... GET, permitindo assim a escrita de uma mensagem em caso de erro.

```
@ 5,1 SAY 'Para apagar?' GET resposta ;
    PICTURE '!' ;
    VALID resposta$ "SN" ERROR "Responda S ou N"
READ
```

3.1.5 Estruturas de controlo

As instruções de controlo permitem-nos alterar dinâmicamente a sequência segunda o qual as instruções são executadas. Exemplos de estruturas de controlo são as seguintes:

1 via

```
IF <expressão de controlo>
    <instruções>
ENDIF
```

```
STORE .F. TO excedido
IF (crédito > limite)
   ? 'O limite de crédito foi excedido'
   STORE .T. TO excedido
ENDIF
```

2 vias

```
IF <expressão de controlo>
    <instruções-1>
ELSE
    <instruções-2>
ENDIF
```

```
IF (crédito > limite)
   ? 'O limite de crédito foi excedido'
   STORE .T. TO excedido
ELSE
   STORE .F. TO excedido
ENDIF
```

```
excedido = IIF (Crédito > limite, .T., .F.)
```

várias vias mutuamente exclusivas

```
IF <expressão de controlo-1>
    <instruções-1>
ELSE IF <expressão de controlo-2>
        <instruções-2>
          ELSE IF <expressão de controlo-3>
              <instruções-3>
        ENDIF
     ENDIF
ENDIF
```

várias vias mutuamente inclusivas

```
IF <expressão de controlo-1>
    <instruções-1>
    IF <expressão de controlo-2>
        <instruções-2>
        IF <expressão de controlo-3>
            <instruções-3>
        ENDIF
     ENDIF
ENDIF
```

Duas outras hipóteses de se implementar uma estrutura mutuamente exclusiva é usando o comando DO ... CASE ... ENDCASE

```
WAIT "Introduza a opção" TO opção
DO CASE
          CASE opção = '1'
              DO opção1
          CASE opção = '2'
              DO opção2
          CASE opção = '3'
              DO opção3
         OTHERWISE
              @ 23,1
          @ 23,1 SAY 'Opção inválida'
          ENDCASE
```

ou um *array*

```
SET TALK OFF
DECLARE opção[3]
STORE "opção1"  TO opção[1]
STORE "opção2"  TO opção[2]
STORE "opção3"  TO opção[3]
WAIT "Introduza a opção " TO x
DO &opção[VAL(x)]
RETURN

PROCEDURE opção1
? 'Esta é a opção 1'
DO espera
RETURN

PROCEDURE opção2
? 'Esta é a opção 2'
DO espera
RETURN

PROCEDURE opção3
? 'Esta é a opção 3'
DO espera
RETURN

PROCEDURE espera
xx = 1
DO WHILE xx < 70
     STORE xx+1 TO xx
ENDDO
```

3.1.6 Estruturas de iteração

As estruturas de iteração permitem ao programa executar um grupo de instruções um número determinado de vezes, desde que a expressão de controlo seja verdadeira.

```
DO WHILE <expressão de controlo>
     <instruções>
ENDDO
```

```
xx = 1
DO WHILE xx < 70
     STORE xx+1 TO xx
ENDDO
```

Dentro de um ciclo DO WHILE pode usar-se o comando LOOP. Este comando faz saltar o controlo do programa para o início do ciclo; este ciclo será repetido até a expressão de controlo ser falsa. Se, por outro lado, durante a execução de um ciclo, pretendermos sair e ir directamente para a instrução que imediatamente se segue ao ciclo, usamos o comando EXIT.

O programa que se segue lê um ficheiro sequencial, salta os registos apagados e, para os restantes, escreve o conteúdo do campo nome.

```
GO TOP
DO WHILE .NOT. EOF()
     IF DELETED()
          SKIP
          LOOP
     ENDIF
     ? nome
     SKIP
ENDDO
```

Em alternativa, podemos usar as instruções

```
SCAN
     IF DELETED()
          LOOP
     ENDIF
     ? nome
ENDSCAN
```

3.1.7 Procedimentos de acontecimentos

dBase IV possui vários comandos para processar acontecimentos. Entre eles há a destacar os seguintes:

ON KEY <comando>	Executa o comando sempre que uma tecla é pressionada.
ON ESCAPE <comando>	Executa o comando sempre que a tecla Esc é pressionada.
ON ERROR <comando>	Executa o comando sempre que há uma condição de erro.
ON READERROR <comando>	Útil para erros de leitura, tais como datas não válidas e dados fora dos limites.
ON PAGE <comando>	Executa o comando sempre que há um salto de página. Útil para cabeçalhos.
ON PAGE AT LINE <no> <comando>	Útil para notas de rodapé.

Por vezes, em programas mais complexos, há necessidade de existir um programa de ajuda. E este pode estar disponível sempre que o utilizador carregue na tecla F1, por exemplo

```
ON KEY LABEL F1 DO ajuda01
```

3.2 Desenho de uma base de dados

3.2.1 Criação de uma base de dados

Normalmente, a primeira coisa a fazer quando se desenha uma base de dados é definir qual o tipo de informação que se pretende extrair da base de dados: qual a informação contida nos relatórios, frequência de emissão dos relatórios, aspecto dos relatórios e para quem são eles destinados.

Começa-se então por desenhar os écrans necessários para entrada e saída dos dados. Finalmente é decidido quais os ficheiros a usar. Nas linhas que se seguem vamos usar uma base de dados IDENT com alguns dados de identificação.

1	Número de estudante	NUMERO	N	11,0
2	Nome	NOME	C	60
3	Endereço	ENDEREÇO	C	60
4	Código postal	CODPOST	N	4,0
5	Localidade	LOCAL	C	20
6	Número de telefone	TELEF	C	10

Antes de efectivamente se criar uma base de dados dBase para a nossa aplicação, temos de definir um ficheiro catálogo. Trata-se de um ficheiro usado para controlar todo as bases de dados de uma determinada aplicação. É criado carregando-se F10 (acesso ao Centro de Controlo), seleccionando-se *Use of Different Catalog* do menu *Catalog* e colocando-se o nome do catálogo. No nosso exemplo, vamos chamá-lo de CV. Assim, sempre que criarmos uma base de dados, menus, formulários ou programas estes ficheiros estarão neste catálogo.

Para se criar a nova base de dados IDENT, basta seleccionar a opção *Create* do painel *Data* e introduzir-se os campos com a respectiva descrição. Caso se pretenda um campo de índice, este pode agora ser seleccionado. Logo que a tabela tenha sido definida, usamos Ctrl-End para a gravar em disco. De seguida, o dBase pergunta-nos se pretendemos introduzir dados. Respondamos por agora que não.

A qualquer altura podemos ver a estrutura de uma base de dados, seleccionando a base de dados do painel *Data* e escolhendo a opção *Modify structure/order*. Depois de alterada a estrutura podemos usar Ctrl-End ou Ctrl-W para gravar as alterações. Com Esc ou Ctrl-Q saímos sem gravar as alterações.

Como já vimos, podemos abrir várias bases de dados ocupando cada uma delas uma determinada área de memória

```
USE fichenc IN 1
USE fichcl IN 2
```

Ao fazer-se SELECT 1, por exemplo, estamos imediatamente a aceder aos campos da base de dados de encomendas. No entanto, podemos também aceder a campos de outras bases de dados sem que para isso tenhamos de usar o comando SELECT.

```
SELECT 1
STORE fichcl -> numcl TO mnumcl
```

Para melhorar a leitura do programa podemos também usar pseudónimos (aliases), sendo assim possível aceder a áreas de memória por nome e não por número (o nome, por defeito, de cada área de memória é o identificador do respectivo ficheiro).

```
SELECT 1
USE fichenc ALIAS encomendas
SELECT 2
USE fichcl ALIAS clientes
SELECT encomendas
...
SELECT fichcl
...
```

A forma geral do comando para abrir uma base de dados sem qualquer índice é

USE <base de dados> [IN <área de trabalho> [ALIAS <pseudónimo>]]

3.2.2 Ordenação de registos

Sempre que um registo é adicionado a uma base de dados, este é fisicamente colocado no fim do ficheiro. A indexação é um bom meio para termos os registos sempre ordenados sem termos explicitamente de os ordenar.

O índice é um ficheiro isolado podendo ser dois tipos: os índices de tipo .NDX e os de tipo .MDX. O primeiro, completamente compatível com o dBase III, é útil como índice secundário ou temporário. O segundo, pode conter múltiplas definições de índice.

Podemos criar um ou mais índices . NDX para cada campo, combinação de campos ou combinação entre partes de campos, sendo cada índice armazenado num ficheiro separado. Estes índices são criados através de instruções do tipo

```
USE conta
INDEX ON numconta TO conta
```

ou

```
INDEX ON numconta+dept TO conta
```

Logo que criado, o índice pode ser acedido com a instrução

```
USE conta INDEX conta
```

Desde que o ficheiro contenha dados, estes serão acedidos segundo a ordem determinada pelo índice. A actualização do índice é automática sempre que os ficheiros é alterado.

O outro tipo de índices aceites pelo dBase IV é o índice .MDX. Este tipo de índice é chamado índice de produção e pode conter múltiplos índices, cada um deles praticamente semelhante a um ficheiro separado de índice. Quando o ficheiro de dados é aberto, podemos assim abrir de imediato vários índices.

Os ficheiros de tipo .MDX serão imediatamente criados sempre que, na criação da base de dados, se especificar que se pretende criar um índice. Podemos também criar várias ordens dentro de um ficheiro de índice. Dentro de um programa, cada nova ordem (tag) pode ser criada através do comando

```
INDEX ON <chave> TAG <ordem> OF <ficheiro.mdx> [UNIQUE]  [DESCENDING]
```

Para se poder ver os registos de uma base de dados segundo uma determinada ordem já criada, podemos usar o comando semelhante ao que se segue

```
USE conta ORDER numcl
```

Caso o ficheiro de dados já esteja aberto e pretendermos alterar a ordem, podemos usar o comando

```
SET ORDER TO <ordem> OF <ficheiro.mdx>
```

Para se apagar um índice de ordem usa-se o comando

```
DELETE TAG <ordem> OF <ficheiro.mdx>
```

Há também algumas regras que devemos respeitar sempre que usamos índices. Passaremos de seguida a enunciar algumas delas:

- Para se evitar degradação do tempo de processamento, manter apenas o índice primário (uma ordem dentro de um ficheiro .MDX ou um ficheiro .NDX) sempre activo e criar índices temporários sempre que necessário.

- Garantir que o índice primário esteja aberto sempre que o campo chave é alterado.

- Para se poder manter um melhor controlo sobre o ficheiro de índice, sempre que possível, não permitir ao utilizador alterar a chave primária de índice.

- Usar os ficheiros .MDX como chaves primárias e usar os ficheiros .NDX para relatórios e consultas.

- Nas aplicações, manter sempre uma rotinas para a necessidade de reindexar os índices.

- Nunca usar índices temporários num ficheiro .MDX.

- Para reverter a ordem de um índice, usar a opção DESCENDENT no comando de índice.

O processo usado para pesquisar elementos numa base de dados dBase depende do tipo de ficheiros usados: indexados ou não. Para ficheiros indexados, é mais rápido utilizarem-se os comandos SEEK e FIND. O comando FIND apenas pesquisa cadeias de caracteres, tendo nós previamente de converter para cadeias de caracteres os valores numéricos ou datas. Se usarmos o conteúdo de uma variável de memória, esta tem de aparecer especificada com o carácter &

```
FIND &x
```

Se não fôr encontrado nenhum registo, a função EOF() é posta a *true* . Caso contrário, a função RECNO() devolve o número do registo.

O comando SEEK é semelhante ao comando FIND com a excepção de que pode ser usado com expresões e que não necessita do carácter & quando usado com variáveis de memória. Os campos numéricos e datas podem ser usados directamente sem necessidade de serem convertidos para cadeias de caracteres. Se o registo não é encontrado, EOF() é verdadeira.

```
USE conta INDEX numconta
STORE '1201' TO x
SEEK x
IF .NOT. EOF()
     - registo encontrado -
ELSE
     - registo não encontrado -
ENDIF
```

Ambos os comandos FIND e SEEK actualizam o valor da função FOUND():

```
USE conta INDEX numconta
STORE '1201' TO x
SEEK x
IF FOUND()
     - registo encontrado -
ELSE
     - registo não encontrado -
ENDIF
```

Podemos usar o comando SEEK para evitar duplicações num ficheiro:

```
USE clientes INDEX numcl
STORE SPACE(4) TO mnumcl
@ 3,1 SAY "Introduza o número de cliente:"
@ 4,1 GET mnumcl VALID SEEK(mnumcl);
     ERROR mnumcl + 'não está no ficheiro'
READ
```

Para o acesso a ficheiros indexados podemos usar três interruptores lógicos (set switches): SET UNIQUE, SET EXACT e SET NEAR.

Com SET UNIQUE, os comandos LIST ou DISPLAY apenas listam o primeiro item de cada elemento da chave. Podemos assim listar, por exemplo,todas as cidades diferentes existente num ficheiro de endereços indexado por cidade.

O SET EXACT permita determinar como é que as cadeias de caracteres são comparadas. Colocado em OFF, a cadeia à esquerda do sinal de igual é comparada carácter a carácter só até ao comprimento da cadeia da direita. Assim, 'ABC'='ABCDEF' é falsa e 'ABCDEF'='ABC' é verdadeira. Se SET EXACT estiver colocado em ON, ambas as expressões anteriores são falsas.

O SET NEAR permite-nos encontrar qual o registo que mais se aproxima da pesquisa, caso os comandos SEEK ou FIND falhem (FOUND()=.F.).

