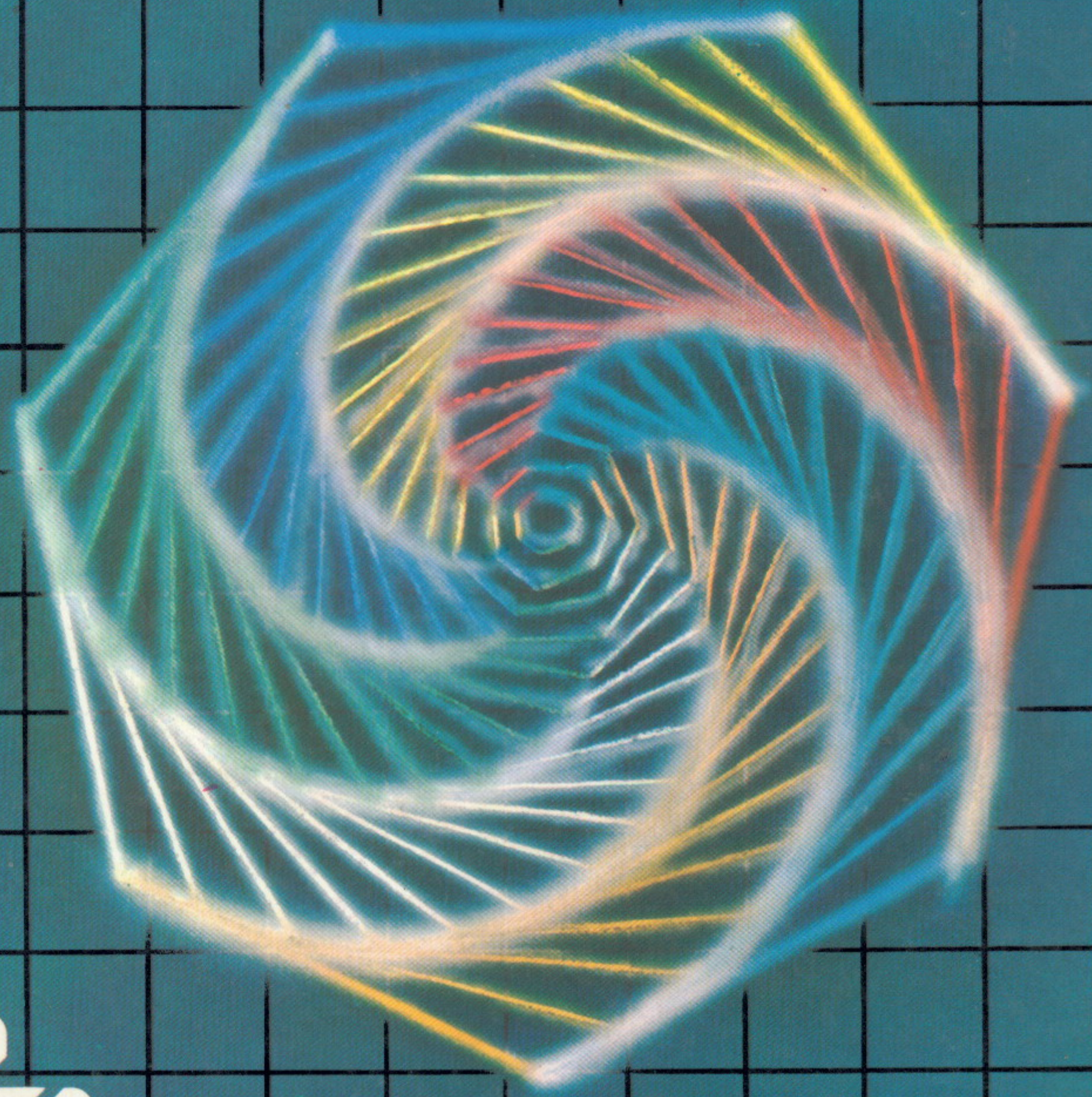
PROGRAMAÇÃO AVANÇADA

ROBIN JONES
IAN STEWART

ZX Spectrum

PARA O

e TK 90X DA MICRODIGITAL



Go to Informática

EDIÇÕES CETOP

COLECÇÃO GO TO INFORMÁTICA

PROGRAMAÇÃO AVANÇADA PARA O ZX SPECTRUM E TK90X DA MICRODIGITAL

Volumes publicados nesta colecção:

1 — *Introdução à Programação em Pascal — Exercícios*
2 — *13 Jogos para o Spectrum 16 K ou 48 K*
3 — *Jogos de Paciências e* Puzzles *para o Spectrum e XZ81*
4 — *24 Jogos para o QL*
5 — *Inteligência Artificial: ZX Spectrum*
6 — *Programação Avançada para o ZX Spectrum e TK90X da Microdigital*

Ian Stewart e Robin Jones

PROGRAMAÇÃO AVANÇADA PARA O ZX SPECTRUM E TK90X DA MICRODIGITAL

EDIÇÕES CETOP ®

APARTADO 33 ▪ 2726 MEM MARTINS CODEX
SEDE: ESTRADA LISBOA-SINTRA, km 14 ▪ TELEF. 926 32 22 ▪ TELEGS. EDICETOP

Este livro foi traduzido da edição original com o título:
Further Programming for the ZX SPECTRUM

Capa: estúdios P. E. A.

Tradução: Maria Carolina Freire

Edição n.º 0619006/081

Editor: Tito Lyon de Castro

Execução técnica:
Gráfica Europam, Lda.
Mem Martins

ÍNDICE

PREFÁCIO

O nosso leitor é possuidor de um ZX Spectrum que maneja com bastante à-vontade. Sabe utilizar as teclas e é capaz de enfileirar vinte ou trinta linhas de BASIC de modo a funcionarem. A sua iniciação foi feita através do Manual *e de um livro de introdução. Já dactilografou dúzias de programas a partir das revistas da especialidade e chegou à conclusão de que os mais curtos fazem todos a mesma coisa e os maiores, quando não estão cheios de erros, levam horas e horas de trabalho cuidadoso... enquanto, por uns escassos 300$00, é possível comprar uma* cassette *que apresente resultados mais espectaculares. Esta solução seria óptima se o leitor estivesse disposto a continuar a ir gastando 300$00 para comprar* software *feito por outras pessoas; porém, o leitor quer introduzir o seu próprio* software.

Assim sendo, que fazer?

Ainda lhe falta muito até ser capaz de escrever jogos em Linguagem Máquina com a qualidade dos jogos dos centros comerciais, ou de produzir programas que apresentem o céu nocturno em qualquer data entre 4000 a.C. e 6000 d.C. Este livro pode iniciá-lo nessa via, mas não vai certamente levá-lo até lá. Seja como for, vai ajudá-lo a desenvolver as suas capacidades e as do aparelho.

Existem três direcções principais a explorar.

A primeira poderia ser descrita como «Teoria da Computação»: trata-se da forma de desenvolver técnicas para melhorar os seus programas. Pelo que diz respeito a este *livro, apresentamos uma perspectiva muito prática da teoria: concentramo-nos em características específicas do Spectrum, tais como as suas facilidades de cor e as suas imagens, e penetramos um pouco mais no funcionamento do aparelho. O leitor vai aprender mais coisas acerca de caracteres de controlo, funções definidas pelo utilizador, imagens definidas pelo utilizador, variáveis do sistema, ficheiros de apresentação e de atributos e sobre as formas de tirar partido deles.*

A segunda é a «Valorização do Aparelho». Escrevendo programas utilitários adequados, o leitor pode equipar o seu Spectrum com facilidades que o aparelho por si só não possui: a renumeração rápida das linhas de programas em BASIC (a nossa rotina permite a escolha de um determinado bloco de um programa e a renumeração apenas desse bloco, o que é óptimo para ordenar sub-rotinas); a marcação de ima-

gens sem ter de se preocupar com os pontos que ficam fora do visor; o preenchimento automático de desenhos delineados; um sistema eficaz para manusear grandes quantidades de dados contidos em cassette *como se se tratasse de ficheiros, que poderia servir de base a um sistema de armazenamento de registos prático para ser usado em negócios ou em casa: através de etapas simples, o leitor passa de um mero «Sistema de Ficheiros em* Cassette» *a um «Sistema de Gestão de Dados».*

Quanto à terceira... Bem, o leitor já reparou que quando se pergunta a um entusiasta dos computadores «Tudo isso é muito bonito, mas, afinal de contas, para que é que serve?», ele tem tendência a mudar de assunto? É como se o objectivo principal da computação fosse gerar mais *computação: a arte pelo amor da arte, os computadores como um modo de estar no mundo. Contudo, não acham que seria agradável* utilizar *o computador para fazer de facto* qualquer coisa? *Neste livro vai encontrar várias sugestões de utilização: mapas, cartas celestes, experiências psicológicas, estatística simples, criptografia e criptanálise, manipulação de símbolos.*

Duas áreas que não vão ser focadas neste livro são a Linguagem Máquina e a teoria «pura» (assuntos como estruturas de dados e programação estruturada). Estes temas são tratados noutros livros: Machine Code and Better Basic *e* Spectrum Machine Code.

O nosso objectivo principal não é a apresentação de sofisticados programas já prontos. Preferimos enfatuar o penoso mas recompensador processo de transformação de uma ideia inicial num programa que funcione. *Por vezes, em vez de apresentarmos apenas o resultado final, descrevemos rotinas que são subsequentemente modificadas, reescritas, revistas, ou inteiramente retiradas e substituídas. Afinal de contas, qualquer programa não trivial é escrito desta forma, e se disséssemos o contrário estaríamos a enganar o leitor. Não pretendemos dar-lhe a impressão de que escrever programas é fácil e pouco trabalhoso: o que é importante é que se lembre de que* toda a gente *comete erros, por isso não há razão para desanimar quando também* os cometer; *trate é de os encontrar e de os corrigir. Como é evidente, quaisquer métodos que ajudem a diminuir as probabilidades de cometer erros são bem vindos.*

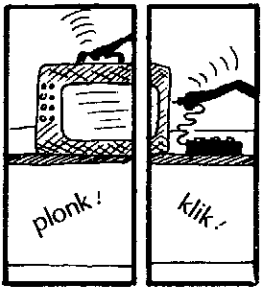
No entanto, algumas das rotinas de aplicação geral são adicionalmente apresentadas em apêndices, separadamente, de forma que não seja necessário examinar em pormenor as descrições das suas construções para ser capaz de as utilizar. Se quiser limitar-se a copiar a listagem e RUN o programa, pode fazê-lo.

Tal como acontece em todos os nossos livros, tentamos manter as explicações claras e simples. Este livro não é um curso com uma estrutura rígida: foi planeado de forma que o leitor pudesse começar em qualquer parte. Alguns dos capítulos são até certo ponto apoiados por capítulos anteriores, mas, quando isto acontece, é sempre perfeitamen-

te claro. Por isto, comece por folhear o livro para ver quais os assuntos que têm mais interesse para os seus gostos pessoais e dedique-se a esses em primeiro lugar.

Vai achá-los muito instrutivos e, além disso, divertidos.

Até este ponto, temo-nos referido a nós próprios como «nós». Porém, tal como já tinha acontecido no nosso livro *Easy Programming*, chegámos à conclusão de que, em certos capítulos, isso não dava resultado. Por essa razão, daqui em diante, vamos referir-nos a nós próprios no singular, como «eu». Sempre que escrevermos «nós», isso quererá dizer «eu e o leitor». Pode parecer uma ideia disparatada, mas o facto é que este processo é mais didáctico.



Quanto mais trabalho estiver preparado para introduzir no seu Spectrum, mais trabalho pode ele ser treinado para realizar. Se perder um par de horas, pode obter um...

1

MAPA-MÚNDI

Eis uma das possibilidades. A mesma técnica permitir-lhe-á desenhar e arquivar (SAVE) figuras lineares de alta resolução, como por exemplo retratos de Isaac Newton ou de Olivia Newton-John, ou paisagens de Marte para serem utilizadas em programas de *Space Inveiglers*.

É uma tarefa fácil, mas demorada. Apresentamos-lhe aqui a reprodução de um mapa-múndi obtido num Spectrum, para que fique convencido de que os resultados obtidos justificam amplamente o tempo gasto.

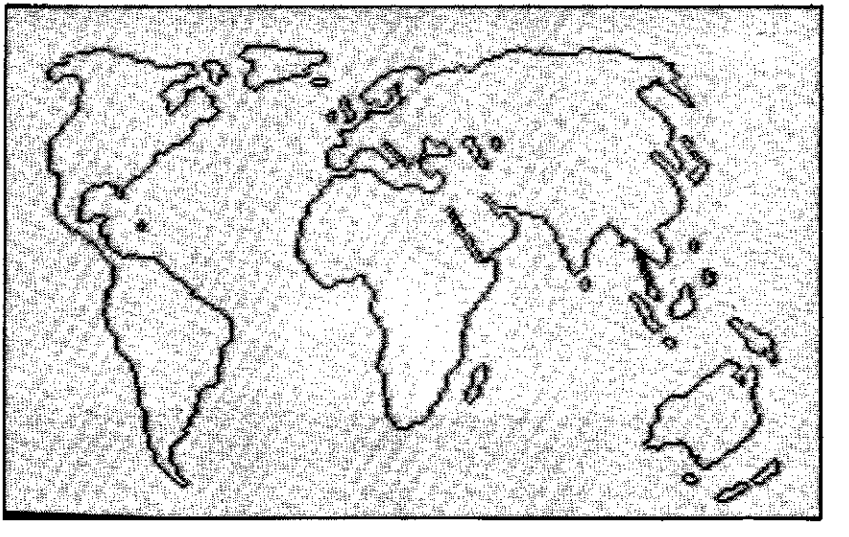


Fig. 1.1 — *Contorno de um mapa-múndi, obtido num Spectrum.*

A forma mais trabalhosa de introduzir no aparelho os dados necessários é debruçar-se longamente sobre quadrículas de latitude e longitude, copiando as coordenadas e introduzindo-as numa rotina de desenho.

A forma que utilizei de facto para o fazer é pouco elaborada, mas eficaz.

1. Corte um pedaço de plástico transparente — um bocado de um saco de politeno, por exemplo — com a medida do visor do seu monitor de TV.
2. Sobre o pedaço de plástico marque o contorno da área central de apresentação do Spectrum, ou seja, da área onde é possível PLOT ou PRINT. A maneira mais fácil de descobrir este contorno é dactilografar no aparelho BORDER 0.
3. Arrange um mapa-múndi de tamanho adequado para caber nesse contorno.
4. Desenhe-o no politeno com uma caneta de ponta de feltro (é suficiente uma imagem pouco elaborada).
5. Deixe a tinta secar bem, e, tendo cuidado para não apagar nada, cole o mapa sobre o visor com fita adesiva, fazendo-o coincidir com a área central.

É este o *hardware* necessário para este método.
Vamos agora tratar do *software*...

SKETCHPAD

6. Dactilografe no seu aparelho um programa Sketchpad que lhe permita controlar um *pixel* móvel sobre o visor a partir do teclado, de forma que ele possa tanto marcar uma posição (PLOT) como mover-se. Também vale a pena possuir uma forma de apagar os erros. Eis um programa que serve estes propósitos e que o leitor pode, é claro, tornar mais sofisticado, se assim desejar.

```
10  LET x = 0: LET y = 175
20  OVER 1
30  LET flag = 0
40  INPUT d$
50  LET x0 = x: LET y0 = y
60  IF CODE d$ < 60 THEN GO TO 100
70  IF d$ = "m" THEN LET flag = 0
```

```
80  IF d$ = "p" THEN LET flag = 1
90  GO TO 40
100 REM Keyboard response
110 GO SUB 10 * CODE d$ - 290
120 IF flag = 0 THEN PLOT x0, y0
130 PLOT x, y: GO TO 40
200 LET x = x + 1: LET y = y + 1: RETURN
210 LET x = x + 1: LET y = y - 1: RETURN
220 LET x = x - 1: LET y = y - 1: RETURN
230 LET x = x - 1: LET y = y + 1: RETURN
240 LET x = x - 1: RETURN
250 LET y = y - 1: RETURN
260 LET y = y + 1: RETURN
270 LET x = x + 1: RETURN
```

7. Utilizando os controlos do teclado (como é pormenorizadamente explicado mais abaixo), desloque o *pixel* para um ponto coincidente com as linhas traçadas no plástico e vá-o movendo ao longo dessas linhas, marcando as posições à medida que vai andando, desenhando assim o contorno dos continentes. Quando tiver dado a volta a um continente, mude para outro e recomece a marcar as posições.

Como vê, é de facto fácil, e os resultados podem ser magníficos se dedicar ao trabalho tempo e atenção.

A UTILIZAÇÃO DE SKETCHPAD

O programa pode estar num de dois «modos»:

m: mover (MOVE) o *pixel* para uma nova posição;
p: marcar a posição actual (PLOT) quando o *pixel* é deslocado.

O programa começa em «m». É possível alterar os modos em qualquer altura, dactilografando no aparelho os símbolos «p» ou «m».

Os movimentos do *pixel* são controlados pelas teclas 1-8, de acordo com a tabela seguinte:

4	7	1
5	*	8
3	6	2

De acordo com os números, o *pixel* desloca-se da sua posição actual (*) para uma casa acima, abaixo, ao lado ou na diagonal. (A ordem escolhida pode parecer estranha, mas seguiu o seguinte critério: as teclas das «setas», 5-8, funcionam da maneira habitual, e as deslocações «diagonais» começam em 1 e seguem o sentido dos ponteiros do relógio.)

Tal como está, o programa exige que o utilizador ENTER cada número ou cada símbolo de modo. Se preferir, pode usar INKEY$; porém, o processo anterior permite-lhe verificar se carregou na tecla correcta antes de ter estragado alguma coisa.

Faça um ensaio com as deslocações. Devido à instrução OVER 1, se PLOT duas vezes no mesmo local, o *pixel* desaparece. Isto permite apagar os erros. Contudo, o leitor deve lembrar-se de duas coisas:

1. Para deslocar o *pixel* para uma nova região, mova-o uma vez *para fora* da sua curva já completa (numa direcção tal que não se lhe sobreponha) *antes* de mudar para modo «m».
2. Ao chegar ao novo pedaço de curva, não carregue em «p» antes de o seu *pixel* estar exactamente alinhado com ela.

Se se enganar ao escolher os modos, tenderão a aparecer no visor *pixels* isolados. Para se livrar deles, coloque o programa em modo «m»; desloque o seu *pixel* até os alcançar (e se lhes sobrepor); *mude para modo* «p»; desloque-o um espaço; volte para modo «m». Experimente, e vai ver que dá resultado.

Não espere obter em dez minutos um resultado brilhante.

ARQUIVE O SEU MAPA

Quando o resultado o satisfizer, guarde (SAVE) o mapa em fita: nunca mais vai precisar de passar duas horas colado ao visor para o

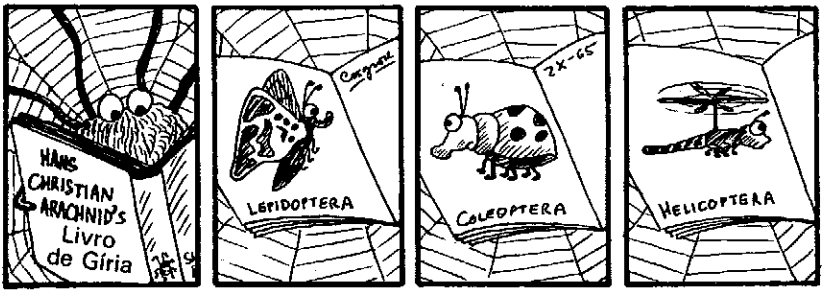
obter. Basta parar (STOP) o programa Sketchpad e, seguidamente, dactilografar no aparelho como uma instrução directa:

SAVE "map" SCREEN$

Para o voltar a introduzir (LOAD), execute a rotina habitual e utilize:

LOAD "map" SCREEN$

Alguns dos últimos capítulos deste livro partem do princípio de que o leitor já desenhou um mapa (pode ser muito mais simples do que aquele que aparece na figura 1.1) e o guardou em fita. Por isso, vá praticando!



Às vezes, o principal problema ao escrever um programa consiste em decidir exactamente aquilo que o aparelho deve fazer... Isso acontece neste programa utilitário de imagens, que sombreia regiões do visor sujeitas a algumas condições que devem ser expressas no próprio programa.

2

PREENCHIMENTO DE ÁREAS DELIMITADAS

A ideia primitiva foi a seguinte: «Não era óptimo obter no visor a apresentação de um mapa-múndi cujas zonas de terra aparecessem *sombreadas?*» A esta ideia seguiu-se imediatamente o pensamento: «Mas, usando o programa Sketchpad, isso leva semanas!» Isto conduziu, como é evidente, à ideia de conseguir que fosse o Spectrum a fazer o trabalho todo.

A princípio, isso parece fácil, e, em casos simples, até é. Contudo, não é possível dizer ao Spectrum: «Encontra as curvas fechadas e sombreia-lhes o interior», porque o aparelho não sabe o que é uma curva fechada, nem tem qualquer noção do que seja o seu interior. De facto, esta abordagem não é viável no nível computacional.

Vamos começar com um caso simples e chegar ao mapa por etapas. A tarefa mais simples é encher uma única área fechada como seja um círculo ou um polígono. Eis um título de trabalho:

POLYFILLER

Suponha que temos desenhado no visor um *único* polígono. De momento, vamos ignorar as questões práticas e concentrar-nos na seguinte questão teórica: «Quais devem ser os passos dados pelo computador de forma a preencher o interior do contorno?»

A resposta é simples:

1. Encontrar, para uma dada carreira horizontal, o ponto do polígono que se encontra mais à esquerda, através de uma busca efectuada ao longo da carreira a partir da sua extremidade esquerda.

2. Encontrar o ponto mais à direita, através de uma busca semelhante com início na extremidade direita da carreira.
3. Unir estes dois pontos por uma linha horizontal.
4. Repetir o processo para todas as carreiras.