Para pesquisas complexas, o processo sequencial, embora mais lento, pode ser o adequado. A instrução seguinte encontrará o primeiro registo que satisfaz as condições:

```
LOCATE FOR APELIDO="Silva" .AND. CODPOSTAL="3000"
```

Podemos também usar-se macros

```
STORE '          ' TO mpesquisa
@ 15,1 SAY 'Introduza a condição de pesquisa' GET mpesquisa
LOCATE FOR &mpesquisa
```

ou localizar um registo apenas por parte de um campo (República no campo ENDEREÇO, por exemplo)

```
LOCATE FOR "República"$ENDEREÇO
```

O comando CONTINUE permite encontrar o registe seguinte que satisfaz as mesmas condições. Os dois grupos de instruções que se seguem executam exactamente a mesma função:

```
USE endereços
LOCATE FOR apelido=mapelido
DO WHILE apelido=mapelido .AND. .NOT. EOF()
     ...
     CONTINUE
ENDDO

USE endereços
SCAN FOR apelido=mapelido
     ...
ENDSCAN
```

3.3 Programação com dBase IV

Nas secções que se seguem iremos analisar algumas componentes essenciais ao desenvolvimento de uma aplicação em dBase IV. Iremos ver como desenhar menus, como programar entradas e saídas, assim como técnicas de edição e eliminação de registos. Finalmente, iremos falar sobre relatórios e como os gerar.

3.3.1 Desenhar menus e janelas

Num sistema conduzido por menus, o utilizador, para executar as diversas tarefas do programa, não necessita conhecer o nome dos vários ficheiros usados nesse programa. Os menus apresentarão as diversas funções, bastando ao utilizador escolher uma delas.

Apenas a título de exemplo, suponhamos que pretendíamos desenhar uma aplicação com a seguinte estrutura de menus:

Esta é uma versão muito simplificada e incompleta do que é um verdadeiro sistema de inventário. No entanto, as nossas intenções, aqui e agora, são apenas ilustrar as várias facetas de manipulação de uma base de dados. Por isso, embora possivelmente mais correcta, uma outra estrutura para o programa de inventário iria trazer consigo uma complexidade desnecessária neste momento.

Há essencialmente três tipos de menus suportados pelo dBase IV. São eles o de múltiplos itens, o de barra e o chamado *pop-up* . O menu de múltiplos itens apresenta uma lista de opções da qual o utilizador selecciona una única escolha. Devido à sua simplicidade é muito usado nas aplicações. Para desenhar um menu para o sistema de inventário podíamos usar o seguinte programa:

```
* menu principal
* exemplo de menus de múltiplos itens
set talk off
clear all
set console on
set device to screen
set print off
mtitulo = 'INVENTÁRIO'
save to titulo
do while .t.
  clear all
  clear
  restore from titulo
  @  1,1 say '**********'+mtitulo+'**********'
  @  2,1 say 'MENU PRINCIPAL'
  @  2,60 say dtoc(date())
  @  4,1 say OPÇÕES :'
  @  5,1 say '   D   DISQUETES'
  @  6,1 say '   M   MANUAIS'
  @  7,1 say '   H   HARDWARE'
  @ 10,1 say '   S   SAIR PARA O dBASE'
  @ 12,1 say '   X   SAIR PARA O SISTEMA OPERATIVO'
```

```
  @ 15,1 say 'ESCOLHA UMA OPÇÃO: '
  store ' ' to opcao
  wait to opcao
  store upper(opcao) to opcao
  do case
     case opcao='D'
          do InvDis
     case opcao='M'
          do InvMan
     case opcao='H'
          do InvHar
     case opcao='S'
          cancel
     case opcao='X'
          clear all
          quit
     otherwise
          @ 23,1 say 'ERRO: OPÇÃO INVÁLIDA'
          store 1 to xx
          @ 23,60 say chr(7)
          do while xx<35
               store xx+1 to xx
          enddo
  endcase
enddo

procedure InvDis
clear
? 'Inventário de disquetes'
wait
return

procedure InvMan
clear
? 'Inventário de Manuais'
wait
return
```

```
procedure InvHar
clear
? 'Inventário de Hardware'
wait
return
```

Podemos também criar menus de barra horizontal no cimo do écran e colocar os sub-menus como barras verticais. O utilizador move-se nas barras por intermédio das setas e carrega no Enter para activar a escolha. Vejamos um exemplo de menus de barras:

```
* menu principal
* Exemplo de menus de barra
set status off
set heading off
set safety off
set talk off
clear all
do DefMenu  &&  definição do menu
set clock on
activate menu menupr
release menu menupr
release menu menuDis
release menu menuMan
release menu menuHar
set clock off
cancel
* fim do programa principal

procedure DefMenu
* definição  do menu principal
define menu menupr message 'Menu principal do Inventário'
define pad Dis of menupr prompt 'Disquetes' at 0,4
define pad Man of menupr prompt 'Manuais' at 0,16
define pad Har of menupr prompt 'Hardware' at 0,26
```

```
define pad Fim of menupr prompt 'Fim' message 'Sair para o dBase' at 0,37

* activar os menus descendentes a partir do menu principal
on selection pad Dis of menupr activate menu Disquetes
on selection pad Man of menupr activate menu Manuais
on selection pad Har of menupr activate menu Hardware
on selection pad Fim of menupr deactivate menu

* definição do menu Disquetes
define menu Disquetes message 'Inventário' de disquetes'
define pad ConDis of Disquetes prompt 'Consultar disquetes' at 1,4
define pad AdiDis of Disquetes prompt 'Adicionar disquetes' at 2,4
define pad EliDis of Disquetes prompt 'Eliminar disquetes ' at 3,4
define pad LisDis of Disquetes prompt 'Listar disquetes   ' at 4,4

* selecção o dentro do menu Disquetes
on selection pad ConDis of Disquetes do InvDis
on selection pad AdiDis of Disquetes do InvDis
on selection pad EliDis of Disquetes do InvDis
on selection pad LisDis of Disquetes do InvDis
return

* este é o primeiro dos submenus
procedure InvDis
do case
  case pad()='CONDIS'
    @ 12,10 say 'Consultar disquetes'
    wait
    @ 1,10 clear
  case pad()='ADIDIS'
    12,10 say 'Adicionar disquetes'
    wait
    @ 1,10 clear
case pad()='ELIDIS'
    @ 12,10 say 'Eliminar disquetes'
    wait
    @ 1,10 clear
```

```
case pad()='LISDIS'
    @ 12,10 say 'Listar disquetes'
    wait
    @ 1,10 clear
endcase
return
```

O procedimento DefMenu é usado para definir o menu principal usando o comando

DEFINE MENU <menu> [MESSAGE "<mensagem longa>"]

Uma vez o menu definido, é mantido em memória até ser libertado ou apagado. A mensagem longa definida pela opção MESSAGE é escrita no centro da última linha do écran.

O passo seguinte é definir cada opção (PAD) do menu através do comando

DEFINE PAD <opção> OF <menu> PROMPT "<mensagem curta>"

[AT <lin>,<col>] [MESSAGE "<mensagem longa>"]

O nome do menu é o definido pelo comando DEFINE MENU. A mensagem curta é o texto que aparece no écran para selecção.

Para cada opção (PAD) podemos também definir uma acção que será iniciada sempre que a tecla RETURN for carregada após selecção de opção.

ON SELECTION PAD <opção> OF <menu> [<comando>]

Para apresentar um menu, temos de o activar através do comando

ACTIVATE MENU <menu> [PAD <opção>]

Este último comando faz aparecer o menu e espera que haja uma selecção, comportando-se de uma maneira semelhante à do comando READ. A opção PAD permite-nos seleccionar por defeito uma qualquer opção. Para desactivar um menu, podemos usar a tecla ESC ou o comando DEACTIVATE MENU.

Em dBase podemos também criar menus pop-up, extremamente úteis quando pretendermos que o utilizador seleccione opções em sub-menus ou listas. Vejamos um exemplo:

```
* Menu principal
* Exemplo de menus pop-ups
set status off
set heading off
set safety off
set talk off
clear all
tit1='INVENTÁRIO'
define window ecran from 1,0 to 23,79 double
define window pausa from 15,00 to 19,79 double
define window sair from 11,17 to 15,62 double
do DefMenu  && definição dos menus
set clock on
activate menu menupr
release menu menupr
release popups Disquetes
release popups Manuais
release popups Hardware
set clock off
cancel
* fim do programa principal
```

```
procedure DefMenu
* definição do menu principal
define menu menupr
define pad Dis of menupr prompt 'Disquetes';
    message 'Inventário de disquetes' at 0,4
define pad Man of menupr prompt 'Manuais';
    message 'Inventário de manuais' at 0,16
define pad Har of menupr prompt 'Hardware';
    message 'Inventário de hardware' at 0,26
define pad Fim of menupr prompt 'Fim';
    message 'Sair para o dBase' at 0,37

* activar os menus descendentes a partir do menu principal
on selection pad Dis of menupr do popini with 'Disquetes'
on selection pad Man of menupr do popini with 'Manuais'
on selection pad Har of menupr do popini with 'Hardware'
on selection pad Fim of menupr deactivate menu

* definição do menu Disquetes
define popup Disquetes from 10,30 message 'Inventário' de disquetes'
define bar 1 of Disquetes prompt ' Inventário de disquetes' skip
define bar 3 of Disquetes prompt ' Consultar disquetes'
define bar 4 of Disquetes prompt ' Adicionar disquetes'
define bar 5 of Disquetes prompt ' Eliminar disquetes'
define bar 6 of Disquetes prompt ' Listar disquetes'
define bar 7 of Disquetes prompt ' Voltar ao menu principal '
* seleção dentro do menu de Disquetes
on selection popup Disquetes do InvDis

* definição do menu Manuais
define popup Manuais from 10,30 message 'Inventário de manuais'
define bar 1 of Manuais prompt ' Inventário de manuais' skip
define bar 3 of Manuais prompt ' Consultar manuais'
define bar 4 of Manuais prompt ' Adicionar mauais'
define bar 5 of Manuais prompt ' Eliminar manuais'
define bar 6 of Manuais prompt ' Listar os manuais'
define bar 7 of Manuais prompt ' Voltar ao menu principal '
```

```
* seleção dentro do menu Manuais
on selection popup Manuais do InvMan

* definição  do menu Hardware
define popup Hardware from 10,30 message 'Inventário  de hardware'
define bar 1 of Hardware prompt ' Inventário  de Hardware' skip
define bar 3 of Hardware prompt ' Consultar equipamento'
define bar 4 of Hardware prompt ' Adicionar equipamento'
define bar 5 of Hardware prompt ' Eliminar equipamento'
define bar 6 of Hardware prompt ' Listar equipamento'
define bar 7 of Hardware prompt ' Voltar ao menu principal '
* seleção dentro do menu de Hardware
on selection popup Hardware do InvHar
return

* este procedimento é usado por todos os popups
procedure popini
parameters menus
clear
activate popup &menus
return

procedure InvDis
do case
     case bar()=3
        do EdiDis
     case bar()=4
        do AdiDis
     case bar()=5
        do EliDis
     case bar()=6
        do LisDis
     case bar()=7
        deactivate popup
        return
endcase
clear
```

```
return

procedure InvMan
do case
     case bar()=3
       do EdiMan
     case bar()=4
       do AdiMan
     case bar()=5
       do EliMan
     case bar()=6
       do LisMan
     case bar()=7
       deactivate popup
       return
endcase
clear
return

procedure InvHar
do case
     case bar()=3
       do EdiHar
     case bar()=4
       do AdiHar
     case bar()=5
       do EliHar
     case bar()=6
       do LisHar
     case bar()=7
       deactivate popup
       return
endcase
clear
return
```

```
procedure pausa
parameters msg
*  msg é a linha de mensagem
activate window pausa
@ 1,1 say msg
wait 'Carregue numa tecla ...'
deactivate window pausa
return
```

Com os menus pop-up seleccionamos as opções através das setas ou da primeira letra de cada opção. O menu é definido usando o comando

DEFINE POPUP <popup> FROM <lin>,<col> [TO <lin>,<col>]

[MESSAGE "<mensagem longa>"]

A posição FROM indica o canto superior esquerdo do menu e a posição TO, quando existente, o canto inferior direito.

As barras de menu são definidas pelo comando

DEFINE BAR <número> OF <popup> PROMPT "<mensagem curta>"

[MESSAGE "<mensagem longa>"]

O número da barra define a posição da barra no menu, podendo nós usar a função BAR() para determinar o número da última barra seleccionada. Uma vez definidas as barras, para definir a acção a ser executada, usamos o comando

ON SELECTION POPUP <popup> <comando>

De seguida, iremos ver algumas regras gerais para o desenho de écrans e particularmente para o desenho de menus. Estas regras seguem de perto as indicações do *Common User Access* (CUA) da *System Application Architecture* (SAA) (Berry, 1988).

- Manter o écran simples, legível, com frases simples, precisas e num vocabulário entendível pelo utilizador.
- No cimo de cada écran, colocar um cabeçalho centrado no écran contendo, pelo menos, o nome da aplicação, quem é o responsável por ela (nome da empresa, por exemplo) e a data.
- Usar cores, caracteres carregados e vídeo invertido sempre que possível.
- Os écrans devem ser apresentados segundo uma sequência linear e lógica.
- Dar sempre ao utilizador a possibilidade de abortar uma má escolha.
- Evitar colocar comentários desnecessários ou sons invulgares.
- Usar caracteres gráficos para separar zonas de entrada de dados.
- Usar a última linha (linha 25).
- Em programas que utilizem vários écrans, o primeiro écran é normalmente usado para a introdução do valor da chave primária.

Usando estes princípios como base, as linhas de orientação que se seguem dizem respeito ao desenho de menus.

- No menu principal, as primeiras letras de cada opção devem ser maiúsculas e únicas de modo a permitir a escolha da opção apenas por uma letra.
- A primeira opção deve começar a quatro espaços da margem esquerda.
- Devem também existir quatro espaços entre as opções dos menus horizontais.
- A última opção deve ser para sair do menu.
- Caso exista uma opção de ajuda (help), esta deve ser colocada na penúltima posição.
- As restantes opções deverão ser apresentadas segundo a ordem normal da sua utilização.
- Nos menus pop-up debaixo do menu principal, colocar a primeira letra do menu pop-up debaixo da primeira letra da opção da barra.
- Usar uma linha dupla para os menus.
- Num menu pop-up, separar com uma linha grupos de itens relacionados.
- Usar reticências (...) a seguir a uma escolha que dê lugar a uma caixa de diálogo.
- Usar uma seta para a direita (→) a seguir a uma escolha que dê lugar a outro menu.
- Reduzir de intensidade ou apagar as opções não disponíveis a cada momento.
- A tecla ESCAPE deve ser usada para cancelar uma escolha e RETURN parar a aceitar.
- Nos menus principais, desenhar os menus pop-up de tal modo que uma combinação

Alt+<letra> possa seleccionar a opção desejada.

- Coloque um máximo de sete a nove itens em cada menu.

É também possível usarem-se janelas em dBase IV. As posições (linha, coluna) usadas nos comandos GET e SAY dizem respeito ao canto superior esquerdo da janela. Tal como nos menus, as janelas são definidas e mantidas em memória até serem expressamente apagadas ou libertadas. Vejamos um exemplo:

```
define window Introdução from 10,7 to 16,73 double
activate window Introdução
@ 1,23 say 'Programa de Inventário'
@ 3,6 say 'Copyright 1991, XPTO-Sistemas Informáticos'
wait ''
release window Introdução
return
```

Os menus também podem ser criados através do Gerador de Aplicações. Um primeiro esquema de programa é gerado automaticamente, podendo ser editado posteriormente.

3.3.2 Programar saídas de informação

Existem essencialmente duas técnicas para extrair informação do dBase. São elas os comandos ?/?? e SAY. O comando ? envia os resultados para o periférico de saída activado. A diferença entre ? e ?? reside no facto do comando ? produzir um salto de linha antes de escrever.

Para se enviarem os resultados para uma impressora, podemos proceder do seguinte modo:

```
SET PRINT ON
? 'Isto é um teste'
SET PRINT OFF
```

O comando ?/?? tem a seguinte forma geral:

?/?? [<expressão1> PICTURE <código>] [FUNCTION <lista de funções>] [AT <número>]

[STYLE <número do tipo de letra>] [,<expressão2> ...] [...]

A opção PICTURE (ver apêndice) permite-nos formatar os resultados. Por exemplo, a instrução

```
x='abcd'
? x PICTURE '!A!A'
```

escreverá AbCd.

As funções do comando PICTURE (ver apêndice) são precedidas do sinal @. Por exemplo, a instrução

```
x='abcd'
? x PICTURE '@!'
```

escreverá ABCD.

A opção AT especifica a coluna a escrever.

A opção STYLE permite definir os atributos do texto a ser impresso. Os valores possíveis são B(carregado), I(itálico), U(sublinhado), R(expoente) e L(índice). Esta opção permite também indicar o tipo de letra usada para a impressão, desde que especificada no ficheiro CONFIG.DB.

Por fim, podemos usar a função REPLICATE() para enviar vários caracteres do mesmo tipo, como por exemplo,

```
? REPLICATE ('*',10)
```

O comando @ ... SAY é do tipo

@ <lin,col> [SAY <exp> [PICTURE <formato>] [FUNCTION <função>]]

[COLOR [<padrão>] [.<expandida.]]

A opção COLOR permite-nos especificar a cor to texto ou do fundo. Em apêndice encontra-se a tabela das cores. Segundo esta tabela, a instrução

```
@ 1,1 SAY 'texto' COLOR GR+/BG
```

escreverá em amarelo com um fundo em azul turquesa (cyan).

O comando @ também pode ser usado para produzir molduras no visor. Por exemplo,

```
@ 1,1 TO 10,20 DOUBLE
```

cria uma moldura rectangular com uma linha dupla da posição 1,1 à posição 10,20. Se usarmos a palavra PANEL em vez de DOUBLE obtemos uma linha grossa. Se não usarmos nenhuma destas palavras, a linha será simples.

Como já vimos atrás, o comando CLEAR apaga todo o visor. No entanto, se pretendermos apagar apenas parte do visor, podemos usar o comando @ com as seguintes opções:

@ <lin>,<col>	apaga a linha lin a partir de col
`@ <lin>,<col> CLEAR`	apaga de lin,col até ao canto superior direito
`@ <lin1>,<col1> CLEAR TO <lin2>,<col2>`	apaga de lin1,col1 até lin2,col2

Podemos também usar este comando para colorir uma caixa no visor:

@ <lin1>,<col1> FILL TO <lin2>,<col2> [COLOR <atributo de cor>]

Por fim, com a função CHR() podemos escrever qualquer carácter no écran. Por exemplo, o pedaço de programa que se segue desenhará uma caixa no visor e colocará um título dentro dessa caixa:

```
Igual=replicate(chr(205),39)
clear
cima = chr(201)+igual+chr(187)
baixo= chr(200)+igual+chr(188)
meio = chr(186)+space(39)+chr(186)
@ 10,5 say cima
@ 11,5 say meio
@ 12,5 say meio
@ 13,5 say meio
@ 14,5 say meio
@ 15,5 say baixo
@ 12,12 say 'Isto é um teste'
```

3.3.3 Programar entradas de informação

Para se introduzir dados, o dBase dispõe de quatro comandos: WAIT, ACCEPT, INPUT e @...GET. O comando WAIT é útil para introduzir um carácter isolado, como por exemplo,

```
WAIT "Ligue a impressora e carregue numa tecla" TO x
```

O comando ACCEPT é usado para obter cadeias com até 254 caracteres de comprimento e tem a forma

ACCEPT ["<texto>"] TO <variável de memória>

O comando INPUT é semelhante ao comando ACCEPT com a excepção de que aceita qualquer tipo de variável. Não é muito aconselhável o seu uso uma vez que não há qualquer hipótese de validação.

A função INKEY() permite-nos, sem parar o programa, obter o valor ASCII da última tecla pressionada. O argumento facultativo desta função permite especificar o tempo, em segundos, de espera. Esta função é útil para se controlar o uso de teclas tais como PgUp, ←, → e HOME.

Finalmente, o comando @...GET permite introduzir informação em qualquer parte do visor e tem o seguinte formato:

```
@ <lin,col>
     [ SAY <exp>  [ PICTURE <expC> ]  [ FUNCTION <lista de funções>] ]
     [ GET <variável>
          [ [ OPEN ]  WINDOW  <nome da janela> ]
          [ PICTURE <expC> ]  [ FUNCTION <lista de funções> ]
               [ RANGE <exp1,exp2> ]
               [ VALID <condição>  [ ERROR <expC> ] ]
               [ WHEN <condição> ]  [ DEFAULT <exp> ]
               [ MESSAGE <expC> ] ]
     [ COLOR [ <padrão> ]  [ , <expandida> ] ]
```

onde

SAY	define texto ou valores a aparecerem no écran
PICTURE	define esquema ou função para o valor de entrada ou saída
FUNCTION	define função para o valor de entrada ou saída
GET	define variável de entrada
WINDOW	define janela para o campo memo (OPEN abre-o automaticamente)
RANGE	define intervalo válido para a variável de entrada
VALID	define uma condição de validade para a variável de entrada

ERROR	define uma mensagem para valor de entrada errado
WHEN	define uma condição para o GET ser activado
DEFAULT	define um valor por defeito para uma variável de entrada
MESSAGE	define uma mensagem para a linha 23
COLOR	define as cores para as variáveis

Como já vimos atrás, só através de uma instrução READ é que os comandos GET são executados, e isto segundo a ordem em que aparecem no programa. No entanto, antes de se usar uma variável no comando GET, esta deve ser inicializada com um qualquer valor do mesmo tipo e tamanho do valor que se pretende ler.

A opção PICTURE fornece-nos um processo rápido de verificação dos dados entrados. Vejamos alguns exemplos.

```
@  8,19 SAY 'Número de conta
@  8,37 GET num PICTURE '9999'
@  9,19 SAY 'Descrição'
@  9,37 GET desc PICTURE 'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX'
@ 10,19 SAY 'Tipo'
@ 10,37 GET tipo PICTURE '!'
@ 11,19 SAY 'Preço'
@ 11,37 GET Custo PICTURE '99999999.99'
```

Para definir limites mínimo e máximo aceitáveis para a variável a entrar podemos usar a opção RANGE. Por exemplo, se pretendermos que o número de conta seja um numérico entre 0 e 3000, escrevemos a instrução

```
@ 8,19 SAY 'Número de conta: ' GET num PICTURE '9999' RANGE 0,3000
```

A opção WHEN pode ser usada se quisermos validar a entrada de dados em determinadas situações. Por exemplo, se definirmos que só determinados utilizadores podem introduzir o número de conta, podemos escrever

```
@ GET num PICTURE '9999' WHEN numutil>3
```

Podemos também usar a opção VALID para determinar quais os valores aceitáveis. Nos seguintes três exemplos, a variável TIPO só poder aceitar os valores H e S:

```
@ 4,1 SAY 'Tipo (H/S): ' GET tipo PICTURE '!' VALID tipo$'HS'
@ 4,1 SAY 'Tipo (H/S): ' GET tipo PICTURE '!' VALID tipo$'HS'
    ERROR 'Os valores aceitáveis são H e S'
@ 4,1 SAY 'Tipo (H/S): ' GET tipo PICTURE '!' VALID tipo$'HS' DEFAULT 'S'
```

Para se garantir que não há duplicações sempre que se adiciona um registo a um ficheiro indexado podemos usar uma instrução do tipo

```
@  8,19 SAY 'Número de conta: ' GET num PICTURE '9999';
    VALID .NOT.SEEK(num);
    ERROR 'Este número de conta já existe no ficheiro';
    MESSAGE 'Para sair introduza um número em branco'
```

3.3.4 Adicionar registos a uma base de dados

Para podermos ilustrar os capítulos que se seguem, consideremos a base de dados de material informático (hardware) com a seguinte estrutura:

1	Número interno	NUM	N	6,0
2	Número de série	SN	C	10
3	Descrição	DESC	C	20
4	Localização	SALA	N	4,2
5	Sistema instalado	SIST	N	2,0
6	Data da aquisição	DATA	D	
7	Custo	CUSTO	N	11,2

O processo mais simples para adicionar registos a uma base de dados é através do comando APPEND. Os comandos EDIT e BROWSE também permitem acrescentar registos.

Usando o dBase no modo directo, podemos escrever

```
. USE hardware
. APPEND
```

e aparecerá um écran com os campos em branco para serem preenchidos. Em apêndice encontram-se os comandos de controlo para o APPEND. Se estivermos no Centro de Controlo, basta percorrer o ficheiro até ao fim e aparecerá uma frase perguntando se pretendemos adicionar mais registos. Neste modo a tecla F2 permite-nos comutar entre os modos de edição (EDIT) e de leitura (BROWSE).

Se estivermos a usar um ficheiro indexado e pretendermos que o índice seja automaticamente actualizado sempre que um registo é introduzido, basta usarmos a opção `USE hardware ORDER num`

```
APPEND
```

Se pretendermos que os campos apareçam no écran com uma determinada disposição, podemos criar um ficheiro de formato e seleccioná-lo antes de usar o comando APPEND. Um tal ficheiro de formato poderia ser, por exemplo,

```
@  5,16 to 20,53 double
@  8,19 say 'Número interno: ' get num
@ 12,19 say 'N/S: ' get sn
@ 13,19 say 'Descrição: ' get desc
@ 15,19 say 'Localização:  sala ' get sala
@ 15,43 say 'Sist ' get sist
@ 17,19 say 'Aquisição: ' get data
@ 17,39 get custo
```

Se este ficheiro se chamasse entrada.fmt poderia ser seleccionado através do comando

SET FORMAT TO entrada

O esquema geral para usar os comandos SET FORMAT e APPEND é o seguinte:

```
*  abrir as bases de dados
*  inicializar as variáveis de memória
SET FORMAT TO entrada   &&  abrir o ficheiro de formato
READ
SET FORMAT TO           &&   fechar o ficheiro de formato
APPEND BLANK            &&   adicionar um registo em branco
*  usar o comando REPLACE para colocar os dados no registo
```

Vejamos agora um exemplo de programa para adicionar um novo registo à base de dados de equipamento.

```
*  adicionar equipamento ao ficheiro hardware
*  ADIHAR
*  nota: este programa não contem todos os campos

*  iniciar e abrir bases de dados
CLOSE DATABASES
```

```
mensagem='0'
ON ERROR mensagem=LTRIM(STR(ERROR()))+' '+MESSAGE()
STORE 'HARDWARE' to x
USE (x) ORDER num
ON ERROR
erro=VAL(mensagem)
IF erro>0
          DO Pausa WITH 'Erro ao abrir HARDWARE.DBF'
          erro=0
          RETURN
ENDIF
RELEASE mensagem

*  criação de títulos
tit2='ADICIONAR HARDWARE'
cp=len(tit2)  &&  comprimento do título do programa
initit=(79-cp-6)/2
STORE initit+cp+6 TO fimtit

*  definir as janelas
DEFINE WINDOW titulo FROM 3,initit TO 8,fimtit DOUBLE
DEFINE WINDOW chave FROM 11,16 TO 15,65 DOUBLE
DEFINE WINDOW dados FROM 11,16 TO 20,64 DOUBLE
*  permanecer no ciclo até tudo estar acabado
DO WHILE .T.
          *  apresentar o primeiro écran para obter a chave primária
          ACTIVATE WINDOW ecran
          CLEAR
          ACTIVATE WINDOW titulo
          inicio=(cp-len(tit1))/2
          @ 1,inicio+3 SAY tit1
          @ 2,3 SAY tit2
          mnum=0
          ACTIVATE WINDOW chave
          @ 1,1 SAY 'Introduza o número interno a adicionar ' GET mnum;
               PICTURE '999999' VALID .NOT. SEEK(mnum);
               ERROR 'Este número já existe no ficheiro';
```

```
        MESSAGE 'Introduza zero para sair'
    READ
    IF Vmnum=0
        RELEASE WINDOW titulo
        RELEASE WINDOW chave
        RELEASE WINDOW dados
        DEACTIVATE WINDOW ecran
        CLOSE DATABASES
        RETURN
    ENDIF
    DEACTIVATE WINDOW chave
    STORE SPACE(20) to mdesc
    STORE 0.00 TO mcusto
    *  apresentar o segundo écran para os dados não-chave
    ACTIVATE WINDOW dados
    @ 1,1 SAY 'NUMERO INTERNO '+str(mnum)
    @ 2,1 SAY 'DESCRIÇÃO ' GET mdesc ;
        MESSAGE 'Branco para sair sem adicionar registo'
    @ 3,1 SAY 'CUSTO ' GET mcusto PICTURE '99999999.99' ;
        MESSAGE 'Branco para sair sem adicionar registo'
    READ
    DEACTIVATE WINDOW dados
    IF SUBSTR(mdesc,1,2)='  '
        LOOP
    ENDIF

    *  gravar na base de dados
    APPEND BLANK
    REPLACE num WITH mnum
    REPLACE desc WITH mdesc
    REPLACE custo WITH mcusto
ENDDO
```

Há também algumas regras gerais que devemos tentar seguir, sempre que possível, quando escrevemos programas para adicionar registos em bases de dados.