A maneira de descobrir se um dado ponto foi marcado (PLOT) consiste em usar a função POINT (*Easy Programming for the ZX Spectrum*, de Ian Stewart e Robin Jones, Shiva, p. 36). O valor de POINT (x,y) será 1 no caso de (x,y) ter sido marcado e Ø se o não tiver sido. Consequentemente, o programa que nós queremos é o seguinte (neste programa, utilizei o *l* minúsculo em itálico para o distinguir do numeral 1):

```
1Ø  FOR y = Ø TO 175
2Ø  LET xl = Ø
3Ø  IF POINT (xl, y) = 1 THEN GO TO 6Ø
4Ø  LET xl = xl + 1
5Ø  IF xl < = 255 THEN GO TO 3Ø
6Ø  LET xr = 255
7Ø  IF POINT (xr, y) = 1 THEN GO TO 1ØØ
8Ø  LET xr = xr - 1
9Ø  IF xr > = Ø THEN GO TO 7Ø
1ØØ IF xl > = 256 THEN GO TO 12Ø
11Ø PLOT xl, y: DRAW xr - xl, Ø
12Ø NEXT y
```

Para testar o programa, introduza esta listagem; seguidamente, introduza, por comando directo, por exemplo:

```
CIRCLE 127, 87, 87
```

A seguir, introduza:

```
GO TO 1Ø
```

(Não dactilografe RUN, pois apagaria o conteúdo do visor!)

Este processo é, sem dúvida, lento (todas as rotinas de sombreamento o são, pois, dado que é necessário tratar dos 45 056 pontos do visor, a computação no seu todo é, inevitavelmente, muito longa), mas é eficaz.

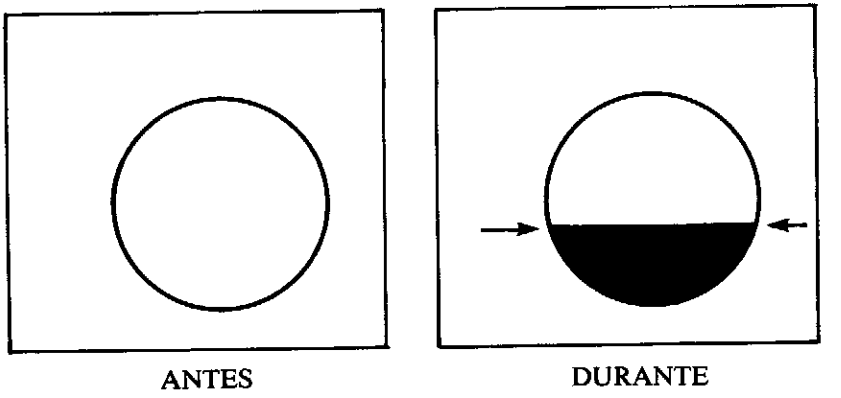


Fig. 2.1 — *Em formas simples, dá resultado o sombreamento partindo do ponto mais à esquerda para o ponto mais à direita.*

Se o combinar com a rotina de desenho de polígonos do capítulo 13 de modo que seja primeiro desenhado um polígono e, seguidamente, o seu interior seja sombreado, vai ver que continua a dar resultado.

DIFICULDADES INESPERADAS

No entanto, quando existem várias áreas a ser sombreadas (e ainda noutros casos), este processo falha completamente. Experimente usá-lo com:

```
CIRCLE 50, 50, 49: CIRCLE 160, 50, 49
GO TO 10
```

Não era esse o resultado que tinha em mente, pois não?

O que o computador fez foi encontrar o ponto mais à esquerda num dos círculos e o ponto mais à direita *no outro;* seguidamente, uniu *esses pontos.* Uma das formas de evitar este contratempo é tentar distinguir as curvas entre si, mas isso é lento e confuso. Além disso, podemos continuar a ter problemas, mesmo com uma única curva. Experimente este caso:

```
PLOT 100, 50: DRAW 50, -50: DRAW -50, 100:
DRAW -50, -100: DRAW 50, 50:
GO TO 10
```

Trata-se de um único polígono fechado, cuja forma lembra um bocado uma ponta de seta, e o programa sombreia uma área excessiva.

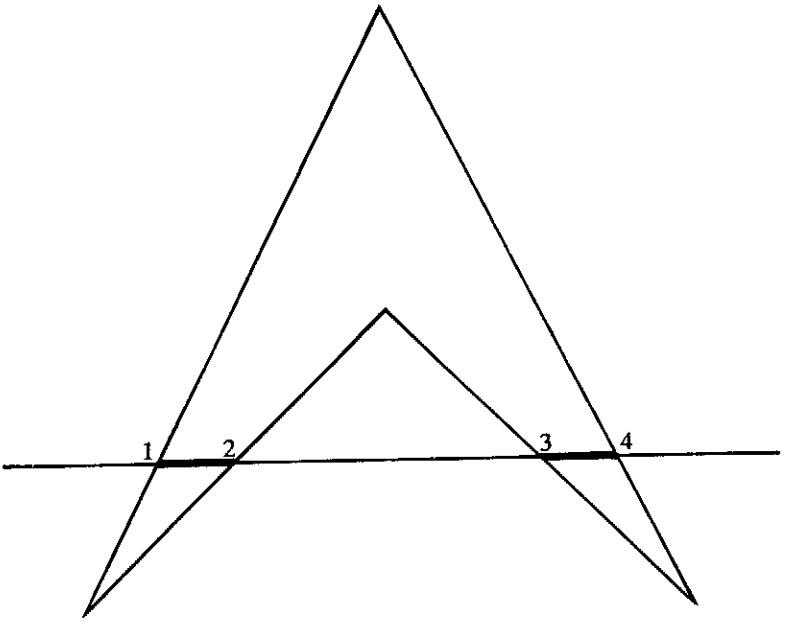


Fig. 2.2 — *Com formas mais complicadas, o processo anterior já não resulta.*

Neste caso, a razão do erro está no facto de algumas das carreiras horizontais, encontrarem o polígono em mais de dois pontos. Por exemplo, uma linha como a da figura 2.2 intersecta-o em *quatro* pontos, e apenas nos interessa sombreá-lo entre o primeiro e o segundo, e o terceiro e o quarto, *não* entre o segundo e o terceiro.

Isto sugere que façamos o seguinte:

1. Percorrer a carreira procurando pontos da curva e tomar nota de todos os pontos encontrados.
2. Unir o primeiro com o segundo, o terceiro com o quarto, e, em geral, unir os pontos assinalados por um número ímpar (2 * i + 1) com aqueles assinalados por um número par (2 * i + 2), tomando i valores de 1 até onde for necessário.

Se o leitor pensa que este processo lhe resolve o problema, tente escrever um programa que o implemente. Seguidamente, teste-o com o nosso polígono em forma de seta.

Ora bolas!

Dá mal logo na primeira linha. Nessa linha só há dois pontos, mas não se quer uni-los, porque são os vértices de duas extremidades salientes do polígono entre as quais se situa uma «baia».

Por meio de desenhos feitos em papel, o leitor deve ser capaz de concluir que, à parte o problema deste tipo de pontas, esta ideia dá resultado. A forma apresentada na figura 2.3, por exemplo, é correctamente sombreada nas carreiras A e B, mas não nas carreiras C e D, que incluem pontas. Note-se que o sistema resulta apesar de estarem desenhadas várias curvas fechadas.

Posto isto, o problema que se nos apresenta agora é como reconhecer uma ponta.

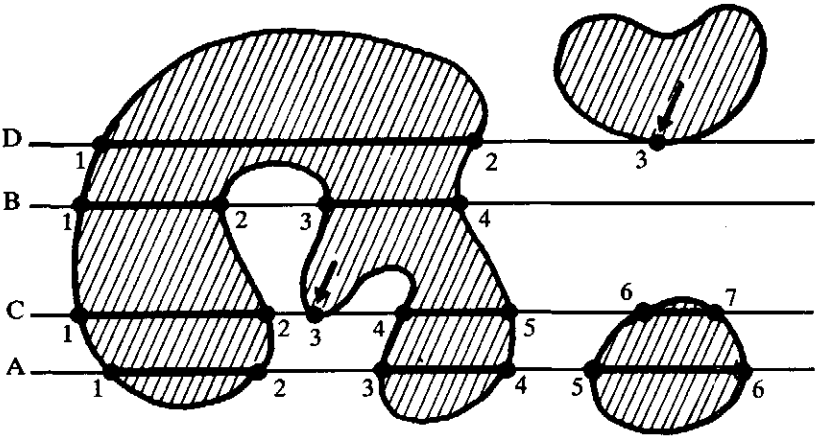


Fig. 2.3 — *O sombreamento entre intersecções ímpares e pares dá resultado, excepto na ponta de uma península, como acontece nas linhas* C *e* D.

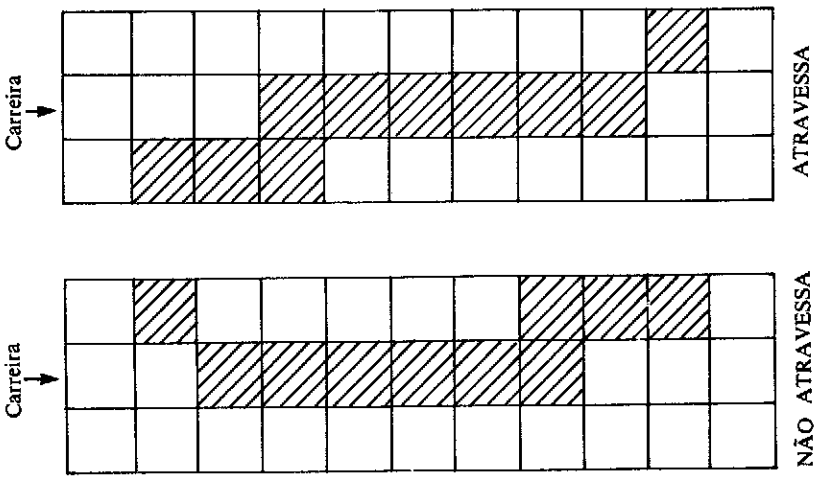


Fig. 2.4 — *Para localizar as pontas das penínsulas, veja se a curva atravessa a carreira ou não.*

É óbvio que a característica principal de uma ponta é que a curva não chega a *atravessar* a linha horizontal: atinge-a de um dos lados, pode coincidir com ela durante uma distância determinada, mas volta finalmente a deixá-la pelo mesmo lado por onde se tinha aproximado. Compare os dois casos ilustrados pela figura 2.4.

Portanto, aquilo que parece que precisamos é de uma rotina que verifique se a curva atravessa a carreira ou não e que, neste último caso, a ignore. Para ver se uma determinada carreira é atravessada, precisamos de saber reconhecer a parte da curva que se lhe *sobrepõe*. Seguidamente, olhamos para os *pixels* que rodeiam as extremidades dessa sobreposição e vemos se se encontram convenientemente preenchidos (ver a fig. 2.5).

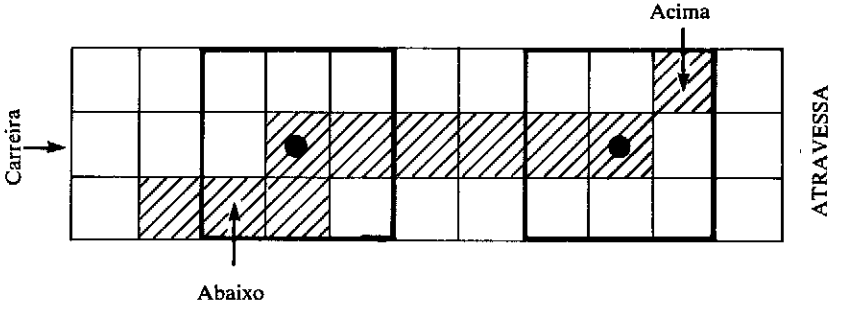


Fig. 2.5 — *Detecção de carreiras atravessadas através da inspecção de dois quadrados de 3×3* pixels.

Na realidade, ainda estamos a ser um bocado ingénuos, porém, temos uma ideia sobre a qual é possível trabalhar, e que leva ao programa seguinte.

SUPERFILLER

```
10  LET track = 1000
20  LET test = 2000
30  LET list = 3000
40  LET shade = 4000
50  DIM a(20)
60  DIM b(20)
200 FOR y = 1 TO 174
205 LET q = 0
210 LET x = 1
```

```
220  IF x > = 255 THEN GO TO 400
230  IF POINT (x, y) = 0 THEN LET x = x + 1: GO TO 220
240  LET xl = x: GO SUB track
250  GO SUB test
260  LET x = xr + 1: GO TO 220
400  GO SUB shade
500  NEXT y
1000 REM track
1010 LET c = 0
1020 IF xl + c > = 255 THEN RETURN
1030 IF POINT (xl + c, y) = 0 THEN GO TO 1060
1040 LET c = c + 1: GO TO 1020
1060 LET xr = xl + c - 1
1070 RETURN
2000 REM test
2005 LET ll = 0: LET lu = 0: LET rl = 0: LET ru = 0
2010 FOR e = -1 TO 1
2020 IF POINT (xl + e, y - 1) = 1 THEN LET ll = 1
2030 IF POINT (xl + e, y + 1) = 1 THEN LET lu = 1
2040 IF POINT (xr + e, y - 1) = 1 THEN LET rl = 1
2050 IF POINT (xr + e, y + 1) = 1 THEN LET ru = 1
2060 NEXT e
2070 IF ll + rl = 0 OR lu + ru = 0 THEN RETURN
2080 GO SUB list
2090 RETURN
3000 REM list
3010 LET q = q + 1: LET a(q) = xr
3020 RETURN
4000 REM shade
4010 IF y < = 1 THEN RETURN
4020 FOR t = 1 TO 20 STEP 2
4030 IF b(t + 1) = 0 THEN GO TO 4100
4040 PLOT b(t), y - 1: DRAW b(t + 1) - b(t), 0
4050 NEXT t
4100 FOR t = 1 TO 20
```

```
4110  LET b(t) = a(t)
4120  NEXT t
4130  DIM a(20)
4140  RETURN
```

Antes de passar a descrever algumas das peculiaridades desta rotina, gostaria de sugerir ao leitor que as experimentasse. Dactilografe-a no seu aparelho e acrescente:

```
1  CIRCLE 50, 50, 48: CIRCLE 50, 50, 44:
   CIRCLE 200, 40, 37: CIRCLE 205, 38, 20
```

Terá assim alguma coisa para sombrear. Agora, RUN o programa. É bastante lento, mas, como a figura 2.6 demonstra, parece resultar.

Vamos agora às explicações. As linhas 200-500 estabelecem o programa principal: trata-se de um *loop* que verifica cada uma das carreiras horizontais (exceptuando a superior e a inferior), procurando ao longo de cada uma delas fragmentos de curva. Se for encontrado algum fragmento, o programa segue para uma sub-rotina *track* que segue a curva ao longo da carreira para encontrar as suas extremidades (como acontece na figura 2.5); o programa dirige-se seguidamente para *test,* que, através do exame da área de 3 × 3 *pixels* que rodeia cada uma das extremidades, determina se a curva atravessa a carreira ou não. No caso de atravessar, a rotina *list* toma nota das coordenadas relevantes.

Depois de ser perscrutada desta forma, a carreira é sombreada por meio da união do primeiro ponto com o segundo, o terceiro com o quarto, e assim sucessivamente, de acordo com o que já tinha sido sugerido. Há, no entanto, uma característica espinhosa: se a carreira for preenchida demasiado depressa, isso interfere com a rotina *test* na carreira seguinte, e o resultado dá asneira. Dado este facto, a lista de pontos é armazenada num *buffer* — a variável indexada *a* —, sendo seguidamente transferida para outra variável indexada *b,* podendo então os pontos ser marcados no visor enquanto a carreira seguinte é perscrutada. As linhas 4100-4130 executam esta tarefa.

Para manter a simplicidade do programa, parte-se do princípio de que as curvas a sombrear *não tocam na linha de fronteira* (carreiras 0 e 175, colunas 0 e 255). Assim, o programa apenas perscruta da carreira 1 à 174 e da coluna 1 á 254.

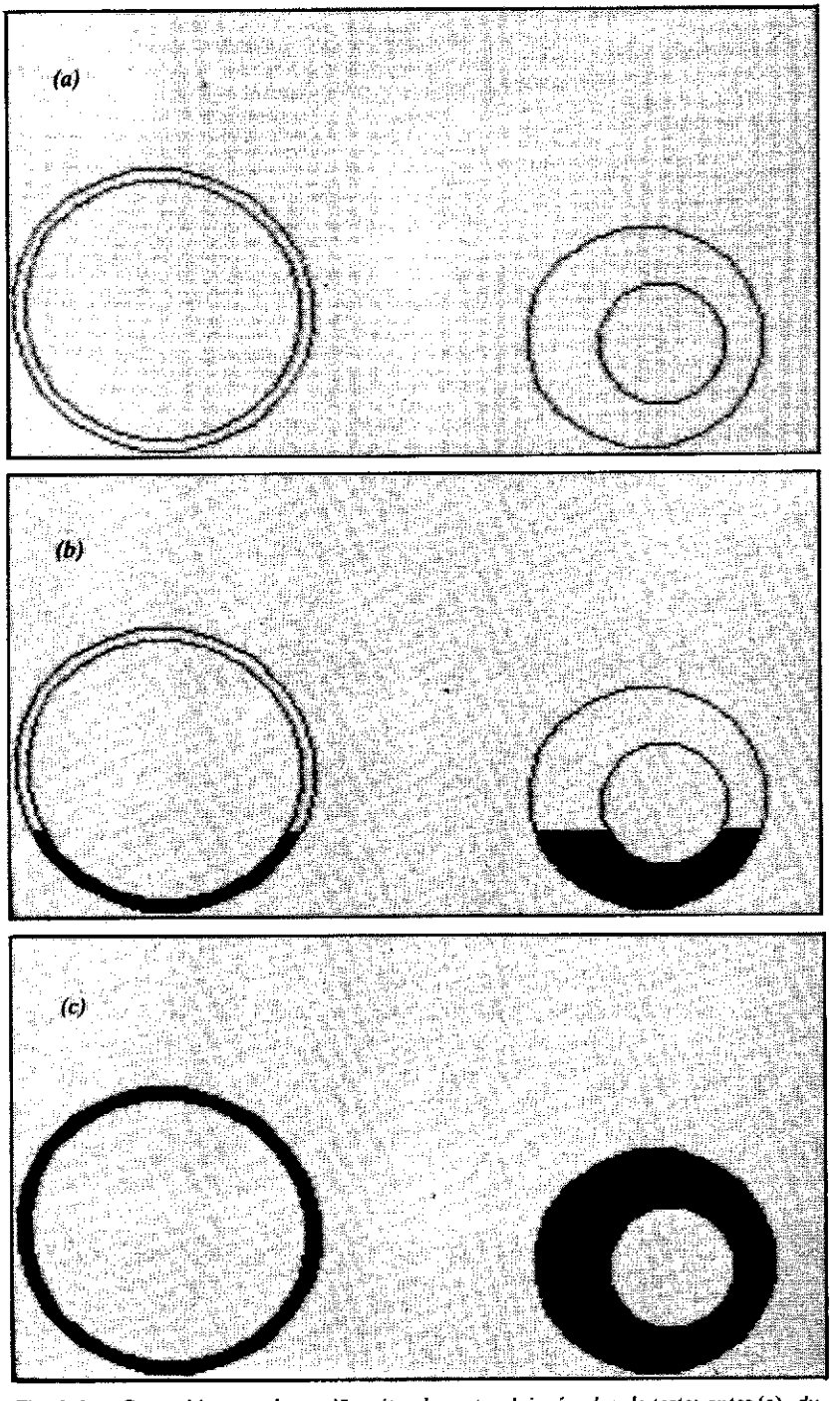


Fig. 2.6 — *Preenchimento das regiões situadas entre dois círculos de teste: antes* (a), *durante* (b) *e depois* (c).

O MAPA

Até agora, vai tudo muito bem, mas será capaz que o programa passa na prova mais difícil? Será que ele é capaz de sombrear o mapa-múndi?

Introduza o seu mapa, utilizando:

```
LOAD "map" SCREEN$
```

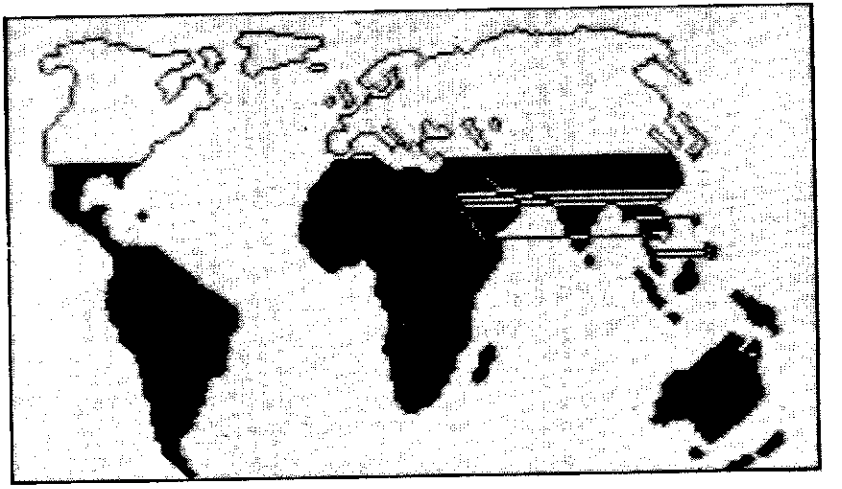


Fig. 2.7 — *Sombreamento de um mapa-múndi delineado — uns quantos gatos...*

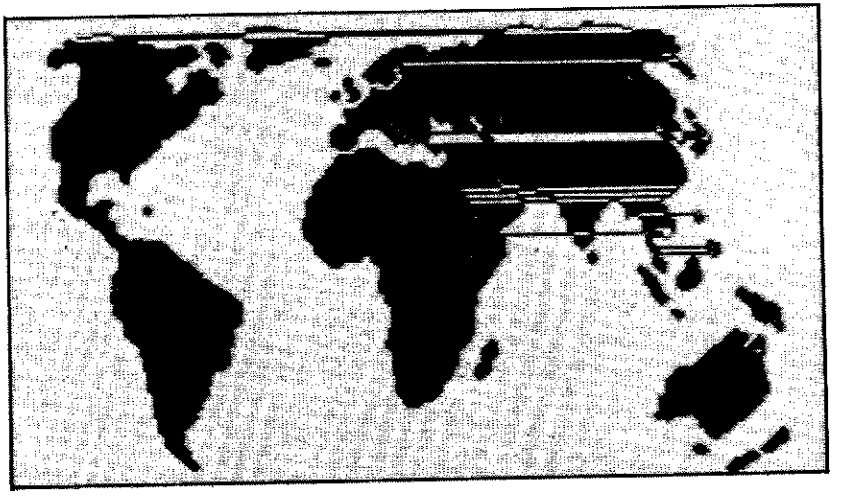


Fig. 2.8 — *...e ainda mais alguns!*

Seguidamente, dactilografe:

GO TO 10

(*Não* carregue em RUN, pois isso apagaria o mapa!)

Talvez o mapa do leitor não fique tal qual o meu. O que eu obtive foi o que se vê nas figuras 2.7 e 2.8.

Não se pode dizer que seja mau de todo, mas também não é perfeito. Há sarilhos na Malásia e confusão nas Aleutas. Que é que não correu bem?

Não é que haja gatos na rotina de sombreamento. O que acontece é que a teoria em que ela se apoia não dá resultado em certos casos irritantes. Se examinarmos detidamente o mapa, é possível atribuir todos os erros de sombreamento a uma de duas causas:

1. «Extremidades pendentes», como é o caso na figura 2.9, *a*;
2. «Curvas que se tocam», como é o caso na figura 2.9, *b*.

O método também falha se as curvas se intersectarem — não foi planeado para tratar dessa possibilidade.

Há duas maneiras de resolver este problema. Uma delas é aperfeiçoar o programa de forma que ele procure as extremidades pendentes e as curvas que se tocam e trate delas; porém, isso é *difícil* e *leva tempo*. A outra consiste em utilizar o programa apenas sobre curvas que não possuam estas indesejáveis características.

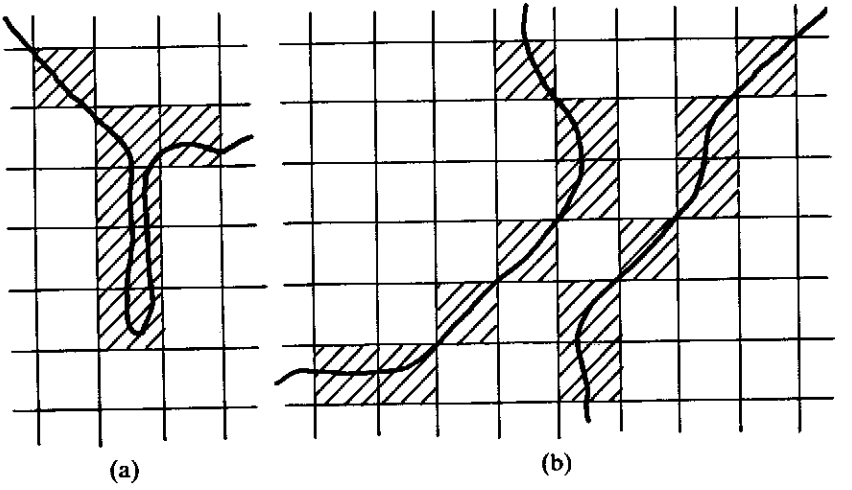


Fig. 2.9 — *As causas dos gatos: extremidades pendentes* (a) *e curvas que se tocam* (b).

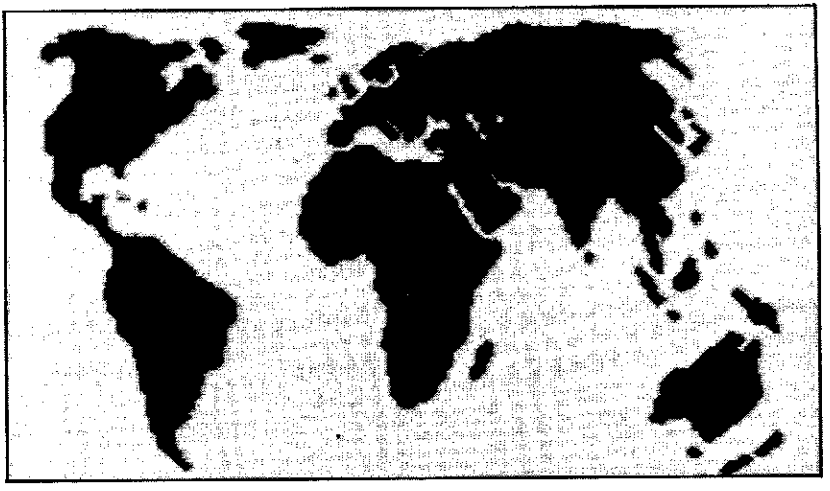


Fig. 2.10 — *Sem gatos e acabado.*

Portanto, se a rotina falhar sobre o seu mapa, como aconteceu com o meu, eis o que deve fazer: use o programa Sketchpad para *modificar* o seu mapa; *seguidamente,* sombreie-o.

É possível que ainda se mantenham algumas falhas: não posso garantir que não existam outras características indesejáveis (interrupções nas curvas causam estragos, mas, de facto, agem como se fossem extremidades pendentes). Contudo, o leitor vai ser capaz de descobrir onde é que elas ocorrem e de usar o Sketchpad para as eliminar. Após algumas tentativas, vai finalmente acabar por obter qualquer coisa semelhante à figura 2.10. Nessa altura, guarde o resultado sob um novo nome, por exemplo

```
SAVE "solidmap" SCREEN$
```

O seu mapa está agora pronto a ser usado em outros programas.

É possível que ache a forma do mapa da figura 2.10 um pouco estranha. Isso deve-se ao facto de eu ter usado como base uma projecção Hammer de área igualada em vez de uma projecção Mercator, mais habitual.

O MÉTODO DA PRADARIA EM CHAMAS

Esta é uma abordagem completamente diferente do problema do sombreamento, e talvez o leitor goste de a considerar. A ideia-base consiste em fornecer ao Spectrum uma pista, «ateando um incêndio»

numa das regiões terrestres. Para o fazer, basta fornecer-lhe (através do teclado ou por meio de um *pixel* móvel) as coordenadas do ponto a incendiar. Por exemplo, o ponto 125,80 situa-se no meio da África.

Agora, deixe o incêndio alastrar, ou seja, preencher todos os *pixels* vizinhos, *excepto se for atingida a linha de costa*. Usando estes *pixels* como novos pontos de partida, espalhe o fogo até mais longe ainda: só irá parar ao alcançar a costa. De facto, o incêndio africano espalhar-se-á pela Ásia e, finalmente, pela Europa. Nunca chegará ao Reino Unido, aos Estados Unidos da América ou à Austrália: o leitor vai ter de introduzir «fagulhas» convenientes para atear incêndios também nestes locais... na realidade, vai precisar de um ponto de ignição para cada porção de terra delimitada por uma linha curva fechada.

Este método não é difícil de programar, mas o esboço acima terá de ser tornado muito mais preciso. Se quiser um projecto interessante, aqui tem um.

LANÇAR FOGO AO MAR

Dado que o mar é bastante mais contínuo do que a terra, é capaz de ser uma ideia melhor iluminar *o mar* (vai ter de proceder separadamente para os mares Aral e Cáspio), tendo previamente estabelecido que a cor da terra é PAPER.

Quando houver uma expressão que surja repetidamente, apresentando as suas variáveis diferentes valores, não utilize uma sub-rotina: experimente usar

3

FUNÇÕES DEFINIDAS PELO UTILIZADOR

Uma função é uma espécie de «caixa preta». O leitor fornece-lhe alguns números ou grupos de caracteres, e em troca ela fornece-lhe outros. Por exemplo, se fornecer à função LEN o grupo de caracteres "gthily", ela devolve-lhe o número 6 — o comprimento do grupo — porque resolve

LEN "gthily" = 6.

O Spectrum tem incorporadas muitas funções, como VAL, COS, TAN, EXP, e por aí fora. Contudo, às vezes o leitor chega à conclusão de que está a usar consecutivamente a mesma expressão com diferentes variáveis. *É possível* implementá-la como uma sub-rotina; porém, geralmente, isso implica uma data de instruções do tipo LETa=39: LETb=21... antes de se poder chamar a rotina.

A tecla DEF FN (E-mode/SYMBOLshift/1) permite-lhe estabelecer as suas próprias funções, e a tecla FN (E-mode/SYMBOLshift/2) chama-as. Cada uma delas deve ser seguida por uma única letra: DEF FNa, e FNa; ou DEF FNb, e FNb; e assim sucessivamente, seguindo o alfabeto. Se o seu valor for um grupo de caracteres, é necessário usar FNa$, FNb$, etc. Também *é possível* usar letras maiúsculas, mas o Spectrum não leva isso em conta, ou seja, FNa e FNA são consideradas como sendo idênticas. Assim, o leitor tem ao seu dispor vinte e seis funções de cada tipo.

Suponha, por exemplo, que chegamos à conclusão de que o nosso programa necessita repetidamente de somar três números e de, seguidamente, dividir o resultado por três, obtendo uma média. Ao longo da listagem está espalhada uma quantidade de expressões do tipo (x+y+z)/3 e (preço1+preço2+preço3)/3. Vamos então estabelecer uma função como esta:

```
10  DEF FNa (p, q, r) = (p + q + r) / 3
```

A partir deste momento, podemos substituir as expressões anteriores por:

```
FNa (x, y, z)
FNa (price1, price2, price3)
```

Esta substituição pode ser feita aonde quer que elas apareçam. Vamos agora testar a função:

```
10 DEF FNa(p, q, r) = (p + q + r) / 3
20 INPUT x, y, z
30 PRINT FNa (x, y, z)
```

Faça passar (RUN) estas linhas, usando dados deste tipo:

1	2	3	(resultado 2)
4	4	4	(resultado 4)
77	91	3	(resultado 57)

Assim, poderá ter a certeza de que a função está correcta.

Sobre este assunto há vários pontos que vale a pena focar. Em primeiro lugar, as variáveis p, q, r que aparecem na definição da função agem como simples marcadores de lugar. Estas letras podem ser utilizadas em qualquer outra parte do programa sem que isso exerça qualquer efeito negativo, além de que a *mesma* função pode ser definida usando outras letras, por exemplo:

```
10 DEF FNa (a, b, c) = (a + b + c) / 3
```

Nem sequer tem importância o facto de a letra que dá o nome à função (neste caso «a») aparecer como uma variável dentro dos parênteses que a definem: o Spectrum é capaz de as distinguir uma da outra.

Seguidamente, no interior dos parênteses têm de ser usadas apenas letras isoladas: na *definição* não podem ser usadas variáveis como preço1. No entanto, não há problema nenhum se se usar preço1 quando, numa parte qualquer do programa, a função é calcular: FNa (preço1, preço2, preço3) dá perfeitamente.

É possível ter na definição variáveis literais, mas elas têm igualmente de ser representadas por uma letra apenas, seguida de $. É possível misturar números e grupos de caracteres. O computador entende-se perfeitamente com, digamos,

```
10 DEF FNb (b, b$) = b * LEN b$
```

apesar de o «b» aparecer em três locais sempre com diferentes signifi-

cados. A função FNb multiplica o comprimento de uma variável literal b$ pelo número b. Se o leitor pedir

```
FNb (7, "cat")
```

recebe como resposta o número 21, que é encontrado através da solução de

```
7 * LEN "cat" = 7 * 3 = 21.
```

A definição de uma função *não necessita* de aparecer no programa antes de a função ser usada; porém, tem de vir mencionada numa parte qualquer do programa (como acontece com as definições de dados).

Como mais um exemplo, experimente a função com valores literais.

```
10  DEF FNm$ (u$, v$) = u$ + u$ + v$
```

e descubra o que obtém para

```
FNm$ ("B", "C")
FNm$ ("a", "gh!")
FNm$ ("co", "nut")
FNm$ ("Zsa□", "Gabor")
```

Todas as definições de função *têm* de incluir os parênteses. Contudo, eles podem não encerrar variáveis nenhumas! Se se escrever

```
10  DEF FNk ( ) = 77
```

o número 77 é-lhe fornecido de cada vez que chamar FNk. (Há ocasiões mais subtis nas quais este tipo de coisa pode ser, de facto, útil, embora aqui tenha um ar meramente perverso.)

Além disso, a definição de função pode conter variáveis que não estejam dentro dos parênteses, desde que essas variáveis estejam atribuídas no programa. Assim,

```
10  DEF FNa (x) = x + q
20  LET q = 5
30  PRINT FNa (10)
```

dá o resultado 15.

Porém,

```
10 DEF FNa (x) = x + q
20 PRINT FNa (10)
30 LET q = 5
```

dá uma mensagem de erro. (AVISO: antes de verificar isto, carregue na tecla CLEAR; caso contrário, o valor de q, da tentativa anterior, permanecerá no aparelho.) Para verificar que a definição pode realmente vir em qualquer lugar, experimente

```
10 LET q = 5
20 PRINT FNa (10)
30 DEF FNa (x) = x + q
```

ALGUMAS UTILIZAÇÕES

Apenas vale a pena definir desta forma uma função no caso de: (1) usar repetida e frequentemente a mesma expressão, ou (2) o seu programa manipular uma função que não é sempre a mesma — tal como acontece com um programa de imagens para marcar o gráfico de uma dada função. Neste último caso, o leitor pode preferir escrevê-lo para manipular, por exemplo, FNa, sendo o utilizador EDIT o valor da função desejada. (Também é possível fazer sorrateiramente um bocado de batota com VAL, podendo então o valor ser INPUT; porém, esta esperteza paga-se, pois o programa fica mais lento — ver o capítulo 16.)

Por exemplo, a distância entre os pontos (a, b) e (c, d), do visor (sendo o *pixel* a unidade de comprimento), é dada por:

```
10 DEF FNd (a, b, c, d) = SQR ( (a - c) * (a - c) + (b - d) * (b - d) )
```

Esta função constitui a versão para o Spectrum do teorema de Pitágoras. No caso de ter de tratar de muitas distâncias, isto pode ser útil. (E não apenas com problemas matemáticos: o leitor pode ter estabelecido um mapa dos Estados Unidos da América e querer saber que distância vai de Los Angeles a Oklahoma City.)

Tenha presente a possibilidade de usar *valores lógicos.* Lembre-se de que uma declaração lógica, como "x< >4", é considerada pelo computador como tendo um valor *numérico:* 1 se for verdadeira, 0 se for falsa. Considere, por exemplo:

```
10 DEF FNo (u, v) = u > = 0 AND u < = 255 AND v > = 0 AND v < = 175
```

Esta função pode parecer disparatada aos olhos do leitor, mas para o Spectrum é perfeitamente clara. Na realidade, FNo verifica se a posição de um *pixel* (u, v) se encontra no visor ou não:

FNo (u, v) = 1 se (u, v) estiver sobre o visor,
FNo (u, v) = Ø se (u, v) estiver fora dele.

Dado isto, se o leitor precisa habitualmente de fazer este tipo de verificação, eis uma função em que vale a pena pensar.

Como outro exemplo, considere as tarifas postais para envio aéreo de jornais e revistas para fora da Europa. Estas tarifas estão dependentes do peso e da zona (A, B ou C), de acordo com o quadro seguinte:

	Zona A	Zona B	Zona C
Primeiros 1Øg	24	26	29
Cada 1Øg (ou parcelas deste peso) adicionais	11	14	15

Vamos estabelecer uma função definida pelo utilizador, FNp, que nos dê o preço por qualquer peso w gramas para qualquer zona z$ (=«a», «b» ou «c»). A função vai tomar um valor numérico, por isso não precisamos de lhe chamar FNp$. O seu aspecto vai ser:

DEF FNp (w, z$) = qualquer coisa desagradável...

Vamos tratar desta função passo a passo. Primeiro, vamos pensar apenas na zona A. Para simplificar, parte-se do princípio de que o peso é semprte maior do que Ø. Desta forma, os «primeiros 1Ø g» existem sempre, de forma que temos de certeza de pagar os nossos 24 pénis. O peso restante vai ser de w-1Ø. Se este valor for igual ou menor do que Ø, temos o problema resolvido: se não, temos temos de o arredondar por excesso para os 1Ø g *seguintes*.

Para arredondar por excesso um número para o múltiplo de 1Ø seguinte, podemos utilizar esta expressão:

$$-1Ø * INT(-n/1Ø)$$

Experimente usá-la: se n = 43, vamos ter

$-n = -43$

$-n/1Ø = -4.3$

INT (−n/1Ø) = −5 (sim, experimente! INT arredonda *por defeito*)

$$10 * INT(-n/10) = -50$$
$$-10 * INT(-n/10) = 50$$

que é exactamente o que queremos.

Como vai ser necessário utilizar este processo várias vezes, vamos *definir uma função.*

$$DEF\ FNr(n) = -INT(-n/10)$$

Trata-se da função de arredondamento por excesso dividida por 10. Óptimo!

Agora, na zona A, o preço a pagar é

$$24 + 11 * FNr(w - 10)$$

desde que $w > 10$, sendo apenas 24 se $w <= 10$. Hum... valores lógicos! O preço a pagar vai ser

$$24 + (w > 10) * 11 * FNr(w - 10)$$

dado que $(w > 10)$ toma o valor 1 quando $w > 10$, acrescentando a parte suplementar, mas toma o valor 0 quando $w <= 10$, deixando apenas os 24.

Para as zonas B e C obtemos expressões semelhantes, mas com diferentes valores nos lugares de 24 e 11. Como é que vamos proceder à integração destas expressões?

Se usarmos os caracteres minúsculos «a», «b» e «c» para a variável da zona z$, os valores lógicos vão de novo mostrar-se úteis. O número

$$24 * (z\$ = "a") + 26 * (z\$ = "b") + 29 * (z\$ = "c")$$

toma o valor 24 quando z$ = «a», 26 quando z$ = «b» e 29 quando z$ = «c». (Porquê?) Podemos também lidar da mesma forma com os valores 11-14-15.

Tudo isto nos conduz às seguintes definições:

```
10 DEF FNr(n) = -INT(-n/10)
20 DEF FNp(w, z$) = 24 * (z$ = "a") + 26 * (z$ = "b")
   + 29 * (z$ = "c") + (11 * (z$ = "a") + 14 * (z$ = "b")
   + 15 * (z$ = "c")) * (w > 10) * FNr(w - 10)
```

Agora, FNp (w, z$) dá realmente o preço do envio de um jornal de peso w para a zona z$.

Projectos

1. Estabelecer FNt de modo que FNt (x) seja o preço de x latas de cervejas a 65 cêntimos cada uma.
2. Estabelecer FNu de modo que FNu (x, p) seja o preço de x latas de cerveja a p cêntimos cada uma.
3. Estabelecer FNj$ de modo que FNj$ (a$, b$, c$) nos diga qual das variáveis a$, b$ e c$ vem primeiro em ordem alfabética. (Nota: na notação do Spectrum para o ordenamento de variáveis literais, duas variáveis a$ e b$ estão em ordem alfabética se a$ < = b$.)
4. Os preços de registos de jornais nos correios seguem a seguinte tabela:

	Zona A	Zona B	Zona C
Primeiros 10 g	13	15	16
Cada 10 g (ou parcelas deste peso) adicionais	3	4	5

   Defina FNq (w, z$) de modo que esta função lhe dê o preço do registo de um jornal com peso w a ser enviado para a zona z$.
5. Combine FNq e FNp (no texto) de modo a obter uma função FNr (w, z$, y) em que w representa o peso, z$ a zona e y = 0 para jornais não registados, e y = 1 para registados.

Havia um programador que usava sempre tinta roxa pois gostava que os seus programas permanecessem inviolados[1]...

4

CARACTERES DE CONTROLO

O leitor já descobriu com certeza que as teclas da carreira superior, no seu Spectrum, têm um comportamento diferente das outras no que diz respeito a *modes* e coisas do género. Se ainda não deu por isso, faça a seguinte experiência: carregue, sucessivamente, nas teclas

NEW
CAPS SHIFT e SYMBOL SHIFT [para *extended mode*]
Tecla 4 da carreira superior
«dedos»

Vai concluir que tem dedos verdes...

Pelo uso das teclas da carreira superior em *extended mode* (com ou sem CAPS SHIFT), o leitor pode, a partir do teclado, estabelecer cores, FLASH, BRIGHT, mudar a posição de PRINT, etc. O *Manual* explica pormenorizadamente os efeitos de combinações particulares de teclas e *modes,* na p. 115; para nós, a parte importante é a seguinte:

Tecla	Efeito em *extended mode*	
	Sem CAPS SHIFT	Com CAPS SHIFT
1	PAPER azul	INK azul
2	PAPER vermelho	INK vermelha
3	PAPER roxo	INK roxa
4	PAPER verde	INK verde
5	PAPER turquesa	INK turquesa
6	PAPER amarelo	INK amarela
7	PAPER branco	INK branca
8	BRIGHT desligado	FLASH desligado
9	BRIGHT ligado	FLASH ligado
Ø	PAPER preto	INK preta

[1] Trocadilho intraduzível, baseado na semelhança de pronúncia existente, em inglês, entre *inviolate* (inviolado) e *in violete* (a violeta). *(N. da T.)*

EFEITO SOBRE AS LISTAGENS

Para efeitos de teste, introduza algumas linhas de programa:

```
10 REM
20 REM
30 REM
```

Agora, carregue sucessivamente em:

(a) 1
(b) REM
(c) *extended mode*/CAPS SHIFT e 9
(d) ENTER

Observe cuidadosamente o cursor para se certificar de que está mesmo em *extended mode.* O leitor vai ver que *todo* o programa, com excepção da linha 1, faísca na sua direcção. LIST o programa, e verá que ele continua a faiscar.

Repita este processo, usando diferentes teclas da carreira superior e omitindo por vezes o CAPS SHIFT, em (c). Hum... É pena que afecte *toda* a listagem... ou será que...?