- Se a introdução dos dados é feita para um ficheiro indexado, usar um primeiro écran para entrar apenas o valor da chave primária. Este valor deverá ser validado e analisado para evitar duplicações.
- Usar um segundo écran para fazer entrar todos os outros dados.
- Não acrescentar (append) nenhum registo em branco antes de verificar todos os dados introduzidos.
- Em cada écran, informar o utilizador do que deve fazer para sair sem gravar.
- Usar as regras de desenho de écrans.
- Antes dos dados de entrada serem colocados nos respectivos campos da base de dados, usar variáveis de memória previamente inicializadas.
- Prever todas as possíveis situações de erro e enviar mensagens claras ao utilizador.

3.3.5 Editar registos numa base de dados

Nesta secção iremos ver como editar ou alterar os dados existentes numa base de dados. Estando a usar o Centro de Controlo, seleccionamos a base de dados e pedimos para ver os registos. A tecla F2 permite-nos comutar entre um écran com apenas um registo (modo EDIT) ou com vários registos (modo BROWSE). Dentro de cada registo ou entre registos, podemos mover o cursor de acordo com os comandos apresentados em apêndice.

Se conhecermos o número do registo a editar e não estivermos sob o comando do Centro de Controlo, poderemos posicionar o ponteiro da base de dados nesse registo e editá-lo.

```
GOTO 3                                  EDIT3
EDIT
```

Caso se não conheça o número do registo, mas apenas o conteúdo de um determinado campo, podemos proceder do seguinte modo:

```
USE hardware
LOCATE FOR num='123456'
```

Para ficheiros indexados, usamos os comandos SEEK ou FIND:

```
USE hardware ORDER num
STORE '123456' TO mnum
SEEK

USE hardware ORDER num
STORE '123456' TO mnum
FIND &mnum
```

Como é evidente, devemos sempre garantir que o registo pretendido foi encontrado.

```
SEEK mnum
IF .NOT. FOUND()
     @ 23,1 SAY mnum+' NÃO EXISTE'
     DO menserro
     RETURN
ENDIF
EDIT
```

Também aqui podemos criar e usar ficheiros de formato para editar registos:

```
USE hardware
SET FORMAT TO edicao
GOTO 10
EDIT
```

Um outro comando usado para editar uma base de dados é o comando BROWSE. Com este comando podemos até apresentar e editar vários registos num só écran. É também possível especificar quais os campos que pretendemos listar e editar:

```
USE hardware
BROWSE FIELDS num,desc,custo
```

À semelhança do que vimos na secção anterior, para se construir um programa para editar registos de uma base de dados, podemos usar o gerador de aplicações. No entanto, a nós interessa-nos estudar como construir tais programas. Comecemos por analisar o seguinte programa:

```
* editar o ficheiro hardware
* EDIHAR
* iniciar e abrir bases de dados
close databases
mensagem='0'
on error mensagem=ltrim(str(error()))+' '+message()
x='HARDWARE'
use (x) order num
on error
erro=val(mensagem)
if erro>0
  do pausa with 'Erro ao abrir HARDWARE.DBF'
  erro=0
  return
endif
release mensagem


* iniciar o écran de títulos
activate window ecran
clear
tit2='EDITAR EQUIPAMENTO'
cp=len(tit2)
```

```
initit=(79-cp-6)/2
fimtit=initit+cp+6

* definir as janelas
define window titulo from 3,initit to 8,fimtit double
define window chave from 11,16 to 15,65 double
define window dados from 11,16 to 20,64 double

* permanecer no ciclo até tido estar acabado
do while .t.
    clear
    * apresentar o primeiro écran para obter a chave primária
    activate window titulo
    inicio=(cp-len(tit1))/2
    @ 1,inicio+3 say tit1
    @ 2,3 say tit2
    mnum=0
    activate window chave
    @ 1,1 say 'Introduza o número interno a editar ' get mnum ;
          picture '999999' valid (seek(mnum) .or. mnum=0) ;
          error 'Este número não existe no ficheiro' ;
          message 'Introduza zero para sair'
    read
    if mnum=0
          release window titulo
          release window chave
          release dados
          deactivate window ecran
          close databases
          return
    endif
    deactivate window chave

    * obter os valores actuais
    mnum=num
    mdesc=desc
    mcusto=custo
```

```
        * apresentar o segundo écran e obter os valores não-chave
        activate window dados
        @ 1,1 say 'NÚMERO INTERNO '+str(mnum)
        @ 4,1 say 'DESCRIÇÃO ' get mdesc ;
              message 'Ctrl/Q para sair sem alterações'
        @ 5,1 say 'CUSTO ' get mcusto picture '99999999.99' ;
              message 'Ctrl/Q para sair sem alterações'
        read
        deactivate window dados

        * actualizar os campos da base de dados
        replace num with mnum
        replace desc with mdesc
        replace custo with mcusto
    enddo
```

Ao comparar este programa com o de adicionar registos vemos que:

- na obtenção do número interno, a validação é agora feita para garantir que o registo existe;
- quando o registo ´´localizado, os valores dos campos são colocados em variáveis de memória;
- não há APPEND BLANK e os valores são simplesmente substituídos.

As seguintes são algumas regras que deveremos seguir sempre que pretendemos editar registos:

- Se se pretender alterar um ficheiro indexado, usar um primeiro écran para introduzir apenas o valor da chave. Este valor deve ser verificado para se garantir que o registo pretendido existe.
- Ter um segundo écran para apresentar e editar os valores não chave.

- Verificar todas as variáveis de memória.
- Informar o utilizador de como deve sair sem alterar o registo.
- Seguir as orientações para o desenho de écrans.
- Usar variáveis de memória para temporariamente manter os valores iniciais até estarmos aptos a editá-los.
- Estar atento às condições de erro e usar mensagens simples.
- Evitar usar muitos índices no ficheiro a ser actualizado, pois isso torna o processo mais lento. Evitar também abrir e fechar ficheiros durante o ciclo de entrada.
- Evitar editar o valor da chave. Se se pretender alterar o valor da chave é melhor eliminar o registo e entrar um novo.

3.3.6 Eliminar registos de uma base de dados

Nesta secção analisaremos algumas estratégias para eliminar registos de uma base de dados. Antes porém, é necessário saber-se que, associado a cada registo, existe uma variável onde, através da instrução DELETE, se marca o registo para ser apagado. Estes registos marcados são assinalados com um asterisco nas listagens produzidas pelo comando LIST, não são copiados pelo comando COPY, podem ser recuperados pelo comando RECALL e são definitivamente apagados da base de dados pelo comando PACK. Para se apagar todos os registos de uma base de dados podemos, no entanto, usar o comando ZAP.

O comando PACK actualiza automaticamente os índices, desde que abertos com o ficheiro. Se, a qualquer altura, necessitarmos de re-indexar um ficheiro, basta usarmos o comando REINDEX.

Um dos métodos possíveis para marcar um registo para apagamento é usar Ctrl-U com o comando EDIT. Outro processo é escrever um programa como o que se segue.

```
* eliminar registos do ficheiro hardware
* ELIHAR
* iniciar e abrir bases de dados
close databases
mensagem='0'
on error mensagem=ltrim(str(error()))+' '+message()
x='HARDWARE'
use (x) order num
on error
erro=val(mensagem)
if erro>0
  do pausa with 'Erro ao abrir HARDWARE.DBF'
  erro=0
  return
endif
release mensagem

* iniciar o écran de títulos
tit2='ELIMINAR EQUIPAMENTO'
cp=len(tit2)
initit=(79-cp-6)/2
fimtit=initit+cp+6
flgeli=.f.

* definir as janelas
define window titulo from 3,initit to 8,fimtit double
define window chave from 11,16 to 15,65 double
define window dados from 11,16 to 20,64 double

* permanecer no ciclo até tido estar acabado
do while .t.
    clear
    activate window ecran
```

```
clear
* apresentar o primeiro écran para obter a chave primária
activate window titulo
inicio=(cp-len(tit1))/2
@ 1,inicio+3 say tit1
@ 2,3 say tit2
mnum=0
activate window chave
@ 1,1 say 'Introduza número interno para eliminar ' get mnum ;
      picture '999999' valid (seek(mnum) .or. mnum=0) ;
      error 'Este número não existe no ficheiro' ;
      message 'Introduza zero para sair'
read
if mnum=0
      release window titulo
      release window chave
      release dados
      if flgeli
         activate window sair
         @ 1,1 say 'A RE-INDEXAR...'
         pack
         deactivate window sair
      endif
      deactivate window ecran
      close databases
      return
endif
deactivate window chave

* obter os valores actuais
mnum=num
mdesc=desc
mcusto=custo
melimina='N'

* apresentar o segundo écran e perguntar se quer eliminar
activate window dados
```

```
        @ 1,1 say 'NÚMERO INTERNO '+str(mnum)
        @ 3,1 say 'DESCRIÇÃO '+mdesc
        @ 4,1 say 'CUSTO '+str(mcusto,11,2)
        @ 6,1 say 'APAGO ESTE REGISTO? ' get melimina picture '!' ;
             valid melimina$'SN' error 'Responda S ou N'
        read
        deactivate window dados

        * eliminar a pedido
        if melimina='S'
             * alterar a flag e eliminar o registo
             flgeli=.t.
             activate window sair
             @ 2,1 say 'A APAGAR...'
             delete
             deactivate window sair
        endif
    enddo
```

Tal como nos programas anteriores a chave é introduzida, validada e os valores do registo são apresentados ao utilizador. É então perguntado se pretende mesmo apagar o registo. No caso de resposta afirmativa, o registo é apagado e o índice reconstruído.

Apresentam-se de seguida algumas regras gerais para o desenho de programas de eliminação de registos em base de dados.

- Se quisermos eliminar um registo de um ficheiro indexado, usar um primeiro écran para introduzir o valor da chave.
- Um segundo écran apresenta os valores do registo a apagar, para que o utilizador se certifique antes de confirmar a eliminação.
- Em cada écran, informar o utilizador do que deve fazer para sair sem apagar.

- Seguir as indicações gerais para o desenho de écrans.
- Detectar todas as possíveis situações de erro.
- Evitar usar ficheiros com muitos índices.

3.3.7 Emitir relatórios

Em dBase IV, podemos criar relatórios de variadíssimas maneiras:

- usando comandos de saída, tais como LIST e DISPLAY;
- criando relatórios rápidos através da tecla shift-F9;
- usando o gerador de relatórios do Centro de Controlo para criar esquemas de relatórios ou os próprios programas;
- usando os comandos ?, ?? ou @...SAY em programas.

Pelas mesmas razões já atrás enunciadas, debruçar-nos-emos um pouco mais sobre a última das hipóteses de criar relatórios.

Os comandos ?\?? têm o seguinte formato:

```
?\??  <exp>  [ PICTURE  <opção> ]  [ FUNCTION <lista de funções> ]
 [ AT <expN> ]  [ STYLE <número de tipo de letra> ]  [ , <exp2> ...]
```

Os comandos SET PRINT ON/OFF e SET CONSOLE ON/OFF são usados para controlar o envio de resultados para a impressora e para o écran. Os saltos de página são conseguidos através do comando EJECT. Percorrer o ficheiro com a ajuda dos comandos DO WHILE ou SCAN e usar o comando SKIP para excluir registos do relatório.

Uma outra hipótese de se produzir relatórios é usando, como aliás já atrás foi mencionado, o comando @...SAY. É preciso apenas ter em atenção que numa impressora, as linhas são escritas da esquerda para a direita e de cima para baixo.

As instruções que se seguem escrevem um cabeçalho.

```
set device to print
@ 1,1 say 'Impressão do Inventário'
@ 1,60 say dtoc(date())
set device to screen
```

Uma situação que importa ter em atenção é o facto de algumas impressoras manterem a última linha a imprimir em memória e só a escreverem quando receberem um comando de fim de linha. Qualquer uma das seguintes séries de comandos resolve o problema.

```
set print on
eject
set print off
```

```
set print on
?? chr(13)
set print off
```

De seguida apresentam-se algumas regras gerais para programas de emissão de relatórios.