Volte à versão faiscante da linha 1; e acrescente:

```
11 REM [extended mode/4]
```

Agora, tudo o que se segue à linha 11 ficou verde (cor 4); porém, continua a faiscar... Não havia no quadro anterior uma indicação referente a FLASH desligado? Talvez aquilo de que precisamos seja:

```
11 REM [extended mode/4] [extended mode/CAPS SHIFT e 8]
```

Assim, vemo-nos livres do efeito faiscante, mas mantemos a cor verde, a partir da linha 20.

Experimente e verá.

CARACTERES DE CONTROLO

Que é que *está* a acontecer?

Se o leitor olhar para a lista de códigos de caracteres do *Manual,* p. 183, vai encontrar no início um grupo (números 6 a 23) que não pode ser impresso. São *caracteres de controlo,* que afectam o comportamento do sistema.

Eis a lista destes caracteres:

Código	Carácter
6	PRINT vírgula.
7	EDIT.
8	Cursor para a esquerda.
9	Cursor para a direita.
10	Cursor para cima.
11	Cursor para baixo.
12	DELETE.
13	ENTER.
14	Número (usado em organização de programas).
15	(Não utilizado.)
16	Controlo de INK.
17	Controlo de PAPER.
18	Controlo de FLASH.
19	Controlo de BRIGHT.
20	Controlo de INVERSE.
21	Controlo de OVER.
22	Controlo de AT.
23	Controlo de TAB.

Estes caracteres têm lugar na memória, como quaisquer outros, mas não se podem PRINT ou LIST. Quando se utilizam as teclas da carreira superior em modo alargado, introduzem-se certos destes caracteres. Por exemplo, a tecla 4 em CAPS SHIFT age sobre o carácter de controlo de INK, dando a cor verde.

Ainda que, numa listagem, um carácter de controlo *não seja visível,* é-o o seu *efeito.* O leitor pode verificar que, de facto, ele se encontra na memória quer espreitando (PEEK) o seu endereço (*Easy Programming,* p. 93), quer através da experiência que se segue. Carregue, sucessivamente, em:

(a) 1
(b) REM
(c) E-mode/1
(d) E-mode/2
(e) E-mode/3
(f) E-mode/4
(g) E-mode/5
(h) E-mode/6

Aqui, E-mode refere-se a *extended mode.* Note-se que, ao contrário do que acontece com o *graphics mode,* é necessário voltar a entrar em E-mode de cada uma das vezes.

O leitor vai ver que, depois de REM, o cursor não se movimenta: limita-se a ir mudando de cor. Agora, use DELETE, mantendo a tecla

correspondente pressionada para auto-repetição. Está a ver o tempo que demora passar através de todos esses caracteres de controlo? Experimente de novo, pressionando repetidamente DELETE, em etapas sucessivas, e observe as mudanças operadas na apresentação. É óbvio que existe na memória um grande número de caracteres que não estão a ser impressos.

LISTAGENS FAISCANTES

Esta facilidade pode, de facto, *ter utilidade,* pois é mais do que um mero truque engraçado. Por exemplo, o leitor pode dar realce às declarações REM numa listagem, para facilitar a visibilidade:

```
1  REM [E-mode/CAPS SHIFT/9] This will stand out [E-mode/CAPS/8]
1Ø REM
2Ø REM
etc.
```

LIST este programa e verifique que a declaração REM continua a faiscar. Agora, guarde-o (SAVE) em fita, NEW, LOAD outra vez... sim, ainda está a faiscar.

À semelhança deste processo, é possível codificar em cor secções de programas, por exemplo sub-rotinas. Os caracteres de controlo são colocados no *início* da primeira linha da rotina (após o número de linha — tudo o que aparece antes do número de linha é ignorado) ou no *fim* da linha anterior. Todas as linhas que se seguirem aparecerão nessa cor.

Para tornar invisível uma listagem, deve estabelecer-se a mesma cor para INK (tinta) e PAPER (papel). (No entanto, LLIST ainda dá o programa correctamente, ou então ele poderá ser listado a partir de uma linha a seguir àquela que contém o carácter de controlo, de forma que isto não serve de protecção contra piratas. Não existe *nenhum* método de protecção contra a pirataria que seja absolutamente eficaz, mas há alguns truques, dos quais este é o mais simples, que ajudam a desencorajar os piratas amadores.)

UTILIDADE EM PROGRAMAS

É possível utilizar em programas os caracteres de controlo para evitar ter de estabelecer INK, PAPER, etc., em todo o lado. Isto revela-se particularmente útil para a criação de apresentações colori-

das, como sejam páginas de título para programas e coisas do mesmo tipo.

Por exemplo, para imprimir a bandeira francesa em qualquer posição r (linha) e c (coluna), use o seguinte programa:

```
10 PAPER 0: INK 7: BORDER 0: CLS
20 INPUT r, c
30 FOR i = 0 TO 2
40 PRINT AT r + i, c; "(E-mode/1) □ □ (E-mode/7) □ □
   (E-mode/2) □ □"
50 NEXT i
```

Nesta listagem, os quadradinhos correspondem a caracteres de SPACE.

Para obter a bandeira italiana, substitua o número 1, na linha 40, pelo número 4.

O leitor não deixou com certeza de notar que a variável da linha 40 aparece gloriosamente listada a azul, branco e vermelho.

Na sequência destes objectivos, propomo-nos algo um tudo nada mais ambicioso: a obtenção da bandeira americana numa única variável literal. É possível PRINT no seu visor uma aproximação da bandeira americana usando caracteres de controlo, servindo-se da figura 4.1 e do *graphics mode*. Vai levar tempo e dar trabalho; porém, quando

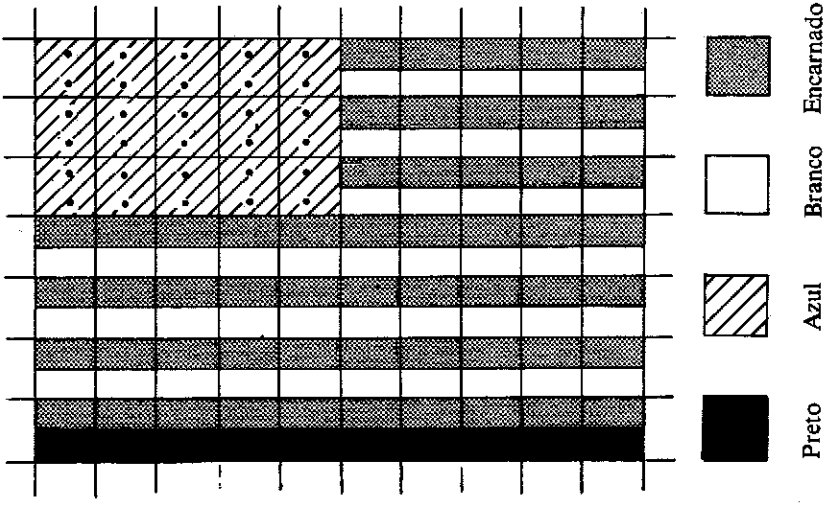


Fig. 4.1 — *A bandeira americana introduzida a partir do teclado numa única variável literal.*

chegar ao fim, o leitor vai compreender bem o funcionamento dos caracteres de controlo a partir do teclado!

Eis uma explicação pormenorizada: o leitor vai vendo a bandeira aparecer à medida que vai trabalhando, e em breve vai conseguir saber de antemão o que vai ser preciso fazer em seguida. Com a prática, vai ser capaz de fazer este tipo de coisa desde o início sem precisar de partir de um esquema rabiscado. (g3c é a tecla 3 em *graphics mode* com CAPS SHIFT.)

```
10 PRINT "(E-mode/1) (E-mode/CAPS/7) : : : : :
(E-mode/2) g3c g3c g3c g3c g3c (E-mode/0)
(22 spaces) (E-mode/1) (E-mode/CAPS/7) : : : : :
(E-mode/2) g3c g3c g3c g3c g3c (E-mode/0)
(22 spaces) (E-mode/1) (E-mode/CAPS/7) : : : : :
(E-mode/2) g3c g3c g3c g3c g3c (E-mode/0)
(22 spaces) (E-mode/2)
(g3c ten times) (E-mode/0)
(22 spaces) (E-mode/2) (g3c ten times)
(E-mode/0) (22 spaces) (E-mode/2) (g3c ten times)
(E-mode/0) (22 spaces) (E-mode/2) (E-mode/CAPS/0)
(g3c ten times) (E-mode/CAPS/7) (E-mode/0)"
```

Ufa! Agora, estabelece-se para PAPER e BORDER preto, para INK branco, e faz-se o programa RUN. A parte das estrelas podia ser melhorada (pense em imagens definidas pelo utilizador), mas vê-se perfeitamente o que a imagem pretende representar. E está tudo numa única variável literal.

Para obter imagens de baixa resolução rápidas e coloridas, o leitor pode utilizar esta técnica de transformar um visor cheio de caracteres gráficos coloridos numa única variável literal, que é impressa no visor quase instantaneamente. Desta forma, qualquer imagem que possa ser planeada numa quadrícula de 64×44 pode ser introduzida, a partir do teclado, sob a forma de uma variável literal com 704 caracteres. Leva tempo e paciência, mas vale a pena para obter uma apresentação atraente, e é um uso de memória eficiente. (Na prática, para facilitar a dactilografia e a edição, sugiro o uso de 5 ou 6 variáveis literais com 128 ou 160 caracteres.)

«Não se trata tanto daquilo que se faz quanto da maneira como se faz.» Os mesmos dados, apresentados de formas diferentes, podem ser claros como água ou parecer uma completa trapalhada. As imagens e as cores facilitam a compreensão. E o leitor alguma vez pensou em utilizar uma fórmula para tocar uma melodia?

5

TÉCNICAS DE APRESENTAÇÃO

Em computação, não se trata apenas de gerar longas listas de *output* numérico, ainda que elas impressionem os burocratas que se encontrem de visita. Também é importante encontrar formas adequadas para a apresentação dos dados. Algumas destas formas são exploradas em vários pontos deste livro. Aqui, vão ser tratadas brevemente cinco formas possíveis de apresentar uma série de números produzida por um processo matemático bastante interessante. O processo em si é discutido no fim, pois conheço muitas pessoas que não se deleitam tanto quanto eu com as matemáticas...

O programa pede que o utilizador escolha o tipo de apresentação desejado entre as cinco opções que constituem o *menu*. Seguidamente, é necessário introduzir um número entre Ø e 1ØØØ. A listagem é de tal forma simples que vai ser dada toda de uma vez:

```
1Ø DIM a(5)
2Ø LET a(1) = 4Ø: LET a(2) = 255:
   LET a(3) = 7Ø4: LET a(4) = 1ØØØ:
   LET a(5) = 7Ø4
2ØØ PRINT "Choose type of display:
     1. Numeric
     2. Graphic
     3. Colour
     4. Sound
     5. Both"
```

```
210  INPUT d
300  PRINT ' "Choose a number between 0 and □□□□ 1000"
310  INPUT k: CLS
320  LET k = k/250: LET x = .7
350  FOR t = 1 TO a(d)
360  LET x = k * x * (1 − x)
370  GO SUB 1000 * d
380  NEXT t
390  STOP
1000 REM numeric
1010 PRINT x,
1020 RETURN
2000 REM graphic
2010 IF t = 1 THEN PRINT k * 250
2020 PLOT t, 0: DRAW 0, 170 * x
2030 RETURN
3000 REM colour
3010 PRINT PAPER INT (8 * x); "□";
3020 RETURN
4000 REM sound
4010 BEEP .05, 20 − 40 * x
4020 RETURN
5000 REM both
5010 GO SUB 3000: GO SUB 4000
5020 RETURN
```

Antes de continuar, talvez o leitor gostasse de experimentar este programa. Qualquer número entre 0 e 1000 é autorizado, mas os números maiores do que 750 dão resultados mais interessantes. A opção 1 imprime listas de números sem grande significado; a opção 2 produz algumas formas pontudas bastante bonitas; a opção 3 desenha barras e blocos coloridos por todo o visor; e a opção 4 toca melodias bastante espantosas, umas vezes rítmicas e repetitivas, outras vezes mais complexas. A opção 5 combina as opções 3 e 4.

UMA ABORDAGEM SISTEMÁTICA

Consideremos este assunto de uma forma mais sistemática, escolhendo o valor do número a ser introduzido e comparando os tipos de apresentação obtidos. Na realidade, vamos tomar quatro valores fixos,

766 88Ø 897 985

que, conjuntamente, ilustram os pontos cruciais.

RUN o programa com a opção 1 e introduza o número 766. Vai obter uma tabela de números em duas colunas. Lendo alternadamente os números sucessivos de uma e outra destas colunas, têm-se valores sucessivos de um número que é calculado pela linha 36Ø do programa. Não é fácil ver neles nada de significativo (é este o problema do *output* numérico tabelado); porém, um olhar mais treinado notaria que, em cada coluna, os valores se vão tornando progressivamente mais semelhantes à medida que se desce. O valor da esquerda aproxima-se de Ø.58 e o da direita de Ø.75. Assim, os números vão alternadamente passando de um dos valores (ou perto dele) para o outro. A sequência de valores é (tende para) algo *periódico,* ou seja, repete indefinidamente os mesmos valores; o *período* é 2.

Repita o processo, com a opção 1, mas introduzindo o número de teste seguinte, 88Ø. O resultado é agora diferente: os números não estabilizam. No entanto, cada coluna tenta alternar entre dois valores: Ø.82 e Ø.87 à esquerda e Ø.51 e Ø.37 à direita. A sequência inteira repete-se de quatro em *quatro* vezes, por isso é periódica de período 4.

Experimente agora a opção 1 com o número 897. Bem... Será que há um padrão ou não? Aparecem bastantes Ø.89 à esquerda e vários valores próximos de Ø.33 à direita; contudo, não é muito claro.

Não se preocupe: a opção 1 com o número 985 ainda é pior. De facto, é uma completa confusão.

A opção 2 facilita-nos muito mais a vida. Para o número 766 dá-nos a figura 5.1, e o comportamento de período 2 é muito claro dada a forma como os bicos sobem e descem sucessivamente. Para o número 88Ø, o período 4 aparece também com muita clareza (fig. 5.2).

Para o número 897 encontram-se traços de periodicidade inegáveis, mas é um pouco irregular (fig. 5.3).

Para o número 985, a disposição dos bicos é bastante ao acaso (fig. 5.4).

A marcação colorida (opção 3) faz salientar as periodicidades de uma forma ainda mais evidente. Em parte, isto acontece porque os períodos 2, 4, etc., são todos divisores do número de caracteres existentes numa carreira (32), o que é uma feliz coincidência. Para o número 766, ver-se-ão riscas verticais verdes e turquesa (excepto mesmo no início),

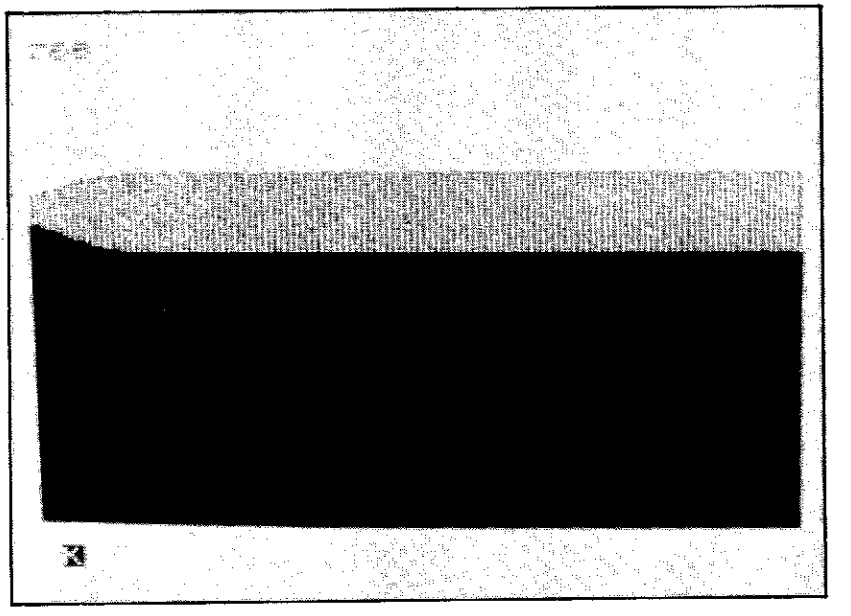


Fig. 5.1 — *Apresentação gráfica: k = 766. Período 1.*

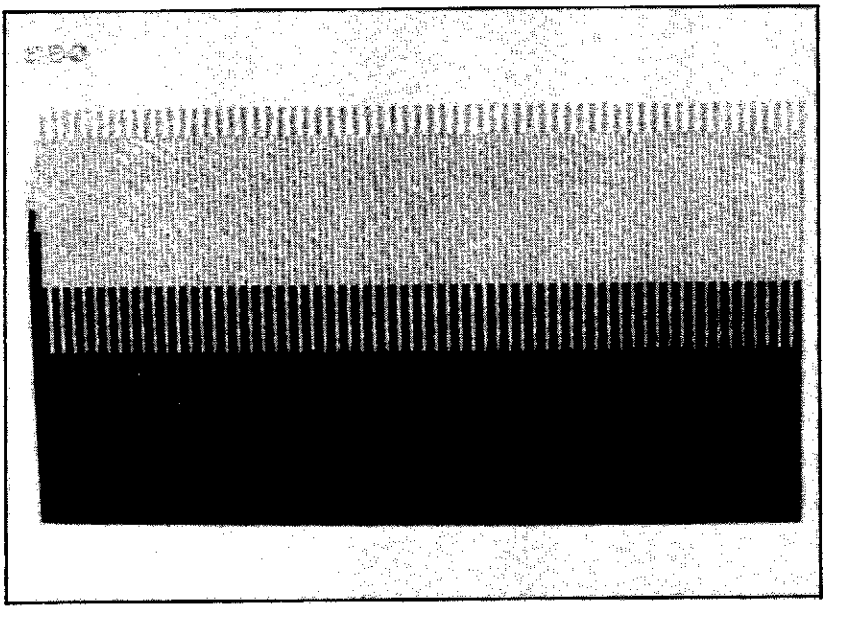


Fig. 5.2 — *Apresentação gráfica: k = 880. Período 4.*

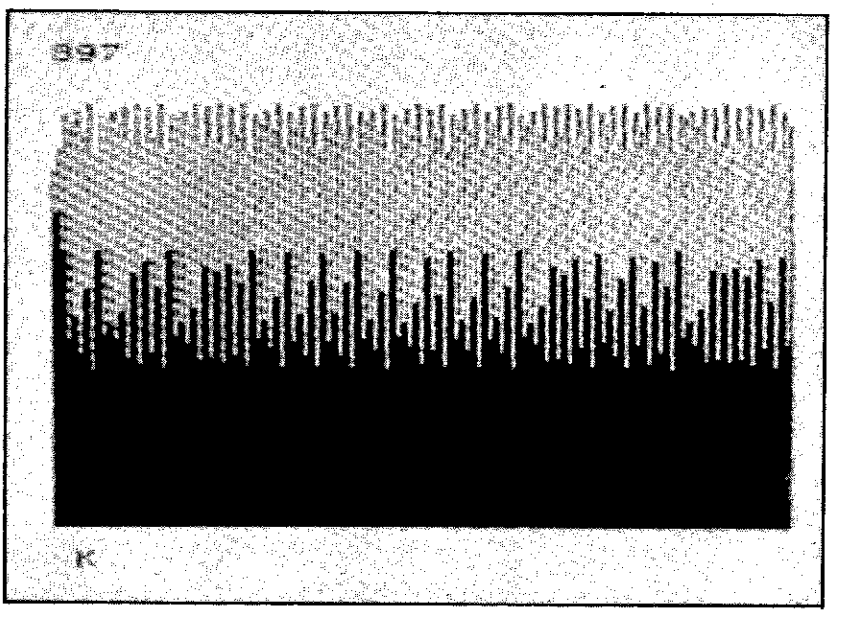


Fig. 5.3 — *Apresentação gráfica: k = 897. Mantêm-se traços de periodicidade.*

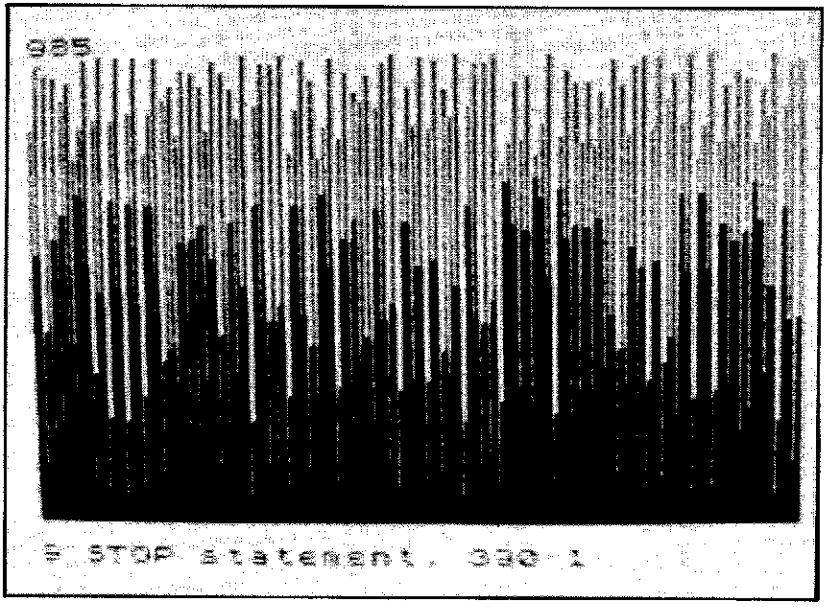


Fig. 5.4 — *Apresentação gráfica: k = 985. É perfeitamente caótica!*

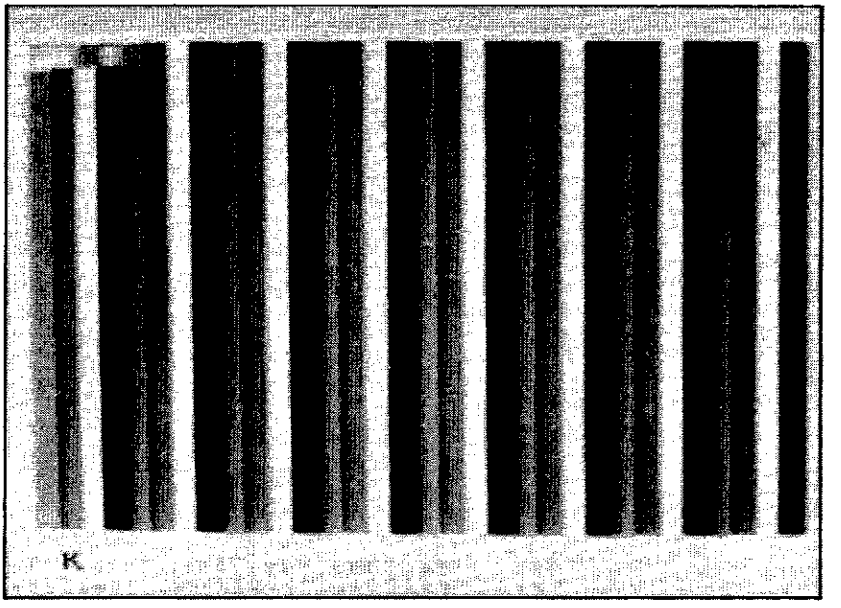


Fig. 5.5 — *Marcação colorida (apresentada aqui a preto e branco). O padrão de barras mostra periodicamente quando* $k = 880$.

Fig. 5.6 — *Marcação colorida (apresentada a preto e branco): quando* $k = 897$, *há periodicidade parcial com lapsos ocasionais.*

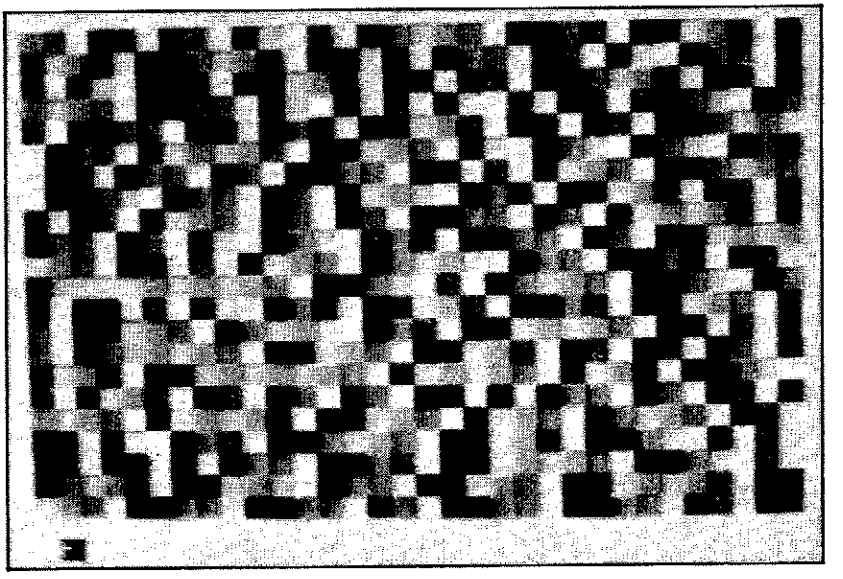


Fig. 5.7 — *Marcação colorida (apresentada a preto e branco): quando k = 985, aparece um padrão caótico de quadrados dispostos ao acaso.*

período 2. Para o número 880 (fig. 5.5), as riscas repetem o padrão quaternário encarnado-amarelo-verde-branco. Para o número 897 (fig. 5.6), aparece um efeito listado inegável, com cores escolhidas no conjunto amarelo-roxo-branco-verde-encarnado; contudo, há lapsos ocasionais, em que um quadrado fica de uma cor diferente da risca a que devia pertencer. Quanto ao número 985 (fig. 5.7), apenas se obtêm quadrados que parecem dispostos ao acaso.

Habitualmente, não se pensa em representar dados por uma *melodia;* no entanto, neste caso, isso dá origem a resultados intrigantes, pois o ouvido apreende a periodicidade como sendo *ritmo*. Passando o programa com a opção 4, o número 766 origina um som monótono, doo/dah doo/dah, como a sirena de uma ambulância (mas com outras notas). O número 880 dá origem a um ritmo mais nítido, repetindo incessantemente uma frase constituída por quatro notas: podia ser usado como fundo para um disco *pop*. O número 897 origina uma agradável mistura de ritmo e irregularidade: certas frases vão aparecendo repetidamente, mas com intervalos irregulares. O número 985 dá origem a uma melodia que soa como mau Schönberg.

A opção 5 permite-lhe ver as cores e ouvir a melodia conjuntamente: estes dois efeitos tendem a reforçar-se mutuamente.

Agora, experimente de novo, dando a k outros valores. Vai verificar que, para números menores do que 75Ø, tudo se fixa sobre um único valor (muito aborrecido) e que o comportamento se vai tornando cada vez mais peculiar à medida que o número aumenta.

MODELAGEM DE POPULAÇÕES

O programa baseia-se numa fórmula que é utilizada em modelos teóricos de populações animais. Imaginem num lago suficientemente grande para sustentar (no máximo) 1ØØ hipopótamos. Como varia o número de hipopótamos, dependendo da taxa de reprodução por geração? Com este programa particular, o número x que é fornecido ao utilizador sob a forma de carácter impresso, de marca colorida ou de som, é dado por

$$\frac{\text{número de hipopótamos na geração actual}}{\text{máximo número possível } (=1ØØ)}$$

e a taxa de reprodução é de 1/25Ø do número introduzido pelo utilizador. Se esta for menor do que 1 ($k < 25Ø$), o número de hipopótamos tende para zero: a população extingue-se lentamente. Se estiver entre 1 e 3 ($25Ø \leq k \leq 75Ø$), o número atém-se a um único valor fixo. Acima disso, oscila de forma cada vez mais descontrolada.

Por exemplo, a sequência de período 2 para $k = 766$ corresponde a ter 58 hipopótamos num ano, 75 no ano seguinte, depois 58 de novo, seguidamente outra vez 75, e assim sucessivamente. O que se passa é que uma população de 58 indivíduos é menos do que o lago pode sustentar, por isso o número de hipopótamos aumenta; no entanto, *ultrapassa* o feliz meio-termo e atinge 78, o que é demasiado. Assim, no ano seguinte, o número volta a cair — mais uma vez sucessivamente — até 58, sendo todo o processo subsequentemente repetido.

O comportamento aleatório observado com $k = 985$ (e parcialmente com $k = 897$) é matematicamente muito interessante, porque nós sabemos que, *de facto,* ele não é absolutamente nada ao acaso. É produzido pela fórmula muito simples que se encontra na linha 36Ø do programa. No entanto, o comportamento *parece* estocástico. É chamado um *caos determinístico* e é uma espada de dois gumes. Por um lado, mostra que acontecimentos aparentemente ao acaso podem ser regidos por uma estrutura simples; por outro, levanta dúvidas sobre a capacidade de efectuar predições úteis de teorias que aparentam ser muito certinhas. Este livro não é o lugar indicado para ensinar ao leitor coisas referentes ao caos; porém, se quiser saber mais sobre este assunto, recomendo-lhe a leitura das pp. 307-318 do livro *Concepts of Modern Mathematics,* de Ian Stewart, publicado pela Penguin Books.

A instrução PEEK (espreitar) permite-lhe ver qual é a informação contida num endereço e a instrução POKE (encaixar) permite-lhe alterá-la. O problema está em saber onde se deve PEEK e o que se deve aí POKE.

6

VARIÁVEIS DO SISTEMA

Como o leitor sem dúvida sabe muito bem, o Spectrum tem dois tipos de memória: a Read Only Memory (ROM — memória apenas para leitura) e a Random Access Memory (RAM — memória de acesso estocástico). É apenas na RAM que os conteúdos de uma localização de memória *(endereço)* podem ser alterados. A maior parte do sistema de funcionamento do aparelho encontra-se em permanência na ROM; contudo, uma parte dele está estabelecida na RAM, de modo a poder ser alterada conforme for necessário. É esta a área de *variáveis do sistema* da RAM, que se estende entre os endereços 23552 e 23733. O *Manual* apresenta, nas pp. 173-176, uma lista muito pormenorizada, mas nem sempre muito clara.

A maior parte destas variáveis não tem uma utilidade por aí além no que concerne à sua incorporação em programas; contudo, há algumas muito úteis. Ainda que lhes sejam dados nomes no *Manual,* estes servem apenas para referência, não sendo directamente acessíveis através do BASIC.

Para descobrir o valor de uma variável do sistema, usa-se PEEK; para o alterar, usa-se POKE (ver *Easy Programming,* p. 93). Para mais pormenores, ver abaixo.

O objectivo deste capítulo é descrever, de entre as variáveis do sistema, aquelas que o leitor tem probabilidade de achar úteis e de lhe fornecer algumas ideias para o seu uso. O que é importante é compreender que as variáveis do sistema *existem* e podem ser modificadas de acordo com a vontade do utilizador, se se proporcionar ocasião para isso.

RESPOSTA AO TECLADO

QUADRO 6.1

Mnemónica	Endereço	Valor-padrão
REPDEL	23561	35
REPPER	23562	5
PIP	23609	0

Estas variáveis controlam o tempo de espera de uma tecla para auto-repetição, a velocidade com a qual esta repetição é efectuada e a duração do BLEEP do teclado ao ser introduzido um carácter. Para que o teclado responda mais rapidamente e com BLEEPs audíveis, introduza (em *command mode* ou a partir de um programa):

```
POKE 23561, 10
POKE 23562, 1
POKE 23609, 50
```

Os números podem variar de acordo com o gosto do utilizador. Neste caso, os limites sensatos para essas variações parecem ser, respectivamente, 10-20; 1-3; 40-100.

Para evitar ter de dactilografar isto de cada vez que liga o aparelho, pode utilizar uma fita com estas instruções e introduzi-la (LOAD) primeiro. É evidente que isto leva ainda mais tempo do que dactilografar as instruções; contudo, se também incluir na fita as suas utilidades favoritas (renumeração de linhas, (ver capítulo 12); mudança de atributos, (ver capítulo 7); algumas imagens definidas pelo utilizador vulgares), ela pode constituir um instrumento útil, que vale a pena incluir na sua programoteca de BASIC.

ORGANIZAÇÃO DE MEMÓRIA

A memória do computador está dividida em blocos, que efectuam diferentes trabalhos. As fronteiras entre estes blocos podem ser mudadas. Os endereços destas fronteiras estão contidos nas variáveis do sistema, de acordo com o quadro 6.2.

Todas estas são variáveis de 2 *bytes*. Isto significa que, para descobrir, por exemplo, onde se encontra o programa, é necessário usar

```
PRINT 23635 + 256 * PEEK 23636
```

QUADRO 6.2

Mnemónica — Endereços		Valor após NEW (aparelho de 16K)
(16384: início do ficheiro de apresentação)		16384
(22528: início do ficheiro de atributos)		22528
(23296: início do *buffer* da impressora)		23296
(23552: área de variáveis do sistema)		23552
(23734: usado para *microdrive*)		23734
CHANS	23631-2	
PROG	23635-6	
VARS	23627-8	
E-LINE	23641-2	
WORKSP	23649-50	
STKBOT	23651-2	
STKEND	23653-4	
RAMTOP	23730-1	32599
UDG	23675-6	32600
P-RAMT	23732-3	32767

Mais geralmente, isto corresponde a (primeiro *byte*) + 256 * (segundo *byte*). Inversamente, para mudar PROG para, digamos, 13244 (não o aconselho a fazê-lo; trata-se apenas de um exemplo para dar uma ideia geral), o leitor vai ter de descobrir a que corresponde 13244 na forma (*byte*) + 256*(*byte*). De facto, o segundo (ou *sénior*) *byte* é dado pelo valor de

INT (13244/256), que é 51,

e o primeiro (ou *júnior*) *byte* é dado pelo valor de

13244 − 256 * INT (13244/256), que é 188.

Assim, a instrução a dar seria

POKE 23635, 188: POKE 23636, 51

Do mesmo modo, para mudar RAMTOP para o valor n (para saber de razões que podem levar a querer fazê-lo, veja mais abaixo), usa-se

POKE 23730, n − 256 * INT (n/256): POKE 23731, INT (n/256)

Neste conjunto de variáveis do sistema, a instrução POKE apenas pode ser usada com proveito com RAMTOP e UDG. No entanto, a instrução PEEK pode ser usada com qualquer delas para descobrir em que zona se encontra cada secção de memória.

CHANS é a área para o sistema de comunicação com *microdrive*, que o leitor não vai achar muito útil. PROG é o início da área de programas em BASIC, e é necessário conhecê-lo, por exemplo, para a rotina de remuneração de linhas do capítulo 12. VARS é o local aonde as variáveis são armazenadas. Se o leitor quiser uma rotina de renumeração de linhas colorida, esta variável do sistema é capaz de lhe ser útil. Ou, se for suficientemente persistente, pode escrever uma rotina para riscar da memória variáveis específicas (uma espécie de CLEAR localizado), ainda que, para este efeito, fosse mais indicada a Linguagem Máquina. As variáveis do sistema desde E-LINE até STKEND são de mais interesse para o Spectrum do que para o seu utilizador.

Contudo, o espaço entre STKEND e RAMTOP é importante: é a quantidade de memória livre disponível. (Na realidade, o aparelho coloca aí dois *stacks* — ver *Machine Code and Batter Basic*—, sendo um deles para a parte Z80A e o outro para os GOSUBs; no entanto, são geralmente muito pequenos.) Assim, para uma estimativa da memória que se encontra livre (com o erro possível de algumas dúzias de *bytes*), usa-se:

```
PRINT PEEK 23730 + 256 * PEEK 23731 - PEEK 23653 - 256 * PEEK 23654
```

ou incorpora-se num programa uma instrução como esta. Na realidade, 256*(PEEK 23731-PEEK 23645) é mais simples e suficientemente aproximado.

RAMTOP está normalmente no início das imagens definidas pelo utilizador; contudo, pode ser baixado para haver espaço para coisas que não quer que sejam afectadas pelo sistema BASIC, como, por exemplo, rotinas em Linguagem Máquina (ver *Machine Code and Better Basic*) ou caracteres definidos pelo utilizador extra (ver abaixo). O que fica acima de RAMTOP não é afectado por NEW — também não é guardado por SAVE, mas é possível contornar este problema através do uso de armazenagem de *bytes* (ver o *Manual*, p. 142).

UDG é onde começa a área de imagens definidas pelo utilizador. A principal razão para querer utilizar isto é estar com falta de memória e não querer utilizar a totalidade dos 23 caracteres definidos pelo utilizador. Nestas condições, o valor é *levantado* para libertar o espaço extra.

RELÓGIO INCORPORADO

A variável do sistema FRAMES conta o número de enquadramentos de visor de TV que foram perscrutados a partir do momento em que o computador foi ligado pela última vez. São perscrutados 50 enquadramentos por segundo, de forma que o aparelho tem de facto um

relógio incorporado. O *Manual,* nas pp. 129-131, trata deste assunto com bastante pormenor, de modo que não vou aqui repetir a descrição; porém, o ponto principal que convém notar é que a variável tem *três bytes:* o seu valor numérico é

PEEK 23672 + 256 * PEEK 23673 + 65536 * PEEK 23674

e o número de segundos passados é este valor dividido por 50. Deve notar-se que o endereço 23674 muda apenas cada 65536/50 = 1310.72 segundos, ou seja mais ou menos cada vinte minutos. Dado isto, para uma série de aplicações (entre elas as que se seguem), os dois primeiros *bytes* são suficientes.

Eis um programa para testar o seu tempo de reacção:

```
10 PRINT "Reaction time test"
20 RANDOMIZE
30 PAUSE (100 + 300 * RND)
40 LET t0 = PEEK 23672 + 256 * PEEK 23673
50 PRINT AT 10, 15, "GO!"
60 IF INKEY$ = "" THEN GO TO 60
70 LET t1 = PEEK 23672 + 256 * PEEK 23673
80 LET t = t1 - t0
90 IF t < 0 THEN LET t = t + 65536
100 PRINT AT 15, 2; "Your reaction time is □"; t/50; "□ seconds"
```

A linha 90 trata da eventualidade (improvável mas possível) de o terceito *byte* de FRAMES ter avançado uma unidade durante o período de reacção.

Para usar o programa, carregue em RUN: assim que aparecer no visor «GO!», prima uma tecla qualquer. (Não faça batota carregando na tecla previamente!)

O CONJUNTO DE CARACTERES

A variável do sistema CHARS contém o endereço do *conjunto de caracteres* — uma tabela de zeros e uns que define a forma dos caracteres do visor. Mais precisamente, a variável contém 256 menos do que o endereço; a tabela começa com o carácter 32, um espaço.

Habitualmente, CHARS toma o valor 15360. As instruções para a impressão do carácter número n estão contidas nos *bytes* 15360 + 8 * n até 15360 + 8 * n + 7 e dão as oito filas do quadrado de 8 × 8 *pixels* desse

carácter em binário, como acontece com os caracteres definidos pelo utilizador (ver *Easy Programming*, p. 49).

O leitor pode PEEK, para ver como um dado carácter está armazenado na ROM e como é construído no visor, usando este programa:

```
10  INPUT "Character ?"; c$
20  IF c$ = " " OR LEN c$ > 1 OR CODE c$ < 32 THEN GO TO 10
30  LET n = CODE c$
40  FOR i = 0 TO 7
50  LET x = PEEK (15360 + 8 * n + i)
60  LET p$ = " "
70  FOR t = 1 TO 8
80  LET xs = INT (x/2): LET xj = x - 2 * xs
90  LET p$ = ("*" AND xj) + ("." AND NOT xj) + p$
100 LET x = xs
110 NEXT t
120 PRINT p$
130 NEXT i
```

Este programa pega nos números binários que se encontram na ROM e transforma 0 num ponto e 1 num asterisco. Assim, a letra «a» resulta no seguinte:

Endereço na ROM	Conteúdos em binário	Efeito gráfico
16136	0 0 0 0 0 0 0 0	. . . . .
16137	0 0 0 0 0 0 0 0	. . . . .
16138	0 0 1 1 1 0 0 0	. . * * * . . .
16139	0 0 0 0 0 1 0 0	. . . . . * . .
16140	0 0 1 1 1 1 0 0	. . * * * * . .
16141	0 1 0 0 0 1 0 0	. * . . . * . .
16142	0 0 1 1 1 1 0 0	. . * * * * . .
16143	0 0 0 0 0 0 0 0	. . . . .

Está a ver a forma de um «a» formada pelas estrelas?

Os pontos e as estrelas são para tornar o objectivo mais claro. Para obter o efeito verdadeiro, mude "." para "□" e "*" para "■".

Pode usar esta técnica em programas para obter letras com um tamanho oito vezes superior ao habitual; e, substituindo cada quadrado por quadrado de 2×2, pode obter um tamanho dezasseis vezes superior ao original — um bocado grandalhão, mas o efeito é dramático.

Através da manipulação dos oito números binários em blocos de 2×2, pode trazer à cena os caracteres gráficos e obter letras com um tamanho quatro vezes superior ao normal (ver acima); e, inventando gráficos definidos pelo utilizador adequados, pode obter caracteres com o dobro do tamanho habitual.

O efeito de POKE sobre CHARS é mais notável.

Introduza na RAM um programa cuja listagem, para obtenção de um melhor efeito, deve ocupar cerca de uma página. Introduza por comando directo:

```
POKE 23606, 2
```

Hum... Ficou um bocado esquisito: os caracteres trocaram um par de linhas e ficaram um tanto espalhados. Agora experimente:

```
POKE 23606, 8
```

(Ignore as mensagens de apresentação do visor, que também parecem esquisitas.) Agora, voltou a ter letras bonitas, mas a listagem parece estar num código qualquer... De facto, cada letra foi trocada pela que se lhe segue no quadro dos caracteres, porque enganámos o computador, fazendo-o olhar oito *bytes* mais à frente do que o lugar onde ele pensa estar.

Para se divertir *a sério,* introduza:

```
POKE 23607, 50
```

O uso judicioso desta instrução num programa tornaria as suas listagens incompreensíveis. No entanto, um pouco de trabalho detectivesco efectuado por um aspirante a pirata torná-las-ia de novo claras... Como a vida é triste...

Para voltar a obter uma apresentação compreensível, introduza:

```
POKE 23606, 0: POKE 23607, 60
```

Ao fazer isto, seja cuidadoso, pois não lhe vai ser possível ver no visor o que está a fazer até estar tudo acabado. Se todos os outros métodos falharem, puxar a ficha por um segundo volta a trazer tudo ao normal.

Tudo isto é extremamente interessante, mas serve para quê? A principal utilidade está em permitir dotar o computador com conjuntos completos de caracteres novos — tantos quantos couberem na memória —, em vez de ter de se contentar com os 23 caracteres definidos pelo utilizador. A forma de realizar este objectivo é explicada no capítulo 15, pois fazê-lo aqui alongaria demasiado este ponto no pre-

sente capítulo. A ideia consiste em construir o novo conjunto de caracteres numa parte da memória que não é usada pelo sistema BASIC, baixando RAMTOP e colocando (POKE) aí os códigos para as carreiras. Seguidamente, efectua-se a pouco usual colocação (POKE) em CHARS, e todos os seus novos caracteres se tornam acessíveis. Se retirar a colocação, voltarão a aparecer os caracteres antigos. Com duas pequenas sub-rotinas para fazer as colocações e 2Ø48 *bytes* de dados obtém 256 novos caracteres — símbolos gráficos, símbolos matemáticos ou o que quiser. Claro que não tem, necessariamente, de efectuar a corveia completa: se preferir, pode estabelecer pelo mesmo método menos do que 256 caracteres.