- Dar sempre ao utilizador a possibilidade de abortar o programa antes deste começar a escrever. Se a saída é para a impressora pelo menos perguntar se a impressora está em condições.

- Enquanto a impressora está e escrever o relatório enviar uma mensagem para o écran a informá-lo.

- Caso o comprimento das linhas a imprimir seja menor ou igual a 80, dar ao utilizador a hipótese de obter o relatório no écran.

- Garantir que a última linha é sempre escrita.

- Colocar num procedimento a programação necessária à escrita dos cabeçalhos.

- Caso já não exista um índice, criar um índice temporário antes de começar a impressão.
- Para imprimir um texto a partir de um programa, pode usar-se o comando TEXT...ENDTEXT.
- É preferível usar-se o comando ?/?? para criar os relatórios na impressora e o comando @...SAY para écran. Usar o comando ?/?? também sempre que escrever para um ficheiro.

3.3.8 Usar memorandos

Os campos de memorandos, adiante chamados campos memo, permitem-nos incluir textos de comprimento variável, sendo fisicamente armazenados num ficheiro distinto e com extensão .DBT. Não é possível criar índices em campos memo, mas estes podem ser pesquisados.

Ao criar um campo memo, imediatamente aparece na estrutura o comprimento 10 correspondente ao espaço ocupado pelo ponteiro que liga este campo ai ficheiro .DBT. Quando estivermos a editar os registos imediatamente notamos que os campos memo terão como conteúdo a palavra 'memo' em minúsculas. Esta palavra passa a maiúsculas quando o referido campo contiver alguma informação.

Para carregar informação num campo memo, basta colocar o cursor nesse campo e carregar numa das teclas `Ctrl-Home` ou `F9`. É então aberto uma janela ocupando todo o écran e na qual a edição do texto se faz segundo as regras gerais de qualquer processador de textos, nomeadamente do WordStar. Em apêndice, encontra-se uma lista dos comandos usados para editar os campos de tipo memo. Comandos especiais para copiar, cortar e colar texto ou criar saltos de página são executados através de menus e teclas de função.

Usando um desenho de écran específico (ficheiro com extensão .FMT), é possível fazer com que o campo memo seja colocado no mesmo écran que os restantes campos.

3.3.9 Ordenar uma base de dados

Nesta última secção iremos ver um comando que nos permite alterar fisicamente a ordem dos registos de uma base de dados. Tem a forma

```
SORT TO <ficheiro> ON <campo> [ /A ] [ /D ] [ /C ]
  [ campo2 [ /A ] [ /D] [ /C ] … ] [ <amplitude> ]
[ ASCENDING / DESCENDING ] [ FOR <EXP> ] [ WHILE <EXP> ]
```

O ficheiro de destino não pode ser o mesmo do de origem. No entanto, como a indexação é uma operação rápida, normalmente nõa necessitamos de utilizar o comando SORT; sempre que precisarmos de ordenar registos para um relatório, basta indexar temporariamente o ficheiro de dados.

Este comando é útil, por exemplo, juntamente com o comando COPY, para segmentar uma base de dados que tenha atingido uma proporção tal que a impeçam de ser copuada para disquete. Depois de fisicamente re-arranjar os registos e de determinar quais os pontos de corte, podemos usar um comando como o seguinte:

```
COPY TO <ficheiro> FOR RECNO() < 100
```

Para finalizar estas notas aconselhamos a consulta dos manuais do dBase, quer para especificações mais detalhadas, quer para outros comandos não apresentados nestas notas.

Bibliografia

Atre, S. (198x). *Database. Structural techniques for design, performance, and management* . Wiley-Interscience.

Berry, R. E. (1988). Common User Access - A consistent and usable human-computer interface for the SAA environments. *IBM Systems Journal* .

Borland International (1990). *Turbo Pascal user's guide* . Scotts Valley, CA: Borland.

Elmasri, R. e Navathe, S. B. (1989). *Fundamentals of database system* . Redwood City, CA: The Benjamin/Cummings Publ Co.

Grogono, P. (1980). *Programming in Pascal* . Reading, MA: Addison-Wesley.

Horowitz, E., & Sahni, S. (1976). *Fundamentals of data structures* . London: Pitman.

Jensen, K., & Wirth, N. (1974). *Pascal. User manual and report* . Berlin: Springer-Verlag.

Martin, J. (1980). *Managing the data base environment* .Savant Institute.

Parsaye, K., Chignell, M., Khoshafian, S. e Wong, H. (1989). *Intelligent databases. Object oriented, deductive hypermedia technonolies* . New York, NY: John Wiley.

Pratt, P. J. (1991). *Microcomputer database management using dBase III Plus* . Boston: Boyd & Fraser Publ Co.

Townsend, C. (1991). *Mastering dBase IV 1.1 programming* . Alameda, CA: Sybex.

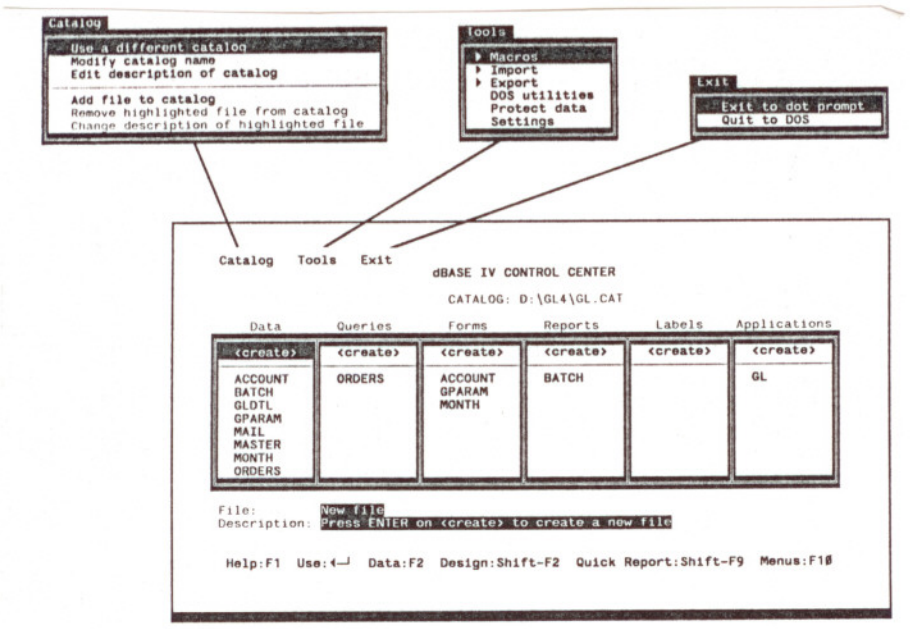
Tremblay, J.-P., Bunt, R. B., & Opseth, L. M. (1980). *Pascal estructurado.* Madrid: McGraw-Hill.

Tremblay, J.-P., & Bunt, R. B. (1983). *Ciência dos computadores. Uma abordagem algorítmica* São Paulo: McGraw-Hill.

Wirth, N. (1976). *Algorithms + Data Structure = Programs* . Englewood Cliffs, N. J.: Prentice-Hall.

Apêndices

Apêndice A - Centro de controlo



Apêndice B - Comandos do editor

KEYS	ACTION
↑, ↓, →, ←	Moves the cursor in the indicated direction
Backspace	Deletes character to left of cursor
Del	Deletes selected item
Home	Moves cursor to beginning of line
End	Moves cursor to end of line
Ctrl-PgUp	Moves to beginning of file
Ctrl-PgDn	Moves to end of file
Ctrl-A	Moves cursor back one word
Ctrl-C *or* PgDn	Scrolls screen down 18 lines
Ctrl-D	Moves cursor one space to right (same as → key)
Ctrl-E	Moves cursor up one line (same as ↑ key)
Ctrl-F	Moves cursor forward one word
Ctrl-G *or* Del	Deletes character at cursor
Ctrl-K R	Reads another file into edited file
Ctrl-K W	Writes file to another file
Ctrl-M *or* Enter	Moves cursor to next line and inserts a blank line
Ctrl-N	Inserts blank line after cursor
Esc *or* Ctrl-Q	Exits without saving new data
Ctrl-R *or* PgUp	Scrolls screen up 18 lines
Ctrl-S	Moves cursor one space left (same as ← key)
Ctrl-T	Deletes to beginning of next word
Ctrl-V *or* Ins	Toggles insert mode
Ctrl-W *or* Ctrl-End	Exits and saves changes
Ctrl-X	Moves cursor down one line (same as ↓ key)
Ctrl-Y	Deletes current line
Ctrl-Enter	Saves and stays in editor

Apêndice C - Comandos de APPEND, EDIT e BROWSE

▼ *APPEND Control Commands*

Ctrl-E	Backs up one field
Ctrl-A	Backs up one word
Ctrl-X	Advances to next field
Ctrl-F	Advances to next word
Ctrl-S	Backs up one character
Ctrl-D	Advances one character
Ctrl-Y	Clears field to blanks
Ins *or* Ctrl-V	Turns insert mode off or on
Ctrl-G	Aborts character under cursor
Backspace	Deletes characters to left of cursor
Ctrl-Q *or* Esc	Aborts screen append without adding record
Ctrl-C	Aborts screen append and adds record or (if at least one field has been entered) goes to next record entry
Ctrl-W	Saves and exits
Ctrl-Home	Enters memo field data
Ctrl-End	Exits from memo field entry
Home *or* Ctrl-Z	Goes to beginning of line
End *or* Ctrl-B	Goes to end of line

▼ *The EDIT and BROWSE Commands*

Ctrl-E	Backs up one field
Ctrl-A	Backs up one word
Ctrl-X	Advances to next field
Ctrl-F	Advances to next word
Ctrl-S	Backs up one character
Ctrl-D	Advances one character
Ctrl-Y	Clears field to blanks
Ins *or* Ctrl-V	Turns insert mode off or on
Ctrl-G *or* Del	Aborts character under cursor
Backspace	Deletes characters to left of cursor
Ctrl-Q *or* Esc	Aborts screen edit without changes
Ctrl-End *or* Ctrl-W	Exits from screen edit and updates record
Ctrl-R *or* PgUp	Backs up one page or record
Ctrl-C *or* PgDn	Goes forward one page or record
Ctrl-B *or* End	Scrolls horizontally to the right
Ctrl-Z *or* Home	Scrolls horizontally to the left
Ctrl-U	Deletes the record

Apêndice D - Lista de funções

CATEGORY	FUNCTION	DESCRIPTION
Date and Time	CDOW( )	Converts date variable to day of week (character string)
	CMONTH( )	Converts date variable to month (character string)
	CTOD( )	Converts character string to date variable
	DATE( )	Returns system date
	DAY( )	Returns day of month (numeric)
	DOW( )	Returns day of week (numeric) from date variable
	DTOC( )	Converts date to character
	DTOS( )	Converts date to character string*
	DMY( )	Converts date to DD/MM/YY format*
	MAX( )	Returns greater of two dates (or times)
	MIN( )	Returns lesser of two dates (or times)
	MDY( )	Converts date to MM/DD/YY format*
	MONTH( )	Returns month (numeric) from date variable
	TIME( )	Returns system time
	YEAR( )	Returns year (numeric) from date variable
String	&	Macro substitution
	AT( )	Searches substring
	LEFT( )	Selects substring from left
	LEN( )	Returns length of character string
	LIKE( )	Compares wildcard*
	LOWER( )	Converts to lowercase
	LTRIM( )	Removes leading blanks
	MAX( )	Returns greater of two strings
	MIN( )	Returns lesser of two strings
	REPLICATE( )	Repeats character expression
	RIGHT( )	Selects substring from right
	RTRIM( )	Removes trailing blanks
	SPACE( )	Outputs spaces
	STUFF( )	Replaces a portion of a string*
	SUBSTR( )	Creates substring
	TRANSFORM( )	Returns character/number in picture format
	TRIM( )	Removes trailing blanks
	UPPER( )	Converts to uppercase

CATEGORY	FUNCTION	DESCRIPTION
Mathematical	ABS( )	Returns absolute value
	ACOS( )	Returns arccosine*
	ASIN( )	Returns arcsine*
	ATAN( )	Returns arctan from tangent*
	ATAN2( )	Returns arctan from sine and cosine*
	CEILING( )	Rounds up*
	COS( )	Returns cosine*
	EXP( )	Returns exponent
	FIXED( )	Converts binary numbers (F) to BCD numbers (N)*
	FLOAT( )	Converts BCD numbers (N) to binary (F)*
	FLOOR( )	Rounds down*
	FV( )	Returns future value*
	INT( )	Returns integer
	LOG( )	Returns natural log
	LOG10( )	Returns common log*
	MAX( )	Returns greater of two values
	MIN( )	Returns smaller of two values
	MOD( )	Returns modulus
	PAYMENT( )	Calculates regular payment amount*
	PI( )	Returns pi*
	PV( )	Returns present value*
	RAND( )	Returns random number*
	ROUND( )	Rounds off
	SIGN( )	Returns sign of expression*
	SIN( )	Returns sine*
	SQRT	Returns square root
	TAN( )	Returns tangent
Conversion	ASC( )	Converts character to ASCII code
	CHR( )	Converts ASCII code to character
	DTOR( )	Converts degrees to radians*
	RTOD( )	Converts radians to degrees*
	STR( )	Converts numeric to character type
	VAL( )	Converts character to numeric type