Através do uso de caracteres gráficos (fila de teclas superior), o leitor pode desenhar símbolos com o quádruplo do seu tamanho normal, o que é bastante agradável para expressar dramatismo. Eis um programa que aceita qualquer variável literal com um comprimento igual ou inferior a 8 e quadruplica o tamanho da sua imagem. (O limite de tamanho serve apenas para que o resultado caiba no visor, e pode ser facilmente modificado se o utilizador quiser escrever várias linhas.) O programa baseia-se na forma inteligente como foram (desta vez!) codificados pelos técnicos da Sinclair os símbolos gráficos. Para obter o código (CODE) de um símbolo gráfico, some os números da quadrícula

2	1
8	4

correspondentes aos seus quadrados *pretos,* e, seguidamente, acrescente ao resultado 128. Por exemplo, ▚ tem o código 2 + 4 + 128 = 134. Pode verificar este assunto na p. 92 do *Manual.*

```
1Ø  DIM q (8, 8)
2Ø  LET iØ = PEEK 236Ø6 + 256 * PEEK 236Ø7
3Ø  INPUT c$
4Ø  LET s = LEN c$: IF s > 8 THEN LET s = 8
5Ø  FOR r = 1 TO s
6Ø  LET i = iØ + 8 * CODE c$ (r)
1ØØ  FOR a = Ø TO 7
11Ø  LET m = PEEK (i + a)
12Ø  FOR b = Ø TO 7
13Ø  LET q(a + 1, 8 − b) = m − 2 * INT (m/2)
```

```
140  LET m = INT (m/2)
150  NEXT b
160  NEXT a
200  FOR u = 1 TO 4
210  LET t$ = " "
220  FOR v = 1 TO 4
230  LET k = q (2 * u - 1, 2 * v) + 2 * q (2 * u - 1, 2 * v - 1)
       + 4 * q (2 * u, 2 * v) + 8 * q (2 * u, 2 * v - 1)
240  LET t$ = t$ + CHR$ (128 + k)
250  NEXT v
260  PRINT AT u, 4 * r - 4; t$
270  NEXT u
280  NEXT r
```

A figura 6.1 mostra o resultado do *input* do «Testing?».

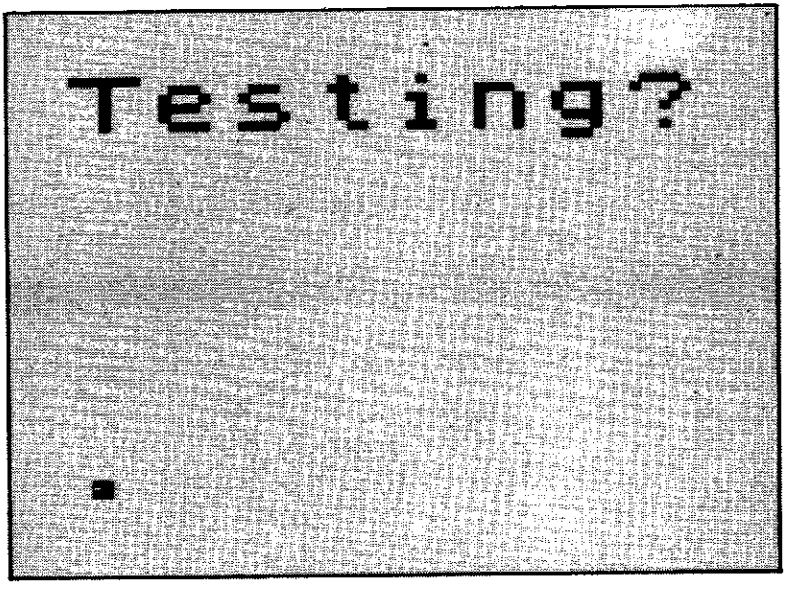


Fig. 6.1 — *Teste da quadruplicação do tamanho de caracteres.*

SALTOS IMPOSSÍVEIS

O Spectrum permite que seja omitida uma grande quantidade de números de linhas (a necessidade de numerar linhas que não vão ser re-

feridas está na base de uma importante crítica feita ao BASIC) escrevendo várias declarações na mesma linha. Por exemplo:

```
10  LET a = 1: LET b = 49: PRINT a + b: GO TO 10
```

Tudo isto está muito bem, mas só é possível saltar (por meio de GO TO) para a *primeira* declaração de uma linha como esta.

A maneira de contornar este problema está em POKE as variáveis do sistema NEWPPC e NSPPC, nos endereços 23618-9 (2 *bytes* para NEWPPC) e 23620 (1 *byte* para NSPPC). Isto obriga a um salto para a declaração cujo número está em NSPPC da linha contida em NEWPPC.

Em vez de estar a explicar a teoria, aqui está um programa que torna tudo claro:

```
 3  INPUT a
 5  POKE 23618, 10: POKE 23620, a
 7  PRINT "I wasn't supposed to come here"
10  PRINT "1": PRINT "2": PRINT "3"
```

RUN este programa e INPUT 1 para a; experimente de novo com a = 2 e a = 3.

A linha 7 nunca chega a ser alcançada. A primeira instrução da linha 5 obriga a um salto para a linha 10. Se a = 1, a segunda instrução da linha 5 obriga a que o salto seja para a primeira parte da linha 10; se a = 2, para a segunda parte; se a = 3, para a terceira.

Em geral, o salto é determinado pela instrução:

```
POKE 23618, nj: POKE 23619, ns: POKE 23620, a
```

Esta instrução tem o mesmo efeito que:

```
GO TO nj + 256 * ns, declaração n.º a
```

(caso esta instrução fosse válida em BASIC, o que não acontece).

Um aspecto peculiar: as declarações IF/THEN. A parte do THEN é tratada pelo computador como uma declaração separada na linha, que tem de ser incluída na contagem. E se, usando NSPPC, se saltar para a parte do THEN, recebe-se uma mensagem de erro «Nonsense in Basic» («Não tem significado em Basic»). Para ver o que acontece, experimente isto:

```
2  INPUT x
3  INPUT a
```

```
5  POKE 23618, 10: POKE 23620, a
7  PRINT "I wasn't supposed to come here"
10 IF x = 0 THEN PRINT "1": PRINT "2": PRINT "3"
```

O computador considera que a linha 10 tem *quatro* instruções:

(IF x = 0) (THEN PRINT "1") (PRINT "2") (PRINT "3")

Dado isto, vai ser necessário experimentar os valores 1, 2, 3, 4 para a. Se x=0, o IF é verdadeiro; se x=1, é falso. O resultado obtido no visor é o seguinte:

x	a	Resultado obtido no visor
0	1	1 2 3 (em colunas)
	2	Nonsense in Basic
	3	2 3
	4	3
1	1	(Nada — a condição não é verdadeira)
	2	Nonsense in Basic
	3	2 3
	4	3

Deve notar-se que, mesmo quando x=1, os saltos para a terceira ou quarta declaração originam uma resposta: o IF/THEN é ignorado (como aconteceria se as declarações estivessem em linhas separadas e tivesse sido usado um GO TO adequado).

A PARTE INFERIOR DO VISOR

Os atributos para a parte inferior do visor (onde aparecem as mensagens de erro) são estabelecidos pelo aparelho. Ocasionalmente, o utilizador pode acabar por ter no fundo do visor um bloco colorido que não condiz com o contorno, se o mudar durante um programa. A variável do sistema BORDCR, no endereço 23624, contém 8 * c, em que c é o número da cor. POKE esta variável não produz um efeito imediato; contudo, se não se importar de apagar as suas variáveis, pode reestabelecer a cor utilizando a instrução:

```
POKE 23624, 8 * c: CLEAR
```

O leitor é capaz de encontrar alguma utilidade para isto. Consulte também o capítulo 7, acerca dos ficheiros de apresentação e de atributos.

Enquanto estamos a tratar da secção de «Mensagens» do visor, convém dizer que se pode chegar a ela de outra forma, através do uso de uma instrução destinada ao *microdrive*. Experimente este programa (# está na tecla 3, *simbol shift*):

```
10 PRINT #0; "Hello there!"
20 GO TO 20
```

(A última serve apenas para impedir a mensagem «OK» de aparecer e obliterar tudo.) Usando PRINT #0, é possível imprimir na secção de «Mensagens». Dão-se alguns factos bizarros: por exemplo, a linha 21 da secção principal tem tendência para se perder e às vezes o visor roda quando menos se espera. É possível PRINT #0 uma mensagem com mais de 64 caracteres: a parte inferior expande-se e roda para cima. Usando instruções de cor e/ou caracteres de controlo, também pode fazer a cor entrar em acção.

Se tem uma impressora adstrita, tente usar, em vez de PRINT #0, PRINT #3. Desta feita, a mensagem vai para a impressora, tal como com LPRINT.

PRINT # 1 também produz mensagens na parte inferior do visor; e PRINT #2 produ-las no local habitual, como simplesmente PRINT. As outras instruções PRINT #n, para n > 3, dizem respeito ao sistema de *microdrive*.

Estes conhecimentos têm utilidade se se quiser imprimir uma série de mensagens, por exemplo, optando umas vezes pelo visor e outras pela impressora. Experimente utilizar:

```
PRINT #n; "Uma mensagem qualquer"
```

Então, se fizer n = 2 a mensagem aparece no visor, se fizer n = 3 ela aparece na impressora. Este método é muito melhor do que utilizar

```
IF n = 2 THEN PRINT "Uma mensagem qualquer"
IF n = 3 THEN PRINT "Uma mensagem qualquer"
```

ROTAÇÃO DO VISOR

A meu ver, a característica mais irritante do Spectrum é a forma como o processo de rotação *(scroll)* do visor se efectua. É necessário passar a vida a carregar no teclado para que o visor rode ou deixe de rodar; comigo, o que acaba quase sempre por acontecer é que o visor roda quando quero que ele pare e pára quando quero que ele rode. O sistema é de uma absoluta perversidade, e eu gostava muito que a Sinclair tivesse dedicado um pouco mais de atenção a este problema.

Se o utilizador quiser forçar o visor a rodar a partir de um programa, pode utilizar a variável do sistema SCR-CT, de endereço 23692. Esta variável contém o número de linhas que ainda faltam até que apareça uma indicação para rodar o visor. Assim, a instrução

```
POKE 23692, 2: PRINT AT 21, 31: PRINT
```

leva o visor a rodar uma linha *sem* que o utilizador tenha de premir uma tecla. (Este facto já tinha sido mencionado em *Easy Programming,* mas vale a pena voltar a referi-lo para que tudo fique completo.)

COORDENADAS DE MARCAÇÃO

Este assunto também já tinha sido mencionado, mas volto a repeti-lo pela mesma razão. A variável do sistema COORDS, com dois *bytes,* contém as coordenadas do *pixel* correspondente ao último ponto que foi marcado no visor (PLOT). Os endereços são

23677 coordenada de linha,
23678 coordenada de coluna.

Assim, se acabou de usar

```
PLOT 47, 124
```

o endereço 23677 contém 47 e o endereço 23678 contém 124.

```
10 INPUT linha, coluna
20 PLOT linha, coluna
30 PRINT PEEK 23677, PEEK 23678
```

COORDS usa-se, é claro, para se saber onde se está imediatamente antes de se utilizar DRAW (que o retira da actual posição de PLOT através dos *offsets* especificados na instrução DRAW, como o leitor sabe muito bem). Se quiser desenhar um ponto específico a, b, e se esqueceu ou não guardou no aparelho a actual posição de PLOT, pode utilizar

```
DRAW a - PEEK 23677, b - PEEK 23678
```

A principal utilização disto encontra-se na marcação de curvas, mas também pode achá-lo útil se utilizar o teclado para movimentar um *pixel* (como acontece no capítulo 1, com o programa Sketchpad) e quiser saber qual é a sua posição.

Já lhe aconteceu ter no visor um diagrama realmente bonito, mas com uma combinação de cores horrível? E a única maneira de alterar as cores consiste em mudar o programa, o que vai apagar o visor e o obriga a si a começar tudo outra vez... Contudo, existe um método melhor.

7

FICHEIROS DE ATRIBUTOS E DE APRESENTAÇÃO

Se o leitor possuiu um ZX81, já sabe que esse aparelho funciona a duas velocidades, sensatamente chamadas FAST (rápido) e SLOW (lento), e que, em FAST, a apresentação no visor desaparece. Também deve saber (ver *Machine Code and Better Basic*) que a apresentação no visor está contida numa parte da RAM chamada o ficheiro de apresentação *(display file),* que é móvel e está contida numa variável do sistema chamada D-FILE. (Se não possuiu nem utilizou um ZX81, salte este parágrafo. Bom, infelizmente é tarde de mais.)

O Spectrum não funciona desta forma.

O *hardware* para a apresentação no visor funciona o tempo todo, não existindo a velocidade SLOW (e, consequentemente, não havendo necessidade da instrução de velocidade). A informação destinada à apresentação no visor está contida em dois *chunks* de memória, o ficheiro de atributos *(attributes file)* e o ficheiro de apresentação *(display file).* Estes encontram-se em posições fixas da RAM e funcionam de maneiras diferentes.

ATRIBUTOS

O ficheiro de atributos é mais fácil de compreender (e, salvo o caso de o leitor ser um perito em Linguagem Máquina, é também mais útil).

A apresentação no visor consiste em 22 linhas (24 se incluirmos a parte inferior, destinada a mensagens), tendo cada uma delas 22 caracteres, o que perfaz um total de 22 * 32 = 704 (ou 24 * 32 = 768) posições.

PRINT AT r, c dá origem ao aparecimento de um carácter na posição n.º c da linha n.º r (contando a partir de Ø). O carácter aí impresso fica no ficheiro de apresentação; contudo, os seus atributos (cor, faiscagem, etc.) vão para o ficheiro de atributos. A ordem pela qual os atributos estão contidos é simples: comece pelo canto superior esquerdo e vá lendo ao longo das linhas, da esquerda para a direita, tal como se lesse um livro. O arquivo de atributos começa no endereço 22528; assim, o atributo para a linha r, coluna c, está contido no endereço:

$$22528 + 32 * r + c$$

É possível POKE para alterar o atributo sem apagar o visor. Isto é útil, porque as instruções PAPER e INK só afectam o material acabado de imprimir.

Cada atributo é um único *byte*, cujos oito *bites* estão divididos da seguinte forma:

FLASH (faiscagem) lig./deslig.	BRIGH (brilho) lig./deslig.	Três *bits*	Para	Cor do papel	Três *bits*	Para	Cor da tinta

Como é habitual, 1 corresponde a ligado e Ø a desligado.

Por outras palavras, sendo o valor de FLASH f, o de BRIGHT b, o de PAPER (papel) p e o de INK (tinta) i, o valor do atributo é:

$$128 * f + 64 * b + 8 * p + i$$

É possível descobrir qual é o atributo da linha r, coluna c, através da instrução:

```
LET att = ATTR (r, c)
```

O atributo pode ser convertido numa lista de valores de f, b, p, i se utilizarmos:

```
LET f = INT (att/128)
LET b = INT (att/64) - 2 * f
LET p = INT (att/8) - 16 * f - 8 * b
LET i = att - 8 * INT (att/8)
```

O programa que se segue converte a apresentação no visor de modo que todas as células tenham o mesmo atributo (escolhido pelo utilizador). Por exemplo, se esteve laboriosamente a introduzir um

mapa-múndi a tinta preta sobre papel branco, e agora quiser tinta vermelha sobre papel amarelo, pode efectuar a mudança sem perder o mapa ou ter de começar outra vez. Usando entrada directa, dactilografe:

```
LET f = 0: LET b = 0: LET p = 6: LET i = 2:
LET att = 128 * f + 64 * b + 8 * p + i: FOR t = 0
TO 703: POKE 22528 + t, att: NEXT t
```

É ainda melhor isto estar previamente na memória, fazendo parte do programa de utilidades geral. O utilizador pode então INPUT f, b, p, i. É evidente que também poderia calcular mentalmente que att = 8 * 6 + 2 = 50, no caso presente. Desta forma, a instrução converter-se-ia em:

```
FOR t = 0 TO 703: POKE 22528 + t, 50: NEXT t
```

Tudo vai lindamente!
Se mudar o *loop* para

```
FOR t = 0 TO 767 etc.
```

A parte inferior do visor também é modificada; se o mudar para

```
FOR t = 704 TO 767
```

será modificada *apenas* essa parte. A altura principal em que isto se mostra útil é quando o utilizador LOAD uma imagem (LOAD "Mona Lisa" SCREEN$) que acaba por ficar com uma cor errada na parte inferior do visor, graças a mudanças operadas em BORDER desde a altura em que a imagem foi guardada (SAVE). Ao contrário de BORDCR, capítulo 6, não é necessário apagar (CLEAR) variáveis. Ainda que não vá precisar disto frequentemente, pode vir a ser-lhe útil.

Para obter com rapidez mudanças de atributos, esta rotina é demasiado lenta. O que tem a fazer é utilizar Linguagem Máquina (ver *Spectrum Machine Code,* de Ian Stewart e Robin Jones, Shiva, capítulo 14).

APRESENTAÇÃO

Aqui, as coisas tornam-se muito mais complicadas, dado que cada carácter de apresentação é guardado como uma lista de *oito bytes* (isto corresponde à forma de 8 × 8 *pixels* apresentada pelos caracteres; ver

capitulo 15). Além disso, *os* bytes *não estão pela ordem que parece óbvia.*

Se o leitor já alguma vez introduziu (LOAD) uma imagem usando SCREEN$, notou com certeza a ordem curiosa pela qual o computador a «pinta». Esta operação é efectuada pela ordem exacta em que os *bytes* estão armazenados no ficheiro de apresentação.

Para ver isto claramente, RUN este programa:

```
10 FOR r = 0 TO 21
20 PRINT "[32 inverse spaces, or graphics character g8c's]"
30 NEXT r
```

Seguidamente, guarde-o: SAVE "blank" SCREEN$. Finalmente, faça CLS e LOAD "blank" SCREEN$, e veja o que acontece. (Se se esquecer do CLS, não se vai divertir lá muito...)

É mais fácil descrever o processo em termos das coordenadas usadas para imagens de alta resolução, uma quadrícula de 256 × 176. Numere as linhas de 175, em cima, até 0, em baixo, e as colunas de 0, à esquerda, até 255, à direita.

O ficheiro de apresentação começa no endereço 16384.

Os primeiros 32 *bytes* contêm as instruções destinadas à linha 175. Cada *byte* governa um segmento dessa linha correspondente a oito colunas. O primeiro *byte* trata das colunas 0-7, o seguinte das colunas 8-15, e assim por diante. Se fizer CLS e, seguidamente, introduzir

```
POKE 16384, BIN 01010101
```

ou o equivalente decimal

```
POKE 16384, 85
```

vai ver, no canto superior esquerdo, uma fila de quatro pontos

. . . .

Mude para POKE 16385, 85, etc., e vai ver como a fila de pontos se move. Os pontos são, evidentemente, o número binário 01010101 convertido em representação gráfica, sendo 1 a indicação «marcar um *pixel*» e 0 a indicação «marcar um espaço» sobre a quadrícula de alta resolução.

Cada uma das carreiras horizontais da apresentação é colocada no ficheiro de apresentação sob a forma de uma sequência ordenada de 32 *bytes*.

Infelizmente, as linhas não são de forma alguma colocadas segundo a sua ordem numérica, como acabamos de ver.

As primeiras a entrar são as linhas:

175 167 159 151 143 135 127 119

(pense nelas como sendo as linhas superiores das primeiras oito carreiras de caracteres). Depois, o computador passa para as linhas seguintes:

174 166 158 15Ø 142 134 126 118

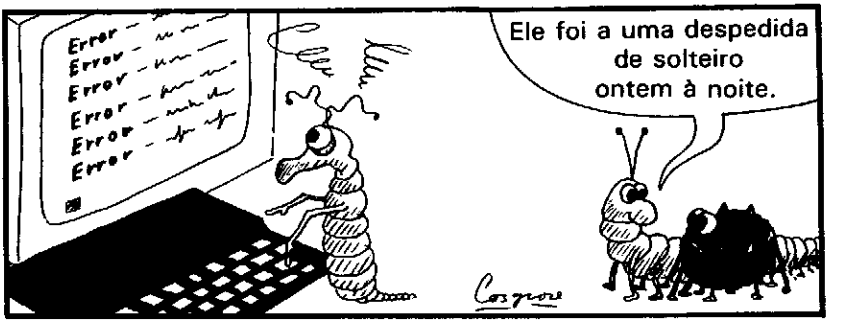
Depois, é a vez das linhas

173 165 157 149 141 133 125 117

e assim sucessivamente, até ser alcançada a linha 122. Nesta altura, o terço superior do visor (em baixa resolução, seriam as carreiras Ø a 7) já está tratado.

O segundo terço é seguidamente tratado segundo uma ordem idêntica; e, finalmente, chega a vez do último terço que inclui a secção «mensagens» do visor.

Este processo foi descrito mais para satisfazer a curiosidade do leitor do que para outra coisa, pois apenas há vantagem em o conhecer para a elaboração de imagens móveis em Linguagem Máquina ou coisas desse tipo. Se quiser *de facto* realizar programas desse género, consulte *Spectrum Machinë Code,* que trata mais em pormenor deste assunto, pois tratá-lo aqui tornar-se-ia demasiado longo.



Projectos

1. Altere o programa de mudança de atributos de modo que sejam dados atributos diferentes a diferentes secção do visor, mediante o uso de declarações de dados adequadas. Use esse programa para obter

um mapa-múndi «sólido», semelhante ao do capítulo 2, mas com os continentes (dentro do que for possível) sombreados em cores diferentes e os oceanos a turquesa.

2. Se tem um espírito matemático, mostre que a sequência de instruções abaixo calcula a posição correspondente no arquivo de apresentação a um *pixel* de alta resolução situado na linha r, na coluna c.

```
10 INPUT r, c
20 LET r1 = 175 - r
30 LET b = INT (r1/64)
40 LET c1 = INT (c/8)
50 LET c2 = c - 8 * c1
60 LET r2 = INT (r1/8)
70 LET r3 = r1 - 8 * r2
80 LET a = 2048 * b + 256 * r3 + 32 * r2 + c1 + 16384
90 LET bitpos = c2 + 1
100 PRINT "The pixel with hi-res coordinates □"; r; "□"; c;
    "□ is stored in the display file at address □"; a; " □ at bit number □";
    bitpos
```

(Aqui, a posição dos *bits*, bitpos, vai de 0 a 7 ao longo do *byte*, partindo do número mais significativo para o menos, desta forma: 01234567.)

Houve com certeza uma boa razão para a Sinclair ter usado esta ordem em particular...

Agora é a vez de o Spectrum se desforrar, deitando uma vista de olhos às variáveis do sistema do leitor...

8

PSICOSPECTROLOGIA

...A arte da experimentação psicológica com o Spectrum.

A ideia subjacente a este capítulo é usar o computador para investigar certos fenómenos curiosos da psicologia da percepção visual, ou seja, do modo como vemos as coisas. O computador é ideal para este objectivo, pois é capaz de estabelecer imagens cuidadosamente controladas («estímulos», como dizem os psicólogos), para pôr em relevo características específicas do mecanismo perceptivo. (Vejam como eu já vou dominando a gíria, apesar de mal ter entrado no assunto.)

IMAGENS RESIDUAIS

Se olhar fixamente para um objecto brilhante durante um período suficientemente longo, as células do olho que recebem a luz «cansam-se» e deixam de responder de forma tão pronunciada. Este facto leva à formação de «imagens residuais», ou seja, de manchas escuras com a mesma forma do que o objecto brilhante original. O efeito desaparece após alguns segundos.

Suponhamos que o objecto original é não apenas brilhante, mas *colorido*. Que será que acontece às imagens residuais? O programa seguinte permite-lhe descobri-lo, usando o Spectrum para gerar e manter os estímulos originais.

```
10 PRINT AT 19, 0; "Afterimages"
20 PRINT "Stare at the square"
30 INPUT "Colour?", p
40 FOR i = 1 TO 6
50 PRINT BRIGHT 1; PAPER p; AT 8 + i, 13; "□□□□□□"
60 NEXT i
```

```
70 PAUSE 500
80 CLS
90 GO TO 10
```

Para RUN este programa, introduza a cor premindo o número correspondente na carreira de teclas superior. Aparece um quadrado brilhante: olhe para ele fixamente, tentando não deixar de o fitar. Quando o quadrado desaparece, o leitor deve ver um difuso quadrado fantasma que se desvanece em poucos segundos. (No caso de não o conseguir ver, ou de o ver muito pálido, experimente aumentar PAUSE (pausa) para cerca de 1000.)

Se os seus olhos funcionarem como os meus, as cores que vê devem ser sensivelmente as seguintes:

Original	Imagem residual
Preto.	Branco.
Azul.	Amarelo.
Vermelho.	Turquesa.
Roxo.	Verde.
Verde.	Roxo.
Turquesa.	Vermelho.
Amarelo.	Azul.
Branco.	Preto.

Bom, na realidade, é possível que o «preto» pareça cinzento e o vermelho seja um bocado acastanhado, mas o resultado não deve andar muito longe do que foi dito.

Note que os códigos para o original e a imagem residual somam sempre 7 (por exemplo, vermelho + turquesa = 2 + 5 = 7). Isto significa que a cor da imagem é *complementar* da do original, ou seja, que contém exactamente os sinais de cor que *não* se encontram presentes no original.

Qual será a razão deste facto? Presumivelmente, a superexposição às cores do original reduz a capacidade de resposta das células dos olhos a essas cores, o que se traduz depois numa espécie de resposta exagerada às cores complementares.

Em circunstâncias normais, não se dá muito pelas imagens residuais (a não ser que olhe para o Sol, o que se torna perigoso se for feito durante mais tempo do que uma fracção de segundo, por isso *não experimente*). Isto deve-se ao facto de os olhos saltarem constantemente de um ponto para outro (chama-se a isto movimento *sacádico*); no entanto, esta experiência com o Spectrum mostra-as claramente. (Tenha cuidado para também não olhar para o Spectrum tempo de mais.)

A ILUSÃO DE HERING

O nome desta ilusão é devido a ter sido descoberta por Ewald Hering, em 1861.

```
10 FOR i = 5 TO 255 STEP 12
20 PLOT i, 0: DRAW 255 - 2 * i, 175
30 NEXT i
50 FOR i = 5 TO 175 STEP 12
60 PLOT 0, i: DRAW 255, 175 - 2 * i
70 NEXT i
100 INPUT "Spacing?", q
110 PLOT 0, 87 - q: DRAW 255, 0
120 PLOT 0, 87 + q: DRAW 255, 0
```

Este programa desenha uma quantidade de linhas que radiam a partir do centro do visor. Seguidamente, pede ao utilizador que introduza um valor. Experimente, em primeiro lugar, um valor entre 10 e 20. Aparecem duas linhas, que parecem *curvas* (ainda que a curvatura do visor de TV possa estragar um pouco a ilusão).

No entanto, é claro, dadas as linhas 110 e 120, que têm de ser *rectas*. O efeito criado pelas linhas radiantes induz em erro os nossos olhos.

Experimente este programa usando diferentes valores para o INPUT de "Spacing" (espaçamento) e diferentes combinações de INK/PAPER. Quais são os valores que levam a uma ilusão melhor?

A ILUSÃO DE WUNDT

Foi preciso esperar até 1896 para que alguém tentasse o que era óbvio e fizesse as linhas curvarem-se no outro sentido. O génio responsável por isto foi Wilhelm Wundt, o primeiro homem a sugerir que os psicólogos podiam levar a cabo *experiências*. Carvalhos Grandes e tudo o mais...

```
10 FOR i = -125 TO 125 STEP 10
20 PLOT 127, 0: DRAW i, 87: DRAW -i, 88
30 NEXT i
40 FOR i = 2 TO 87 STEP 10
50 PLOT 0, i: DRAW 127, -i: DRAW 128, i
60 PLOT 0, 175 - i: DRAW 127, i: DRAW 128, -i
```

```
70  NEXT i
100 INPUT "Spacing?", q
210 PLOT 0, 87 - q: DRAW 255, 0
220 PLOT 0, 87 + q: DRAW 255, 0
```

Este programa funciona de forma muito semelhante ao anterior. Parece-me que esta ilusão é mais eficaz sobre o visor do que a versão de Hering. Qual é a opinião do leitor?

A ILUSÃO DE POGGENDORF

Já se deu conta de que os autores de ilusões são todos alemães? Em 1860, Johann Poggendorf propôs a que se segue.

```
10  FOR i = 0 TO 20
20  PLOT 20, 70 + i: DRAW 200, 0
30  NEXT i
40  PAUSE 50
50  PLOT 20, 10: DRAW 200, 130
60  IF INKEY$ = "" THEN GO TO 60
70  OVER 1: PLOT 113, 70: DRAW 30, 20
80  OVER 0
```

Este programa desenha um bloco rectangular e duas linhas que radiam a partir dele. As linhas parecem não estar sobre a mesma recta.

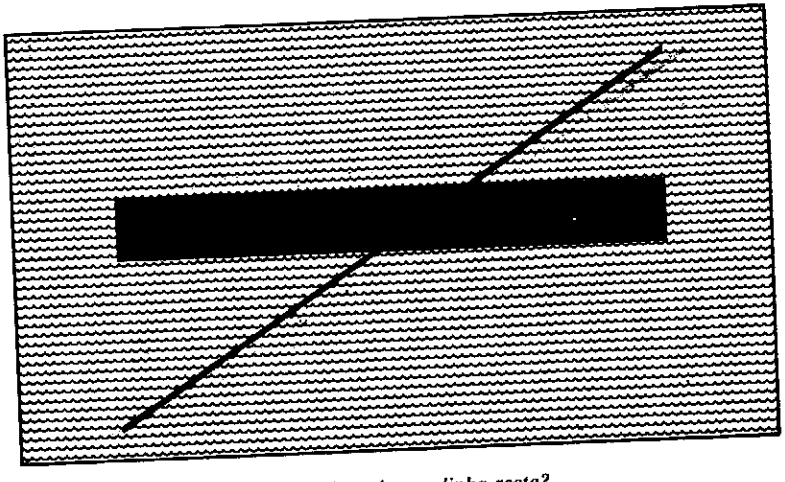


Fig. 8.1 — *A ilusão de Poggendorf: será uma linha recta?*

No entanto, se carregar numa tecla, é desenhada uma linha branca entre as outras duas, mostrando que estas se encontram alinhadas: vê-se agora que se trata de uma linha recta.

A ILUSÃO DE MÜLLER-LYER

Ao princípio, pensei que o nome se referisse a *dois* alemães... Mas não é verdade: o nome de Franz Müller-Lyer escrevia-se com *hífen* (era, como eles dizem, um *Doppelname*).

```
10 PLOT 40, 130: DRAW 180, 0
20 PLOT 40, 60: DRAW 180, 0
30 PLOT 25, 45: DRAW 15, 15: DRAW -15, 15
40 PLOT 235, 45: DRAW -15, 15: DRAW 15, 15
50 PLOT 55, 115: DRAW -15, 15: DRAW 15, 15
60 PLOT 205, 115: DRAW 15, 15: DRAW -15, 15
70 IF INKEY$ = " " THEN GO TO 70
80 FOR t = 1 TO 7
90 PLOT 40, 140 - 10 * t: DRAW 0, -7
100 PLOT 220, 140 - 10 * t: DRAW 0, -7
110 NEXT t
```

O leitor já conhece com certeza esta ilusão. Há duas rectas, cujas pontas (setas) estão viradas numa para fora e na outra para dentro. A de baixo é claramente maior do que a de cima. Está mesmo convencido de que é assim? Então carregue numa tecla qualquer e observe as linhas ponteadas...

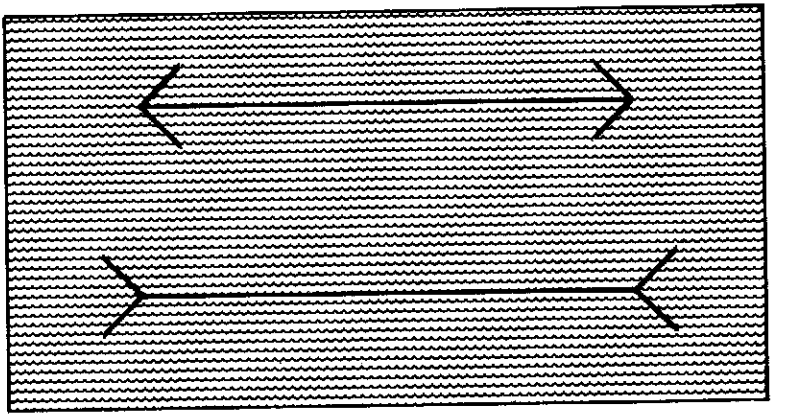


Fig. 8.2 — *A ilusão de Müller-Lyer: será que os comprimentos são iguais?*

A ILUSÃO DE WERTHEIMER

Esta ilusão é devida a Max Wertheimer, que a descobriu em 1912...

```
10 PAPER 0: INK 6: BORDER 0: CLS
20 INPUT "Pause?", p
25 INPUT "Number?", n
30 FOR t = 1 TO 20
35 FOR j = 1 TO n
40 PRINT AT 8, 15 + 2 * (j - 5); "□"
45 NEXT j
50 PAUSE p
55 FOR j = 1 TO n
60 PRINT AT 8, 15 + 2 * (j - 5); "■"
70 PRINT AT 14, 15 + 2 * (j - 5); "□"
75 NEXT j
80 PAUSE p
85 FOR j = 1 TO n
90 PRINT AT 14, 15 + 2 * (j - 5); "■"
95 NEXT j
100 NEXT t
```

Quando o leitor RUN este programa, vai ser-lhe pedido que indique uma pausa (PAUSE) — entre 0 e 100 dá perfeitamente — e um número (NUMBER). Sugiro-lhe que, nas primeiras tentativas, introduza o número 1.

O visor apresenta um (ou dois) quadrados amarelos. Se a pausa for pequena, o leitor vai ver duas luzes (possivelmente faiscantes); se a pausa for grande vai ver uma luz durante um tempo e, seguidamente, a outra. Contudo, no caso de a pausa estar no meio-termo, verá apenas uma luz saltando de cima para baixo. (Experimente com PAUSE = 35.) Este efeito pode ser visto nas estradas britânicas, sendo causado pelas luzes de aviso contra o nevoeiro.

Se o leitor utilizar um número com valor maior que 1 (mas não maior do que 10 ou 12), vai ver uma verdadeira disposição de luzes. Aquilo que vê está até certo ponto dependente do facto de as luzes não faiscarem de forma totalmente sincronizada, pois o programa, em BASIC, leva um certo tempo a percorrer um *loop*. Experimente.

Esta ilusão é importante para os jogos em computador, pois sem

ela as imagens móveis não poderiam funcionar. Pela mesma razão, a televisão e o cinema de animação não existiriam. Não haveria Logie Baird, nem Yogi Bear.

A ILUSÃO DE SCHWINDEL

Esta ilusão é devida a Hans-Wilhelm Schwindel, que a descobriu em 1872. Bom, não foi exactamente assim. Quem a inventou fui eu, mas o leitor nunca o teria adivinhado, a não ser que soubesse que, em alemão, «Schwindel» quer dizer «falsificação». Não duvido de que meia dúzia de alemães a tenham inventado aqui há um século, mas não o fizeram num Spectrum.

```
10 OVER 1
20 POKE 23607, 50
30 FOR i = 1 TO 704: PRINT CHR$ (35 + 10 * RND); : NEXT i
40 POKE 23607, 60
50 LET i = 125 * RND: LET j = 85 * RND
60 LET i1 = 125 * RND: LET j1 = 85 * RND
70 PLOT i, j: DRAW i1, j1
80 PAUSE 20
90 GO TO 50
```

Este programa desenha um visor cheio de pontos de alta resolução colocados ao acaso. (Não carregue em BREAK no meio desta parte, senão vai receber mensagens esquisitas. Se carregar *mesmo* num BREAK, introduza POKE 23607, 60.)

Seguidamente, o programa começa a desenhar linhas. O utilizador vê as linhas irem desaparecendo; porém, mal elas tenham desaparecido, perde completamente a noção do local onde elas se encontravam. O olho detecta *mudanças* no visor que contém os desenhos ao acaso, mas não é capaz de fixar nenhum pormenor da imagem resultante. Isto tem algo a ver com a forma como nos apercebemos das *texturas*.

Para variar, experimente com círculos. Modifique as linhas 50-90, de forma a que seja lido:

```
50 LET i = 50 + 150 * RND: LET j = 40 + 90 * RND
60 LET r = 20 * RND + 10
70 CIRCLE i, j, r
80 PAUSE 20
90 GO TO 50
```

Mais uma vez, o leitor vai vê-los aparecer; porém, a imagem não vai durar. Se não estiver convencido de que os círculos chegam a aparecer realmente, acrescente uma mudança de cor:

```
55 INK 3
```

Agora, vai ver que qualquer coisa está a mudar; contudo, não vai começar a ver os círculos antes que uma boa parte do visor tenha ficado roxa. A mudança de cor parece impressionar o olho (e o cérebro) ainda mais do que a mudança nos *pixels* individuais, e obscurece estes quase por completo.

Muitos dos usos comerciais dos computadores requerem grandes quantidades de dados. Estes dados são armazenados em disco ou em fita magnética sob a forma de...

9

FICHEIROS

No contexto da computação, o que vem a ser um ficheiro? Os cientistas têm o hábito desconcertante de pegar numa palavra de uso corrente e lhe dar um sentido subtilmente alterado para servir os seus objectivos. Os cientistas de computadores fizeram exactamente isso com a palavra «ficheiro». Para eles, essa palavra implica uma colecção, geralmente grande, de itens de dados relacionados com um tópico específico.

Assim, por exemplo, um agente imobiliário teria um ficheiro que consistiria em todas as propriedades que ele tivesse disponíveis para venda. Teria de ter cuidado para, dentro do campo de audição de entendido em computadores, não pedir ao seu secretário que lhe trouxesse o ficheiro sobre o n.° 34 da Avenida da Acácia, por exemplo, pois ser-lhe-á dito com sobranceria que está a usar a palavra incorrecta, dado que aquilo que de facto pretende é apenas um item do ficheiro, a que os entendidos em computadores chamam um *registo (record)*. O nosso agente imobiliário pode retorquir, com bastante justificação, que já usava a palavra «ficheiro» antes de o entendido em computadores ter nascido. Porém, o entendido em computadores evitará sabiamente esta linha de argumentação e esconderá o facto de não ter ripostado fazendo notar uma única característica de um registo, como seja, por exemplo, no caso presente, o actual dono do n.° 36 da Avenida da Acácia, ou o preço que é pedido por essa propriedade, se chama um *campo (field)*. É quase certo que, ao chegar a este ponto, o agente imobiliário se desligue da discussão, visto que a venda de campos, mormente sem autorização de construção, não é coisa que lhe pareça dar muito lucro.

Vejamos outro exemplo de um ficheiro, este de natureza mais pessoal. Suponhamos que o leitor queria guardar pormenores referentes à sua colecção de discos. O ficheiro é a soma total de todas as informações respeitantes a essa colecção. Um registo do ficheiro corresponde à

informação sobre um disco da colecção. Para simplificar, vamos partir do princípio de que todos os discos pertencem à variedade *pop*, tendo em consequência doze pistas com títulos distintos. Mais tarde iremos ver o que havemos de fazer com os *Concertos Brandeburgueses*. Assim, cada registo do ficheiro poderia consistir em campos dados pelo quadro 9.1.

QUADRO 9.1

Número do campo	Descrição	Número de caracteres (máx.)
1	Artista	3Ø
2	Data da compra	6
3	Título da pista 1	2Ø
4	Duração da pista 1 (minutos)	5
5	Título da pista 2	2Ø
6	Duração da pista 2 (minutos)	5
7	:	
8	:	
.	e assim sucessivamente	
.	:	
.	:	
25	Título da pista 12	2Ø
26	Duração da pista 12 (minutos)	3Ø
	Número de caracteres/registo:	336

Cada um dos campos descritos no quadro tem um comprimento fixo. Assim, por exemplo, se o artista for «Pink Floyd», que ocupa apenas 1Ø caracteres (incluindo o espaço entre as palavras), vão ter de ser acrescentados mais 2Ø espaços para serem preenchidos os 3Ø símbolos previamente decididos. O número 3Ø é apenas uma estimativa razoável do número máximo de caracteres que o nome do artista poderia ocupar. Deve servir para a maioria dos casos; ainda dá para «Bonzo Dog Doo-Dah Band»; mas «Dave Dee, Dozy, Beaky, Mick and Titch» já não cabe.

Bom, o leitor pode achar que desta maneira se desperdiça uma grande quantidade de memória e que era melhor permitir que os campos tivessem comprimentos variáveis, sendo cada um deles delimitado por um carácter especial. A única coisa que lhe posso responder é que o seu argumento é bem visto, mas que a adopção dessa sugestão tornaria a programação muito mais difícil do que aquilo que estou preparado para fazer. Seja como for, em alguns casos, pelo menos, o problema do comprimento dos campos não se põe. Por exemplo, o campo da «Data da compra» contém sempre exactamente 6 caracteres: 2 para o dia, dois para o mês e 2 para o ano — Ø31178 significaria «3 de Novembro de 1978». Com a «duração de pista» as coisas passam-se de

forma semelhante, pois a duração de uma pista é *quase* fixa. Reservei espaço para dois decimais, de forma que o tempo possa ser referido como, por exemplo, 3.16 minutos. Isto apenas ocuparia quatro caracteres (contando com o ponto decimal), mas reservei cinco, para salvaguardar a hipótese de uma pista durar dez minutos ou mais.

Como já mostrei, o registo completo ocupa 336 caracteres se se partir destes pressupostos. Assim, se a sua colecção se compuser de 200 discos, o ficheiro vai ocupar 67200 *bytes*, o que ultrapassa em muito a capacidade de memória do Spectrum numa só vez!

Esta característa é típica dos ficheiros de computador: a maior parte das vezes estamos à espera de que eles ocupem mais espaço do que aquele que se encontra disponível na memória principal. Isto significa que têm de ser guardados em qualquer armazém, o que, para os nossos objectivos, quer dizer em fita de *cassette*.

FICHEIROS EM FITA

Que forma deverá o ficheiro tomar na fita?

Antes de responder a esta pergunta, torna-se útil pensar um bocado sobre a maneira como, uma vez obtido, o ficheiro vai ser usado. Uma coisa que vai ser necessária com muita frequência é rever o ficheiro para incluir registos novos e para retirar registos obsoletos. Isto é conhecido pelo nome de *manutenção do ficheiro*.

Bom, não é possível retirar *de facto* qualquer coisa de um pequeno segmento de fita de *cassette*. A única maneira de realizar este trabalho é copiar de uma fita para outra todo o ficheiro menos o pedaço que queremos retirar. A fita original mantém-se inalterada, mas a nova já não contém o registo retirado. A figura 9.1 mostra como esta operação pode ser efectuada, pelo menos em princípio.

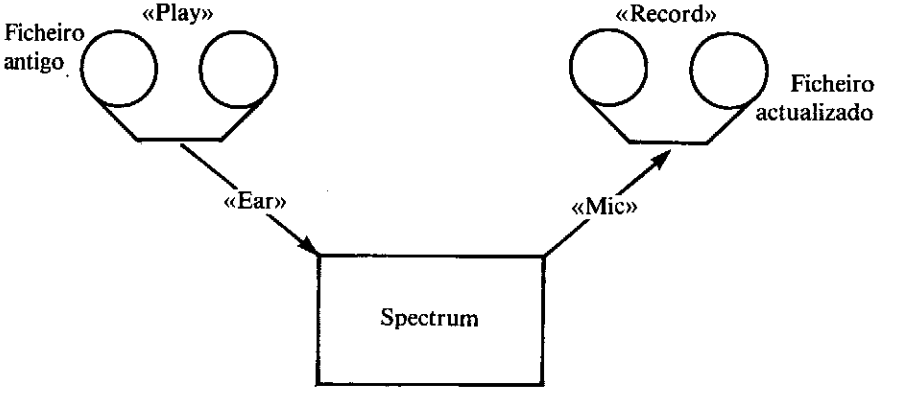


Fig. 9.1 — *Utilização de dois gravadores de* cassettes *para tratamento de ficheiros em fita.*

Assim, precisamos de dois gravadores de *cassettes*, estando um deles em «play» (ligado aos orifícios «ear») e o outro em «record» (ligado aos orifícios «mic»).