CATEGORY	FUNCTION	DESCRIPTION
Miscellaneous	BOF( )	Returns beginning of file
	CALL( )	Executes binary program*
	COL( )	Returns current screen column
	DELETED( )	Returns deleted record
	DIFFERENCE( )	Calculates difference of two SOUNDEX codes (4=close match)*
	DISKSPACE( )	Returns free space on disk
	EOF( )	Returns end of file
	ERROR( )	Returns number of ON ERROR condition
	FILE( )	Tests for file existence
	FOUND( )	Returns result of database file search
	LEN( )	Returns length of character string
	IIF( )	Returns one expression or another (one-line IF...ENDIF)
	ISALPHA( )	Evaluates for letter
	ISCOLOR( )	Evaluates for color mode
	ISLOWER( )	Evaluates for lowercase
	ISUPPER( )	Evaluates for uppercase
	LINENO( )	Returns line number in program file*
	LOOKUP( )	Returns a specified field value for the first record in a database where a value matches a specified field*
	LUPDATE( )	Returns last update of database file
	MESSAGE( )	Returns ON ERROR message string
	PCOL( )	Returns printer column position
	PROW( )	Returns printer row position
	RECCOUNT( )	Returns number of records in database
	RECNO( )	Returns current record number in file
	RECSIZE( )	Returns size of record
	ROW( )	Returns current screen row
	RUN( )	Executes like the RUN command and returns zero
	SEEK( )	Looks up in indexed database
	SELECT( )	Returns number of next work area*
	SOUNDEX( )	Provides a sound-alike code*
	TRANSFORM( )	Permits PICTURE formatting of variable
	TYPE( )	Validates an expression

CATEGORY	FUNCTION	DESCRIPTION
Identification	ALIAS( )	Returns alias name of file in use*
	BAR( )	Returns name of bar
	CERROR( )	Returns last compiler error
	COMPLETED( )	Determines if a transaction is completed*
	DBF( )	Returns name of database file in use
	DELETED( )	Identifies if record is deleted
	DISKSPACE( )	Returns number of bytes remaining on disk
	FIELD( )	Returns name of specified field in database
	FKLABEL( )	Returns name of function keys
	FKMAX( )	Returns maximum number of function keys
	GETENV( )	Gets DOS environment variables
	ISMARKED( )	Denotes if file is in state of change (see COMPLETED( ))*
	KEY( )	Returns key expression for index file
	MDX( )	Returns name of open production file*
	MEMORY( )	Returns amount of unused memory*
	MENU( )	Returns name of active menu*
	NDX( )	Returns names of open index files
	ORDER( )	Returns name of index file or current tag*
	OS( )	Returns name of operating system
	PAD( )	Returns current menu pad*
	POPUP( )	Returns name of current pop-up*
	PRINTSTATUS( )	Returns true if print device is ready*
	PROGRAM( )	Returns name of program (or error)*
	PROMPT( )	Returns name of current menu prompt*
	ROLLBACK( )	Roll backs a transaction file*
	SET( )	Returns status of SET parameters
	TAG( )	Returns tag name in specified file*

CATEGORY	FUNCTION	DESCRIPTION
	VERSION( )	Returns dBASE IV version number
	WINDOW( )	Returns current active window
Memo Fields	MEMLINES( )	Returns number of lines in a memo field*
	MLINE( )	Extracts one line of text*
Input	INKEY( )	Returns key pressed during program execution
	LASTKEY( )	Returns decimal ASCII value of last key pressed*
	READKEY( )	Returns exit key from full-screen editing
	READVAR( )	(Used to create context-sensitive help screens)*
Network	ACCESS( )	Returns access level of user
	CHANGE( )	Determines if record has been changed*
	FLOCK( )	Locks file
	LKSYS( )	Returns log-in name of user and time and date of lock*
	LOCK( )	Locks records
	NETWORK( )	Returns true if system on network
	RLOCK( )	Locks records
	USER( )	Returns user name*

* *New in dBASE IV*

Apêndice E - Função PICTURE

▼ *PICTURE Format Functions*

CODE	MEANING
(	Encloses negative number in parentheses
!	Converts to capital letters
^	Displays in scientific notation
$	Displays data in currency format
A	Displays alphabetic characters only
B	Left-justifies numeric data
C	Displays CR (credit) after positive number
D	Uses SET DATE date format
E	Uses European date format
I	Centers text in field
J	Right-aligns text in field
L	Displays leading zeros
M	Permits entry from a list; the first value in the list will be displayed with GET, use spacebar to select
R	Causes literal characters to be inserted in display (defines a template with literal characters for template; see Chapter 19)
S(n)	Limits field width to n characters and horizontally scrolls contents in displayed window
T	Trims leading and trailing blanks
X	Displays DB (debit) after negative number
Z	Displays zero numeric data as a blank string

▼ *PICTURE Template*

CODE	MEANING
!	Converts to uppercase if lowercase; otherwise, displays as is
#	Allows only digits, operations (+, −, and so on), and spaces
$	Displays the SET CURRENCY symbol in place of leading zeros
*	Displays asterisks instead of leading zeros
,	Displays comma only if no numbers are in front of it
.	Specifies decimal position
9	(Same as #)
X	Allows any character to be entered
A	Allows only alphabetic characters (letters)
N	Allows letters and numbers
L	Allows only logical data
Y	Allows logical Y, y, N, and n only; converts to uppercase if necessary

Apêndice F - Lista de comandos dBase IV

CONVENTION	MEANING
[ ]	Specifies optional parameter
<*condition*>	Specifies a condition to be met
<*scope*>	Specifies how much of the database to use in the command (examples: NEXT 10, ALL, RECORD 3); when a FOR or WHILE is used, the default scope is ALL
uppercase letters	Indicates a command keyword
/	Indicates either command may be used (example: TO PRINTER/TO FILE <*file name*>)
<*prompt*>	Displays character string; delimiters required (example: "Test")
<*memvar*>	Specifies memory variable
<*memvarlist*>	Specifies a list of one or more memory variables
<*row,col*>	Specifies row-and column combination (example: 2,3).
<*exp*>	Specifies any dBASE IV expression
<*expC*>	Specifies any character string expression
<*expN*>	Specifies any numeric expression
<*exp list*>	Specifies list of one or more memory variables and operators
<*field list*>	Specifies a list of one or more data fields
<*file name*>	Specifies the name of a file with drive designator
<*template*>	Specifies characters used to control output format
<*commands*>	Specifies one or more dBASE IV commands
<*keyword*>	Specifies a verb recognized by dBASE IV or a dBASE IV parameter (examples: GOTO, ON, ALIAS)

? <*exp*$_1$> [PICTURE <*clause*>] [FUNCTION <*function list*>]
[AT <*expN*>] [STYLE <*font#*>][,<*exp*$_2$>...] Displays the expression list on the next line. †

?? <*exp*$_1$> [PICTURE <*clause*>] [FUNCTION <*function list*>]
[AT <*expN*>] [STYLE <*font#*>][,<*exp*$_2$>...] Displays the expression list on the current line. †

??? <*expC*> Sends output directly to printer.*

@ <*row,col*>
[SAY <*exp*> [PICTURE <*expC*>]
[FUNCTION <*function list*>]]
[GET <*variable*>
[[OPEN] WINDOW <*window name*>]
[PICTURE <*expC*>] [FUNCTION <*function list*>]
[RANGE [REQUIRED] [<*low*>][<*high*>]
[VALID [REQUIRED] <*condition*> [ERROR <*expC*>]]
[WHEN <*condition*>] [DEFAULT <*exp*>]
[MESSAGE <*expC*>]]
[COLOR [<*standard*>] [,<*enhanced*>]] Displays and gets formatted data lines. †

@ <*row,col*> CLEAR [TO <*row,col*>] Clears a portion of the screen.

@ <*row*$_1$,*col*$_1$> FILL TO <*row*$_2$,*col*$_2$> [COLOR <*color attr*>] Fills a rectangular region on the screen.*

@ <*row*$_1$,*col*$_1$> TO <*row*$_2$,*col*$_2$>
[DOUBLE/PANEL/ <*border definition string*>]
[COLOR <*color attr*>] Draws a box on the screen with a single line, double line, or a specified character. Also clears a box. †

ACCEPT [<*prompt*>] TO <*memvar*> Stores an input character string to a memory variable.

ACTIVATE MENU <*menu name*> [PAD <*pad name*>] Activates an existing bar menu and displays it.*

ACTIVATE POPUP <*popup name*> Activates a defined pop-up menu.*

ACTIVATE SCREEN Sends the output of the screen from the active window to the entire CRT screen, covering existing windows.*

ACTIVATE WINDOW <*window name list*> /ALL Activates and displays a defined window.*

APPEND [BLANK] Adds a record to the end of a database file.

APPEND FROM <*file name*>/? [FOR <*condition*>] [[TYPE] <*file type*>] Adds records to the end of a database file from a file (dBASE IV or non–dBASE IV). †

APPEND FROM ARRAY <*array name*> [FOR <*condition*>] Adds records to a database from an array.*

APPEND MEMO <*memo field name*> FROM <*file name*>
[OVERWRITE] Imports an ASCII file to the specified memo field.*

ASSIST Switches dBASE IV to the Control Center mode.

AVERAGE [<*expN list*>] [<*scope*>] [FOR <*condition*>]
[WHILE <*condition*>]
[TO <*memvarlist*>/TO ARRAY <*array name*>] Computes the arithmetic mean to a memory variable. ‡

BEGIN/END TRANSACTION [<*path name*>] Starts/ends transaction recording for rollback.*

BROWSE [NOINIT] [NOFOLLOW] [NOAPPEND] [NOMENU] [NOEDIT]
[NODELETE] [NOCLEAR] [COMPRESS] [FORMAT] [LOCK <*expN*>]
[WIDTH <*expN*>] [FREEZE <*field name*>]
[WINDOW <*window name*>]
[[FIELDS <*field name*$_1$> [/R]] [/<*column width*>]
/<*calculated field name*$_1$> = <*exp*$_1$>
[[,<*field name*$_2$> [/R]] [/<*column width*>]
/<*calculated field name*$_2$> = <*exp*$_2$>]...] Performs menu-driven full-screen database edit. †

CALCULATE [<*scope*>] <*option list*>
[FOR <*condition*>] [WHILE <*condition*>]
[TO <*memvarlist*>/TO ARRAY <*array name*>] Computes a database function.*

CALL <*module name*> [WITH <*expression list*>] Executes memory file loaded with LOAD. †

CANCEL Aborts program execution.

CHANGE [NOINIT] [NOFOLLOW]
[NOAPPEND] [NODELETE] [NOCLEAR]
[<*record number*>] [<FIELDS *field list*>] [<*scope*>]
[FOR <*condition*>] [WHILE <*condition*>]
Edits specified fields in a database. †

CLEAR Clears screen and repositions cursor to lower left of screen.

CLEAR ALL Closes all database, index, and format files. Clears relations. Releases all memory variables and selects first work area. (CLOSE DATABASES closes databases without releasing variables.)

CLEAR FIELDS Clears fields set with SET FIELDS command.

CLEAR GETS Releases current GET variables from READ. In dBASE IV, this is done automatically by the READ.

CLEAR MEMORY Erases all current memory variables.

CLEAR MENUS Clears menus from screen and memory.*

CLEAR POPUPS Clears all pop-up menus from screen and memory.*

CLEAR SCREENS Clears screens from memory.*

CLEAR TYPEAHEAD Empties type-ahead buffer.

CLEAR WINDOWS Clears all windows from screen and memory.*

CLOSE ALL/ALTERNATE/DATABASES/FORMAT/INDEX/PROCEDURE
Closes specified objects.

COMPILE <*file name*> [RUNTIME] Compiles a source file without executing it.*

CONTINUE Continues to search for next record using conditions specified by a LOCATE command.

CONVERT [TO <*expN*>] Adds a field to the current database file to monitor multiuser locking.*

COPY FILE <*file name*> TO <*file name*> Duplicates any type of file.

COPY INDEXES <*.ndx file list*> [TO <*.mdx file name*>] Copies multiple index files to a single .MDX file.*

COPY MEMO <*memo field name*> TO <*file name*> [ADDITIVE]
Exports data from a single memo field to the disk, appending a .TXT extension if not specified.*

COPY STRUCTURE TO <*file name*>
[FIELDS <*field list*>] [[WITH] PRODUCTION] Creates a dBASE IV database file structure from an existing file structure. The resulting database will contain only a header.

COPY TAG <*tag name*> [OF <*.mdx file name*>] TO <*.ndx file name*> Converts multiple index files (.MDX) to index (.NDX) files.*

COPY TO <*file name*> STRUCTURE EXTENDED Creates dBASE IV database file with five fields and one record. The fields define the field name, type, field lenth, decimal places, and index of the current database file. Use this command with CREATE FROM to create a new file from an existing structure.*

COPY TO <*file name*>
[[TYPE] <*file type*>]/[[WITH] PRODUCTION] [FIELDS <*field list*>]
[<*scope*>] [FOR <*condition*>] [WHILE <*condition*>] Copies database file in use to another file. †

COPY TO ARRAY <*array name*> [FIELDS <*field list*>]
[<*scope*>] [FOR <*condition*>] [WHILE <*condition*>] Copies fields from database to an array.*

COUNT [<*scope*>] [FOR <*condition*>] [WHILE <*condition*>]
[TO <*memvar*>] Tallies the number of records that meet the condition specified.

CREATE <*file name*> [FROM <*structure extended file name*>]
Defines a new dBASE IV database file.

CREATE APPLICATION <*file name*>/? Provides access to Applications Generator work surface.*

CREATE LABEL <*file name*>/? Creates a program to define a form for printing mailing labels. Creates a label object. Object is .LBL, program is .LBG.

CREATE QUERY <*file name*>/? Provides access to query design work surface. Creates query (.QBE) and update (.UPD) screens.

CREATE REPORT <*file name*>/? Provides access to report design work surface. Creates .FRM object and .FRG program.

CREATE SCREEN <*file name*>/? Provides access to form design work surface. Creates .SCR object and .FMT program.

CREATE VIEW <*file name*>/? Provides access to query design work surface. Creates query (.QBE) and update (.UPD) screens.

CREATE VIEW <*.vue file name*>/? FROM ENVIRONMENT Creates a view file from the current working environment.

DEACTIVATE MENU Deactivates current bar menu, removing it from the screen while keeping it in memory.*

DEACTIVATE POPUP Deactivates current pop-up menu, removing it from the screen while keeping it in memory.*

DEACTIVATE WINDOW <*window name list*> /ALL Deactivates specified windows from the screen without releasing them from memory.*

DEBUG <*file name*>/<*procedure name*> [WITH <*parameter list*>] Initiates program execution with debugger.*

DECLARE <*array name*$_1$>[{<*# rows*>}{,<*# columns*>}] [,<*array name*$_2$> [{<*# rows*>}{,<*# columns*>}]...] Declares an array. (Brackets are required here; curly braces indicate optional items.)*

DEFINE BAR <*line number*> OF <*popup name* > PROMPT <*expC*> [MESSAGE <*expC*>] [SKIP [FOR <*condition*>]] Defines a single option in a pop-up menu.*

DEFINE BOX FROM <*print column*> TO <*print column*> HEIGHT <*expN*> [AT LINE <*print line*>] [SINGLE/DOUBLE/<*border definition string*>] Defines a box for report text.*

DEFINE MENU <*menu name*> [MESSAGE <*expC*>] Defines a menu in memory that is later displayed with the ACTIVATE MENU command.*

DEFINE PAD <*pad name*> OF <*menu name*> PROMPT <*expC*> [AT <*row*>,<*col*>] [MESSAGE <*expC*>] Defines a pad in a bar menu.*

DEFINE POPUP <*popup name*> FROM <*row*$_1$>,<*col*$_1$> [TO <*row*$_2$>,<*col*$_2$>] [PROMPT FIELD <*field name*> /PROMPT FILES [LIKE <*skeleton*>]/PROMPT STRUCTURE] [MESSAGE <*expC*>] Defines a pop-up menu in memory that is later displayed with the ACTIVATE POPUP command.*

DEFINE WINDOW <*window name*> FROM <*row*$_1$>,<*col*$_1$> TO <*row*$_2$>,<*col*$_2$> [DOUBLE/PANEL/NONE/ <*border definition string*>] [COLOR [<*standard*>] [,<*enhanced*>] [,<*frame*>]] Defines a window in memory that is later displayed with the ACTIVATE WINDOW command.*

DELETE [<*scope*>] [FOR <*condition*>] [WHILE <*condition*>] Marks for deletion records in a database that meet the specified conditions.

DELETE TAG <*tag name*$_1$> [OF <*.mdx file name*>]/<*.ndx file name*$_1$>[,...] Deletes tags from .MDX files or closes .NDX files.*

DIRECTORY/DIR [[ON] <*drive*>:] [<*path*>] [[LIKE] [<*path*>] <*skeleton*>] Displays the names of the files that meet the specified drive, path, and template description.

DISPLAY [[FIELDS] <*expression list*>] [OFF] [<*scope*>] [FOR <*condition*>] [WHILE <*condition*>] [TO PRINTER/TO FILE <*file name*>] Displays fields of records or the results of expressions using fields. There is a pause between each screen display. If no scope is indicated, the scope defaults to the current record.

DISPLAY FILES [LIKE <*skeleton*>] [TO PRINTER/TO FILE <*file name*>] Displays all file names matching the specified skeleton. If no skeleton is specified, only database files are listed.

DISPLAY HISTORY [LAST <*expN*>] [TO PRINTER/TO FILE <*file name*>] Displays previously executed commands. Pauses between screen displays. †

DISPLAY MEMORY [TO PRINTER/TO FILE <*file name*>] Displays the current active memory variables. Pauses between screen displays. †

DISPLAY STATUS [TO PRINTER/TO FILE <*file name*>] Displays current status information about active databases, indexes, alternative files, and system parameters. Pauses between screen displays. †

DISPLAY STRUCTURE [IN <*alias*>] [TO PRINTER/TO FILE <*file name*>]
Displays the structure of an active database. Pauses between screen displays. †

DISPLAY USERS Identifies work stations logged in to dBASE IV. Pauses between screen displays.*

DO <*file name*>|<*procedure name*> [WITH <*parameter list*>]
Executes a program or procedure. †

DO CASE
  CASE <*condition*$_1$> <*commands*>
  [CASE <*condition*$_2$> <*commands*>]...
  [OTHERWISE <*commands*>]
ENDCASE Executes one of several alternate program paths.

DO WHILE <*condition*>
  <*commands*>
  [LOOP] [EXIT]
ENDDO Executes a loop within the program until a specified condition is met.

EDIT [<*scope*>] [NOINIT] [NOFOLLOW] [NOAPPEND] [NOMENU]
  [NOCLEAR] [NODELETE]
  [NOEDIT] [FIELDS <*field list*>] [<*record number*>]
  [FOR <*condition*>] [WHILE <*condition*>] Edits the contents of a database record. †

EJECT [PAGE] Executes a form feed on the printer. The PAGE option supports the ON PAGE command. †

ERASE <*file name*>|? Deletes the specified file from the directory.

EXIT Escapes from a DO WHILE loop without regard to the specified condition.

EXPORT TO <*file name*> [TYPE] PFS/DBASEII/FW2/RPD
  [FIELD <*field list*>] [<*scope*>]
  [FOR <*condition*>] [WHILE <*condition*>] Exports data to an external file format. †

FIND <*key value*> Positions record pointer in an indexed database to the first record that meets the specified condition.

FUNCTION <*procedure name*> Identifies the start of a user-defined function.*

GO/GOTO [BOTTOM/TOP/[RECORD] <*record number*>]
  [IN <*alias*>] Positions record pointer to a specified record in a database. †

HELP [<*keyword*>] Switches to dBASE IV's menu-driven help mode.

IF <*condition*>
  <*commands*>
[ELSE <*commands*>]
ENDIF Permits execution of the specified commands if a particular condition is met.

IMPORT FROM <*file name*> [TYPE] PFS/DBASEII/FW2/RPD/WK1 Creates new dBASE IV files from external files. Destroys any existing files with the same name. †

INDEX ON <*key expression*> TO <*.ndx file name*>
  /TAG <*tag name*> [OF <*.mdx file name*>] [UNIQUE]
  [DESCENDING] [FOR<*condition*>] Creates an index for a database on a specified key. †

INPUT [<*prompt*>] TO <*memvar*> Permits entry of data to a specified memory variable.

INSERT [BLANK] [BEFORE] Inserts a new record into the specified position in the database.

JOIN WITH <*alias*> TO <*file name*>
  FOR <*condition*> [FIELDS <*field list*>] Combines records and fields from two databases to a third database.

KEYBOARD <*expC*> [CLEAR] Enters a series of characters into keyboard type-ahead buffer as if entered from the keyboard.

LABEL FORM <*file name*>/? [<*scope*>] [SAMPLE] [TO PRINT]
[FOR <*condition*>] [WHILE <*condition*>]
[TO PRINTER/TO FILE <*file name*>] Prints labels using the specified label form file. †

LIST [[FIELDS] <*expression list*>] [OFF]
[<*scope*>] [FOR <*condition*>] [WHILE <*condition*>]
[TO PRINTER/TO FILE <*file name*>] Lists fields of records or the results of expressions using fields. There are no pauses between screen displays. If no scope is indicated, the scope defaults to all records in the database. †

LIST FILES [LIKE <*skeleton*> [TO PRINTER/TO FILE <*file name*>] Lists all file names matching the specified skeleton. If no skeleton is specified, only database files are listed. †

LIST HISTORY [LAST <*expN*>] [TO PRINTER/TO FILE <*file name*>]
Displays previously executed commands. †

LIST MEMORY [TO PRINTER/TO FILE <*file name*>] Displays the current active memory variables. †

LIST STATUS [TO PRINTER/TO FILE <*file name*>] Displays current status information about active databases, indexes, alternative files, and system parameters. †

LIST STRUCTURE [IN <*alias*>] [TO PRINTER/TO FILE <*file name*>]
Displays the structure of an active database. †

LIST USERS Identifies work stations logged in to dBASE IV.*

LOAD <*binary file name*> Places binary file in memory to execute from the CALL command.

LOCATE [FOR<*condition*>] [<*scope*>] [WHILE <*condition*>]
Positions record pointer to first record in a database meeting the specified condition. Uses unindexed files, scanning the file until the match is found.

LOGOUT Forces a logout, and either returns to the dot prompt or displays a screen for a login.*

LOOP Skips all commands from the current command to the end of the DO loop.

MODIFY APPLICATION <*file name*>/? Provides access to the Applications Generator work surface.*

MODIFY COMMAND/FILE <*file name*> [WINDOW <*window name*>] Modifies a program using dBASE IV's internal editor or, if specified in the CONFIG.DB file, an external word processor. †

MODIFY LABEL <*file name*>/? Modifies a label object.

MODIFY QUERY <*file name*>/? Provides access to query design work surface. Modifies query (.QBE) and update (.UPD) screens.

MODIFY REPORT <*file name*>/? Provides access to report design work surface. Modifies .FRM object and .FRG program.

MODIFY SCREEN <*file name*>/? Provides access to form design work surface. Modifies .SCR object and .FMT program.

MODIFY STRUCTURE Modifies the structure of a database, if possible without losing the data within the database.

MODIFY VIEW <*file name*>/? Provides access to query design work surface. Modifies query (.QBE) and update (.UPD) screens.

MOVE WINDOW <*window name*> TO <*row*>,<*column*>/
BY <*delta row*>,<*delta column*> Move a window to a new screen location.*

NOTE/*/&& [<*text*>] Indicates nonexecuting comment. NOTE and * are used to identify a full comment line. && is used to add a comment to an existing command line.

ON ERROR [<*command*>] Executes specified dBASE command when error occurs. If no argument is specified, dBASE IV uses its internal error recovery procedure.

ON ESCAPE [<*command*>] Executes the specified procedure when the Esc key is pressed.

ON KEY [LABEL <*key label name*>] [<*command*>] Executes the specified command when the specified key is pressed. Useful for displaying help screens when F1 is pressed. †

ON PAD <*pad name*> OF <*menu name*> [ACTIVATE POPUP <*popup name*>] Activates the menu specified by ON PAD whenever cursor is placed on pad of *menu name*.*

ON PAGE [AT LINE <*expN*> <*command*>] Specifies action when ?/?? output reaches specified line. Primary use is for creating footers or headers.*

ON READERROR [<*command*>] Activates a command program on input error using @...GET command.*

ON SELECTION PAD <*pad name*> OF <*menu name*> [<*command*>]
Specifies action for a bar menu pad.*

ON SELECTION POPUP <*popup name*>/ALL [<*command*>]
Specifies action, program, or procedure to be activated by a pop-up menu selection.*

PACK Removes all records in the database that are marked for deletion. If the database is open with an index, the index will be updated. This will change the record numbers in the database of all records that follow any records marked for deletion.

PARAMETERS <*parameter list*> Specifies memory variables to be used with the DO...WITH command. If parameters are passed to a procedure, this must be the first executable command in the procedure.

PLAY MACRO <*macro name*> Plays a series of keystrokes that have been previously recorded. Macros are recorded in last-in, first-out order (LIFO). The last keystroke entered will be the first played.*

PRINTJOB
<*commands*>
ENDPRINTJOB Specifies structured programming commands that can be used to control a print job. *

PRIVATE ALL [LIKE/EXCEPT <*skeleton*>] Specifies memory variables to be considered local to the program and not passed to higher-level programs.

PROCEDURE <*procedure name*> Identifies an executable procedure in the program file or open procedure file.

PROTECT Activates security system for either single-user or multiuser environment.*

PUBLIC <*memvarlist*>/[ARRAY <*array list*>] Defines variables to be available to all programs at any level (global variables). †

QUIT Closes all files and exits from dBASE IV.

READ [SAVE] Permits user to enter data to all pending GETs.

RECALL [<*scope*>] [FOR <*condition*>] [WHILE <*condition*>]
Unmarks specified records that were previously marked for deletion (cannot be used to recover if the database has been packed since the records were marked).

REINDEX Rebuilds all .NDX and .MDX indexes in the current work area. †

RELEASE ALL [LIKE/EXCEPT <*template*>]
/ [<*memvarlist*>]
/MODULE <*name*>
/MENUS [<*menu name list*>]
/POPUPS [<*popup name list*>]
/WINDOWS [<*window name list*>] Erases specified objects, removing them from memory. †

RELEASE SCREEN Releases screen saved with SAVE SCREEN and recovers memory space.

RENAME <*old file name*> TO <*new file name*> Changes the name of any file.

REPLACE [<*scope*>] <*field*$_1$> WITH <*exp*$_1$>
[ADDITIVE] [<*field*$_2$> WITH <*exp*$_2$>...]
[FOR <*condition*>] [WHILE <*condition*>] Changes the contents of the specified database fields. †

REPLACE FROM ARRAY <*array name*> [<scope>] [FIELDS <*field list*>] [FOR <condition>] [WHILE <*condition*>] Replaces fields in database with data from an array.*

REPORT FORM <*file name*>/? [<*scope*>] [FOR <*condition*>]
[WHILE <*condition*>] [PLAIN] [HEADING <*expC*>]
[NOEJECT] [SUMMARY] [TO PRINTER/TO FILE <*file name*>] Creates a report using a previously created report form from the current database. †

RESET [IN <*alias*>] Resets integrity tag set by the BEGIN TRANSACTION command. This command cannot be used in a program.*

RESTORE FROM <*file name*> [ADDITIVE] Retrieves saved memory variables (see SAVE TO).

RESTORE MACROS FROM <*macro file*> Restores macros saved to the current macro library.*

RESTORE SCREEN FROM <*screen name*> Restores screen previously saved.

RESTORE WINDOW <*window name list*>/ALL FROM <*file name*> Restores window definitions from a disk file to memory.*

RESUME Resumes a suspended program.

RETRY Reexecutes a command procedure that caused an error.

RETURN [<*exp*> /TO MASTER/TO <*procedure name*>] Returns to the specified or calling program.

ROLLBACK [<*database file name*>] Undoes any changes made since the last BEGIN TRANSACTION command.

RUN/! <*DOS command*> Runs a program external to dBASE IV.

SAVE TO <*file name*> [ALL LIKE/EXCEPT <*skeleton*>] Saves current memory variables to a memory file.

SAVE MACROS TO <*macro file*> Saves current macros to a disk file.*

SAVE SCREEN TO <*screen name*> Saves screen to memory variable.*

SAVE WINDOW <*window name list*>/ALL TO <*file name*> Saves window definitions to a disk file. *

SCAN [<*scope*>] [FOR <*condition*>] [WHILE <*condition*>]
[<*commands*>]
[LOOP]
[EXIT]
ENDSCAN Permits scanning on all records of a file or while a condition is true (similar to a DO WHILE...ENDDO).*

SEEK <*exp*> Positions the record pointer to the first record in an indexed file with a key value matching the specified expression.

SELECT <*work area*>/<*alias*> Switches dBASE IV between active work areas. You can define up to 10 work areas with the USE command.

SET [<*switch*> <*status*>] Menu-driven command to set environment switches. See SET commands in Appendix F. †

SHOW MENU <*menu name*> [PAD <*pad name*>] Displays a bar menu without activating it. Used in program development to check a menu without activating it.*

SHOW POPUP <*popup menu*> Displays a pop-up menu without activating it. Used in program development to check a menu without activating it.*

SKIP [<*expN*>] [IN <*alias*>] Moves the current record pointer backward or forward by the amount specified in the expression.

SORT TO <*file name*> ON <*field*$_1$> [/A] [/C] [/D] [<*scope*>]
[,<*field*$_2$> [/A] [/C] [/D]...] [ASCENDING/DESCENDING]
[FOR <*condition*>] [WHILE <*condition*>] Creates a database with records sorted in the specified sort order. Default is ascending order. †

STORE <*exp*> TO <*memvarlist*>/<*array element list*> Stores an expression or value to a memory variable or array.

SUM [<*scope*>] [<*expN list*>]
[TO <*memvarlist*>/TO ARRAY <*array name*>]
[FOR <*condition*>] [WHILE <*condition*>] Computes the sum of an expression. †

SUSPEND Halts execution of a command file for debugging.

TEXT
  <text>
ENDTEXT Displays or prints a block of text from a command file.

TOTAL ON *<keyfield>* TO *<file name>* [*<scope>*]
  [FIELDS *<field list>*]
  [FOR *<condition>*] [WHILE *<condition>*] Creates a summary database of a presorted file containing totals.

TYPE *<file name>* [TO PRINTER/TO FILE *<file name>*] [NUMBER]
Displays or prints an ASCII file. (Note: the TYPE() function, unlike the TYPE command, is a function and is used to test an expression.) †

UNLOCK [ALL/IN *<alias>*] Releases record and file locks in a network environment. †

UPDATE ON *<keyfield>* FROM *<alias>* REPLACE *<field$_1$>* WITH
  <exp$_1$> [,*<field$_2$>* WITH *<exp$_2$>*...] [RANDOM] Allows modification of a file from a batch file.

USE [*<database file name>*/?] [IN *<work area number>*]
  [INDEX *<.ndx or .mdx file list>*]
  [ORDER *<.ndx file name>*
  /[TAG] *<.mdx tag>*[OF *<.mdx file name>*]]
  [ALIAS *<alias>*] [EXCLUSIVE] [NOUPDATE] Opens the specified database in the specified work area. †

WAIT [*<prompt>*] [TO *<memvar>*] Suspends program execution until any key is pressed.

ZAP Removes all records from a database.

SET ALTERNATE on/OFF Switches output to alternate file on or off.

SET ALTERNATE TO <*file name*> [ADDITIVE] Defines an alternate file that can be used to save output. †

SET AUTOSAVE OFF/on Forces an automatic write to disk when buffer is changed.*

SET BELL ON/off Turns on the bell during data entry when the data field has been filled.*

SET BELL TO [<*frequency*>,<*duration*>] Sets tone and duration of bell.*

SET BLOCKSIZE TO <*expN*> Sets blocksize of .MDX and memo files. *Blocksize* defines the number of bytes in each block transferred for .MDX and memo field files. The actual block size is determined by multiplying *expN* by 512. The default value of 1 is 512 bytes, the same as in dBASE III.*

SET BORDER TO [SINGLE/DOUBLE/PANEL/NONE/<*border definition string*>] Defines borders for windows, boxes, pop-ups, menus, and lines.*

SET CARRY on/OFF Copies data from a prior record to a new record when APPEND or INSERT is used.

SET CARRY TO [<*field list*> [ADDITIVE]] Selects fields to carry forward. †

SET CATALOG on/OFF Adds file opened to catalog file.

SET CATALOG TO [<*file name*>/?] Creates a catalog file and opens it in work area 10.

SET CENTURY on/OFF Controls the display of the century in data displays.

SET CLOCK on/OFF Sets clock in display.*

SET CLOCK TO [<*row*>,<*column*>] Defines position to display clock.*

SET COLOR OF NORMAL/MESSAGES/TITLES/BOX/HIGHLIGHT /INFORMATION/FIELDS TO [<*color attribute*>] Defines color and display attributes of display items.*

SET COLOR ON/OFF Toggles color display on color or monochrome monitor. (Default depends on monitor being used.)

SET COLOR TO [<*standard*>], [<*enhanced*>][<*perimeter*>][<*background*>] Sets specified area of screen or objects to specified color.

SET CONFIRM on/OFF Requires carriage return on data entry.

SET CONSOLE ON/off Sends or suspends output to the screen.

SET CURRENCY LEFT/right Defines position of currency symbol.*

SET CURRENCY TO [<*expC*>] Defines currency symbol.*

SET CURSOR ON/OFF Controls display of cursor.*

SET DATE [TO] AMERICAN/ansi/british/french/german/italian /japan/usa/mdy/dmy/ymd Determines format of date display. †

SET DBTRAP ON/OFF Improves functionality of UDFs if set off, but also enables you to crash dBASE if you are not careful.*

SET DEBUG on/OFF Controls whether the SET ECHO ON output is sent to the printer.

SET DECIMALS TO <*expN*> Sets the number of decimal places displayed after a calculation.

SET DEFAULT TO <*drive*>[:] Defines the default drive for all programs and databases, and for all file accesses. Does not set path.

SET DELETED on/OFF Determines whether commands see records marked for deletion.

SET DELIMITERS on/OFF Selects optional delimiter.

SET DELIMITERS TO <*expC*>/[DEFAULT] Specifies delimiters to use for displays.

SET DESIGN ON/off Blocks transfer to design mode.*

SET DEVELOPMENT ON/off Blocks the use of an outdated .DBO file when editing programs.*

SET DEVICE TO printer/SCREEN/file <*file name*> Determines if formatted output will be sent to screen or printer. †

SET DIRECTORY TO [<*drive*:>] [<*path*>] Sets working path.

SET DISPLAY TO MONO/COLOR/EGA25/EGA43/MONO43 Selects monitor mode. (Default depends on current monitor.)*

SET ECHO on/OFF Determines if command lines are echoed on the output device.

SET ENCRYPTION on/OFF Determines whether new database files are encrypted.

SET ESCAPE ON/off Controls whether Esc key can abort a command file execution.

SET EXACT on/OFF Controls whether exact matching is necessary on indexed retrieval.

SET EXCLUSIVE on/OFF Permits user to open a database file on a multiuser system for exclusive use.*

SET FIELDS on/OFF Accepts or ignores fields set with SET FIELDS TO.

SET FIELDS TO [<*field*$_1$>[/R]/<*calculated field id*$_1$>..]
[,<*field*$_2$>[/R]/<*calculated field id*$_2$>..] Specifies fields for access. In dBASE IV, enhanced to support calculated fields. †

SET FIELDS TO ALL [LIKE/EXCEPT <*skeleton*>] Specifies fields for access.

SET FILTER TO [<*condition*>] [<*file name*>/?] Controls filter or template used as a condition by other commands in retrievals.

SET FORMAT TO [<*file name*>/?] Opens a format file used for formatted input.

SET FULLPATH on/OFF Determines whether functions that return the file name return the full path name.*

SET FUNCTION <*expN*> /<*expC*>/<*key label*> TO <*expC*> Sets the function-key actions. †

SET HEADING ON/off Controls the display of the heading with LIST and DISPLAY commands.

SET HELP ON/off Determines whether the pop-up help window is displayed in response to an error.

SET HISTORY ON/off Controls line editing at dot prompt from history file.

SET HISTORY TO <*expN*> Specifies number of lines to save in history. Default is 20.

SET HOURS TO [12/24] Sets display to 12- or 24-hour clock. Default is 12.*

SET INDEX TO [<*file name list*>/? [ORDER
<*.ndx file name*>/[TAG] <*.mdx tag name*>
[OF <*.mdx file name*>]]] Opens the specified index file(s). Enhanced to support .MDX files. †

SET INSTRUCT ON/off Enables display of prompts.*

SET INTENSITY ON/off Sets the display of the reverse-video attribute.

SET LOCK ON/off Specifies if records are locked in multiuser environment.*

SET MARGIN TO <*expN*> Defines the left margin for the printer.

SET MARK TO [<*expC*>] Changes date delimiter.*

SET MEMOWIDTH TO <*expN*> Defines width of memo field output. Default is 50.

SET MENU ON/off (Controls the display of the function-key prompt menu. This command is not used in dBASE IV and is ignored.)

SET MESSAGE TO [<*expC*> [AT <*expN*>[, <*expN*>]]] Displays specified message at specified position. Default is bottom of the screen. SET STATUS must be ON.

SET NEAR on/OFF When a FIND or SEEK fails, this command positions the record pointer after nearest record instead of at end of file.*

SET ODOMETER TO <*expN*> Defines the update interval of record counter for record count displays. The default is 1, and the maximum is 200.

SET ORDER TO [<*expN*>]/[TAG] <*file name*> / <*.mdx tag name*> [OF <*.mdx file name*>] Specifies which of several open index files or tags are used to control FINDs and SEEKs. Enhanced to support .MDX files.

SET PATH TO [<*pathlist*>] Specifies a path for file searches.

SET PAUSE on/OFF Controls the display of the SQL SELECT command and pause of data on screenfuls.

SET POINT TO [<*expC*>] Controls the character used for the decimal point.*

SET PRECISION TO [<*expN*>] Defines digits for dBASE IV to use internally on type N numbers (10–20).*

SET PRINTER [on/OFF] [TO <*DOS device*>] [TO FILE <*file name*>] Directs unformatted output to the printer. Additional options are available in a network environment. Supports NUL, COM3, and COM4 now.*

SET PROCEDURE TO [<*procedure file name*>] Opens the specified procedure file.

SET REFRESH TO <*expN*> Specifies time interval between file checks in networks for screen refreshes. Default is 0. Requires file to have been processed with the CONVERT command.*

SET RELATION TO [<*expression*>] INTO <*alias*> [,<*expression*> INTO <*alias*>]...]] Links two databases on the key expression. dBASE IV supports multiple links. Enhanced to support multiple relations. †

SET REPROCESS TO <*expN*> Specifies number of times dBASE IV tries a network file before producing an error message. Default is zero.*

SET SAFETY ON/off Sets a level of protection on overwriting files.

SET SCOREBOARD ON/off Suppresses the display of certain status information, permitting the user to use line zero on the display.

SET SEPARATOR TO [<*expC*>] Changes separator symbol from default comma.*

SET SKIP TO [<*alias*>[,<*alias*>]...] Supports multiple detail record handling in linked files.*

SET SPACE ON/off Permits ? and ?? commands to print a space between printed expressions.*

SET SQL on/OFF Activates SQL mode.*

SET STATUS ON/off Controls display of status line at bottom of screen.

SET STEP on/OFF Halts execution of a program after each instruction.

SET TALK ON/off Displays the result of each command after execution.

SET TITLE ON/off Controls prompt for descriptive title when a file is added to the catalog with SET CATALOG ON.

SET TRAP on/OFF Activates debugger on an error.*

SET TYPEAHEAD TO <*expN*> Controls size to type-ahead buffer. Default is 20.

SET UNIQUE on/OFF Controls whether all records with a given key appear in the index file.

SET VIEW TO <*file name*>/? Opens a view file. †

SET WINDOW OF MEMO TO <*window name*> Activates a previously defined window for a memo field.*

Apêndice H - Parãmetros para o comando SET COLOR

COLOR SCREENS	
Black low	<space>
Black high	<space>+
Blue low	B
Blue high	B+
Gray	N+
Green low	G
Green high	G+
Cyan low	BG
Cyan high	BG+
Red low	R
Red high	R+
Magenta low	RB
Magenta high	RB+
Brown	GR
Yellow	GR+
White low	W
White high	W+

Note: Adding an asterisk to color or monochrome codes produces blinking or flashing.

MONOCHROME SCREENS	
Normal low	W
Normal high	W+
Underline low	U
Underline high	U+
Reverse low	/W
Reverse high	+/W
Invisible	<space>
Blinking low	W*
Blinking high	W+*
Blinking underline low	U*
Blinking underline high	U+*
Blinking reverse low	*/W
Blinking reverse high	+*/W

Note: Underline codes don't work on all systen