No que diz respeito à programação, quais são as considerações a fazer? Em esboço, o programa vai ser o seguinte:

1. Ler um registo.
2. Se for o registo terminal, acabar.
3. Se for o que deve ser retirado, voltar para 1.
4. Escrever um registo.
5. Voltar para 1.

Bom, se deixarmos a fita a correr, não temos a garantia de que quando os passos 2, 3 e 4 tiverem sido executados, o registo seguinte não tenha já passado, de forma que, obviamente, de cada vez que for lido um registo vai ser necessário parar a fita. De igual modo, não faz sentido deixar a correr a fita que está a ser gravada quando não está a ser lá escrito nada. Uma vez que ler e escrever um registo vai levar menos do que dois segundos, a tarefa de parar a fita e de a pôr a correr de novo vai tornar-se aborrecida muito rapidamente.

BLOCOS DE DADOS

Aquilo de que precisamos é um compromisso. Não podemos guardar na memória todo o ficheiro e não queremos tratá-lo registo a registo; porque não havemos então de o dividir em blocos constituídos por vários registos, de modo que seja possível e prático ter na memória um bloco inteiro de cada vez? Num Spectrum de 16 K, a porção de memória disponível para o utilizador é cerca de 9 K. Se o programa ocupar 1 K, temos disponibilidade para a utilização de blocos de 4 K que foi lido para o computador (para um *input buffer*) e cujo conteúdo está a ser transferido para um *output buffer*, para ser subsequentemente gravado em fita. Assim, a nossa organização toma o aspecto que nos é apresentado na figura 9.2.

No caso da colecção de discos do nosso exemplo, conseguimos ter 12 discos em cada bloco.

Para o utilizador, é vantajoso não ter de se preocupar com os aspectos de organização deste arranjo. De facto, torna-se mais simples se *parecer* que estão a ser lidos e escritos registos simples. Para isso, vamos precisar de duas sub-rotinas, «ler um registo» *(read)* e «escrever um registo» *(write)*. Durante a maior parte do tempo, estas rotinas não vão na verdade ler ou escrever nada, procedendo simplesmente à transferência de dados do *input buffer* ou para o *output buffer*. No entanto, quando o *input buffer* estiver vazio, torna-se necessário ler o bloco

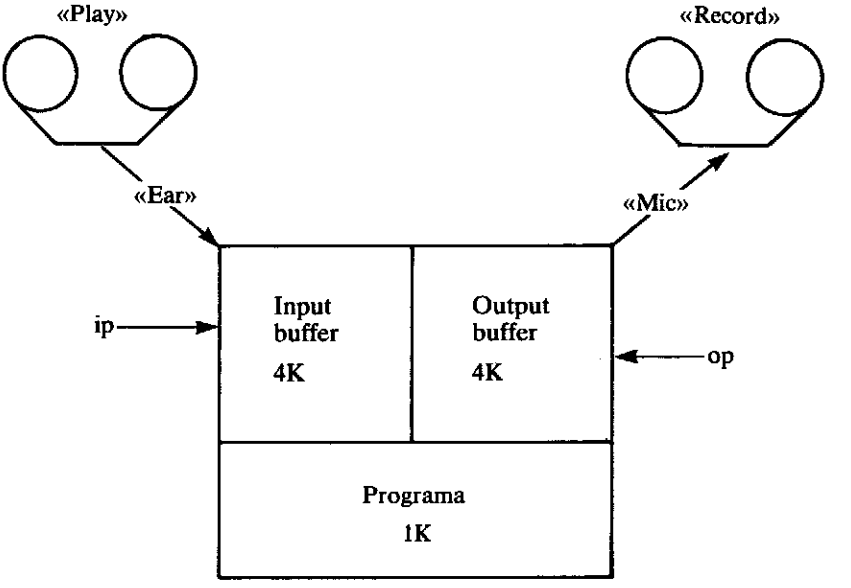


Fig. 9.2 — *Arranjo de memória para tratamento de ficheiros.*

seguinte, e, inversamente, quando o *output buffer* estiver cheio, vai ser preciso escrever um bloco. A estas rotinas vamos chamar *getblock* e *putblock*.

É necessário fazer mais um par de considerações. Primeiro, vamos precisar de dois *indicadores,* ip e op, para mostrarem, em cada momento, até que ponto é que cada um dos *buffers* se encontra cheio. Estes indicadores vão ter de ser, no início, postos a zero. Isto vai ser feito por uma rotina chamada *initcfs* (de «initialize cassette file sistem» — inicializar o sistema de ficheiro em *cassette*), que tratará também de outras inicializações que venham a ser necessárias. Segundo, ainda não pensámos exactamente na maneira como o ficheiro devia ser terminado. Como é evidente, não temos qualquer garantia de que o ficheiro seja constituído por um número exacto de blocos, de forma que precisamos de ter uma maneira de forçar a activação de *putblock* quando for reconhecido um delimitador de ficheiro de qualquer tipo. Vamos adoptar a convenção de o campo 1 do registo terminal ser «cfsend» (a escolha do conteúdo deste campo baseou-se na improbabilidade de este conjunto de letras ter qualquer significado num ficheiro) e os outros campos não contarem, de forma que, na prática, podem conter seja o que for.

Falta acrescentar ainda mais uma coisa. Não dissemos exactamente como é que vão ser estabelecidos o *input buffer* e o *output buffer.* É

claro que são ambos variáveis literais indexadas. Vamos deixar ao utilizador a facilidade de decidir o tamanho que os *buffers* devem ter para uma aplicação particular. O melhor método para fazer isto é o computador pedir a indicação do número de registos por bloco (nrb) e do número de *bytes* por registo (bpr). Nessa altura, o utilizador dimensiona duas disposições:

```
DIM i$ (nrb, bpr)     [input buffer]
DIM o$ (nrb, bpr)     [output buffer],
```

Este processo pode ser efectuado em *initcfs*.

Armados com estas ideias, podemos começar a ver como é que as sub-rotinas *read* e *write* vão funcionar. Em esboço, elas vão apresentar o aspecto seguinte:

Read

1. IF ip=0 OR ip>nrb THEN getblock
2. IF i$ (ip)="cfsend" THEN PRINT "Tentativa de leitura após o fim do arquivo": STOP
3. Transferir i$ (ip) para r$
4. Incrementar ip
5. RETURN

A acção do indicador ip necessita de ser explicada. No princípio, este indicador é posto a zero por *initcfs*, o que, como é lógico, quer dizer que o *buffer* está vazio. Se nesta altura for chamada a sub-rotina *read*, vamos ter de recorrer a *getblock*. A linha 1 trata deste problema. A rotina *getblock* vai ter de dar a ip o valor 1, para indicar onde se encontra o primeiro bloco a ser lido, sendo este seguidamente transferido para r$ (linha 3) para ser usado pelo programa principal. Agora, ip vai ser incrementado, para que, quando *read* for de novo chamada, seja o segundo registo a ser transferido para r$. Isto vai funcionando muito bem até terem sido transferidos todos os registos do bloco, altura em que ip indicará que foi ultrapassada o fim da variável indexada (ou seja, ip vai ser maior do que nrb). É por esta razão que existem duas condições de chamada de *getblock* na linha 1.

Write

1. Incrementar op
2. Transferir r$ para o$ (op)
3. IF op=nrb OR r$="cfsend" THEN putblock
4. RETURN

A linha 1 procede imediatamente à incrementação do indicador op, porque *initcfs* o põe a zero antes do início da variável indexada de *output* o$. Assim, na primeira chamada, o que se encontra em r$ vai ser transferido para o$ (1), que é precisamente o que nós queremos. Sabemos que o *buffer* está cheio quando op = nrb, mas, além disso, queremos obrigar à execução de *putblock* no caso de o nosso marcador de fim de arquivo ter sido ultrapassado. É esta a razão pela qual ambas as condições são testadas na linha 3. A propósito, *putblock* tem de pôr op a zero de novo, de forma que este indicador passa de novo a 1 no princípio da primeira chamada da sub-rotina *write* após ter sido chamada *putblock*.

As rotinas *getblock* e *putblock* também são bastante simples, mas, antes de as esboçarmos, precisamos de ultrapassar mais um problema. Ainda não precisámos como é que vamos chamar aos dados que vão ser guardados. Podíamos dar o mesmo nome a todos os blocos, mas isso criaria confusões e arranjar-nos-ia sarilhos pela certa. É melhor que seja permitido ao utilizador dar nomes aos seus ficheiros (mais uma vez em *initcfs*), sendo subsequentemente o nome alterado automaticamente em *putblock*, de modo que cada bloco tenha um número diferente. Por exemplo, se o utilizador chamar a um arquivo «fred», os blocos serão de facto guardados como sendo fredØ, fred1, fred2, etc. De forma idêntica, *getblock* vai ter de manter um registo do número do bloco em *input*. Mais uma vez, os contadores dos blocos *(block-counts)* vão ser inicializados em *initcfs*.

Posto isto, eis os esboços destas rotinas:

getblock

```
PRINT "ligar gravador PLAY"
LOAD ficheiro de input + n.º de bloco de input DATA i$ ( )
PRINT "desligar gravador PLAY"
LET ip = 1
LET n.º de bloco de input = n.º de bloco de input + 1
RETURN
```

putblock

```
PRINT "ligar gravador RECORD"
SAVE arquivo de output + n.º de bloco de output DATA o$ ( )
PRINT "desligar gravador RECORD"
LET op = Ø
LET n.º de bloco de output = n.º de bloco de output + 1
RETURN
```

Deve notar-se que o n.º do bloco de *input* e o n.º do bloco de *output* (*inblockount* e *outblockcount*) vão ser um pouco mais delicados de manusear do que aparentam, porque umas vezes são usados como variáveis literais a ser acrescentadas aos nomes dos ficheiros, outras vezes são números a serem incrementados. Além disso, agora já temos bastantes variáveis literais, que não são facilmente identificáveis, dado o facto de apenas poderem ter nomes constituídos por um único carácter. Assim, antes de escrevermos a codificação do programa, eis uma lista das variáveis literais e das funções que desempenham:

Nome da variável literal	Função desempenhada
i$	*Input buffer.*
o$	*Output buffer.*
r$	Registo que está aparentemente a ser lido ou escrito.
f$	Nome do ficheiro de *input*.
g$	Nome do ficheiro de *output*.
m$	N.º de bloco a ser acrescentado ao nome do ficheiro de *input* ou de *output*.

CODIFICAÇÃO DO PROGRAMA

Vamos começar o sistema de ficheiro em *cassette* a partir de 95ØØ, deixando 1ØØ para cada rotina, de modo que *initcfs* fica em 95ØØ, *read* em 96ØØ, etc. Assim, vamos precisar das linhas 1 a 3:

```
1 LET initcfs = 95ØØ: LET read = 96ØØ
2 LET write = 97ØØ: LET getblock = 98ØØ
3 LET putblock = 99ØØ
```

Estas linhas são necessárias em qualquer programa que use *cfs*. Escrevamos *initcfs:*

```
95ØØ  LET ip = Ø: LET op = Ø
951Ø  LET inbc = Ø: LET outbc = Ø      [contadores de blocos]
952Ø  INPUT "input file name"; f$
9525  INPUT "output file name"; g$
953Ø  INPUT "no. of bytes per record"; bpr
954Ø  INPUT "no. of records per block"; nrb
955Ø  DIM i$ (nrb, bpr): DIM o$ (nrb, bpr)
956Ø  RETURN
```

Até aqui não há problemas. A rotina *read* deve ser simples:

```
9600  IF ip = 0 OR ip > nrb THEN GO SUB getblock
9610  IF i$ (ip) (TO 6) = "cfsend" THEN PRINT "attempt to read
      past end of file": STOP
9620  LET r$ = i$ (ip)
9630  LET ip = ip + 1
9640  RETURN
```

A rotina *write* também deve ser simples:

```
9700  LET op = op + 1
9710  LET o$ (op) = r$
9720  IF op = nrb OR r$ = "cfsend" THEN GO SUB putblock
9730  RETURN
```

Estas rotinas são bastante semelhantes aos esboços anteriormente apresentados, não acham?

As rotinas *getblock* e *putblock* vão requerer uns certos malabarismos para se efectuarem as conversões de variáveis literais em numéricas:

```
9800  LET m$ = STR$ inbc
9810  PRINT "Turn on PLAY recorder"
9820  LOAD f$ + m$ DATA i$ ( )
9830  PRINT "Turn off PLAY recorder"
9840  LET ip = 1
9850  LET inbc = inbc + 1
9860  RETURN
9900  LET m$ = STR$ outbc
9910  PRINT "Turn on RECORD recorder"
9920  SAVE g$ + m$ DATA o$ ( )
9930  PRINT "Turn off RECORD recorder"
9940  LET op = 0
9950  LET outbc = outbc + 1
9960  RETURN
```

TESTE

Aquilo de que precisamos agora é de um pequeno programa para podermos verificar se tudo isto funciona. Não queremos ter de dactilografar montes de coisas, por isso vamos estabelecer *buffers* pequenos e fazer que o programa gere o seu próprio ficheiro. O mais fácil é geral simplesmente uma sequência de números ascendentes.

Estas linhas devem servir os nossos propósitos:

```
10 GO SUB initcfs
20 FOR n = 1 TO 100
30 LET r$ = STR$ n
40 GO SUB write
50 NEXT n
60 LET r$ = "cfsend"
70 GO SUB write
80 STOP
```

RUN o programa. A primeira coisa que acontece é que *initcfs* lhe pede um nome para o ficheiro de *input*. É óbvio que não há nome nenhum, porque o ficheiro ainda não existe: estamos precisamente a criá-lo. Assim sendo, dê-lhe um nome arbitrário, por exemplo «null». Seguidamente, é-lhe pedido o nome do ficheiro de *output*, que pode ser, por exemplo, «test», ou «fred», ou o que lhe apetecer. Seguidamente, *initcfs* pede-lhe o número de *bytes* por registo. Neste caso, esse número nunca excede 3 (quando n = 100), mas tenha cuidado, porque «cfsend» ocupa 6 *bytes*, pelo que todos os registos têm de ter um comprimento de 6 *bytes* pelo menos. (Se isto não lhe agrada, pode sempre usar um carácter especial, mas olhe que os caracteres de controlo podem causar efeitos estranhos!) Por fim, *initcfs* quer saber qual é o número por bloco. Por razões que dentro em breve se vão tornar claras, deve escolher 20.

Agora o aparelho efectua um bocado de processamento, até ter enchido o *output buffer*, altura em que, como é óbvio, vai ser chamada *putblock*. Será então apresentada no visor a seguinte passagem.

"Turn on RECORD recorder" («ligar o gravador RECORD»)

Uma vez que se segue uma instrução de SAVE, o utilizador recebe então a habitual indicação para ligar o gravador e depois premir uma tecla qualquer. Dado isto, a linha 9910 podia ser dispensada. A única desvantagem seria o facto de a indicação estandardizada da Sinclair não mencionar que o gravador deve estar em RECORD. Vem a propó-

sito dizer que, quando se está a tratar de um ficheiro de *input* e outro de *output*, parece que se precisa de um número de mãos superior ao normal. É possível simplificar ligeiramente a tarefa deixando os dois gravadores respectivamente em PLAY e em RECORD e controlando-os através do botão PAUSE (desde que os seus gravadores tenham esse botão, claro).

Entretanto, no programa, assim que o bloco tenha sido guardado, o leitor recebe uma indicação para desligar o gravador. Neste caso, como o *buffer* se enche muito rapidamente, é recebida quase logo outra mensagem para ligar o gravador em RECORD, de forma que quase nem vale a pena desligá-lo. Se não o desligar entre o SAVE dos vários blocos, apenas acontece que, na fila, os intervalos entre os blocos ficam ligeiramente maiores.

O programa vai guardar um total de seis blocos, dado que os números de 1 a 1ØØ preenchem completamente cinco blocos, o que torna necessária a existência de mais um para conter «cfsend».

Até é preciso verificar se o ficheiro foi, de facto, correctamente guardado. Para o fazer, o método mais simples é modificar as linhas 2Ø, 3Ø e 4Ø de acordo com o que se segue:

```
2Ø GO SUB read
3Ø PRINT r$
4Ø GO TO 2Ø
```

RUN o programa mais uma vez, com a fita no gravador em PLAY. O que deve acontecer é o seguinte: depois de *initcfs* ter pedido os pormenores concernentes ao ficheiro (não se esqueça de que, desta vez, é o ficheiro de *output* que não existe), o sistema indica que o gravador em PLAY deve ser ligado, após o que *getblock* lê 2Ø números, que passa a *read*, que por seu lado os passa, um de cada vez, para r$, sendo depois impressos, e, por fim, aparece uma indicação para o gravador ser desligado. Acontece isto, de facto, mas acontecem também mais algumas coisas. É claro que, antes de tudo, o Spectrum apresenta o nome do ficheiro que está a ler, de forma que devíamos ver:

```
character array: fredØ
```

Isto diria respeito ao primeiro bloco, claro.

PROBLEMAS NA ROTAÇÃO DO VISOR

Na realidade, as coisas passam-se assim, mas com um pequeno senão: o mecanismo de rotação do visor começa a causar aborrecimen-

tos. O sistema pergunta «scroll?» («rodar?»), e, quando o utilizador dá a resposta afirmativa, aparece logo:

"Turn off PLAY recorder" («Desligar o gravador PLAY»)

Assim, se se for lento a voltar a activar a rotação, a fita pode já estar a meio do bloco seguinte antes de o programa o ter começado a ler. (Este problema torna-se particularmente evidente com blocos de 2Ø registos, o que explica a escolha de blocos deste tamanho para o teste.) Isto não constitui um desastre completo porque o Spectrum vai apenas tentar ler o bloco que *devia* ler a seguir, de forma que é sempre possível fazer a fita andar um bocado para trás; mas, seja como for, é bastante maçador. A desactivação completa da paginação durante as chamadas de *getblock* constitui uma alternativa melhor.

Deve lembrar-se, do capítulo 6, que existe uma variável do sistema, chamada SCR-CT, endereço 23692, que podemos levar a causar este efeito. Essa variável contém o número de linhas (mais 1) que vão ser impressas antes de voltar a aparecer a pergunta «scroll?». Podíamos atribuir a esse número o valor 255 (o maior valor possível) sempre que *getblock* fosse chamado, se acrescentássemos a linha:

```
9825 POKE 23692, 255
```

Esta linha tem de estar *depois* do LOAD, pois essa instrução volta a pôr SCR-CT em 1.

Desta forma, temos agora um *output* de 256 linhas antes de aparecer a pergunta «*scroll?*». Se isso é adequado ou não, depende da aplicação. Neste caso, é mais do que suficiente, mas talvez seja mais seguro incluir a declaração também em *read,* além de em *getblock,* pois esta rotina vai ser chamada com maior frequência. Mesmo neste caso, o efeito não é absolutamente garantido, visto que está dependente da quantidade de escrita feita pelo programa no visor entre as leituras; contudo, em circunstâncias normais, não deve haver problemas.

ATÉ AGORA, TUDO VAI BEM...

Vamos parar um pouco, para recobrarmos fôlego e revermos o que tem vindo a ser feito. Em primeiro lugar, devemos notar que *initdfs* foi usado de duas maneiras distintas:

1. Para *criar* um ficheiro a ser subsequentemente escrito.
2. Para *abrir* um ficheiro previamente existente, para ser subsequentemente lido.

Podíamos pois ter escrito, em vez de *initcfs,* duas rotinas separadas, chamadas *create* (criar) e *open* (abrir). Esta abordagem tem vantagens, pois, como vimos, se não quisermos ter um ficheiro de *input* e outro de *output,* temos de inventar nomes de ficheiros-fantasmas para contentar *initcfs.* Além disto, tem de ser o próprio utilizador a tratar da terminação dos ficheiros, ou, por outras palavras, o utilizador tem de saber que o delimitador do ficheiro é «cfsend». Podíamos ter escrito outra sub-rotina, chamada *close* (fechar), que fizesse esse trabalho automaticamente. Deste modo, as linhas 60 e 70 do nosso programa de teste seriam substituídas por:

```
60  GO SUB close
```

A rotina *close* consistiria apenas em:

```
LET r$ = "cfsend"
GO SUB write
RETURN
```

«MICRODRIVE»

Se o leitor olhar para o teclado do Spectrum, vai encontrar as senhas CLOSE # e OPEN #, que são as instruções equivalentes às que tenho estado a analisar, mas para ficheiros contidos nos *microdrives.* (Uma vez que não existe CREATE #, OPEN # tem de fazer os dois trabalhos, como acontece com *initcfs.*) Os equivalentes de *read* e *write* são INPUT # e PRINT # . Tudo o mais (ou seja, a organização de blocos de ficheiro, etc.) está a cargo do sistema operador do Spectrum, de forma que a estrutura de ficheiros nos *microdrives* é, para o utilizador, tão transparente como cfs. Na realidade, as rotinas de manuseamento do *microdrive* têm de fazer muito mais do que aquilo que é feito por *read, write, getblock, putblock* e *initcfs,* mas os princípios são semelhantes.

CONTROLO AUTOMÁTICO

Eis uma pergunta que, provavelmente, se encontra há já algum tempo no espírito do leitor: «Será possível conseguir que o Spectrum controle automaticamente os motores das *cassettes?*»

A resposta é: «Sim.» De resto, nem sequer é muito difícil. Para o conseguir, o leitor precisa de gravadores que possuam bornes de conexão para controlo remoto (a maior parte está nestas condições). Além

disso, vai precisar de um porto I/O paralelo e de um par de *relais* de 5 V para correntes fracas. Os contactos dos *relais* são usados para completar os circuitos de controlo remoto dos motores, e as suas bobinas são conduzidas a partir de qualquer par de *bits* do porto. Seguidamente, em vez de fazermos imprimir mensagens, limitamo-nos a colocar (POKE) no porto uma configuração binária (utilizando BIN) que liga ou desliga a linha apropriada. O apêndice B fornece os pormenores de *hardware*. Infelizmente, a instrução SAVE continua a enviar automaticamente a sua indicação para o gravador ser ligado e espera que o utilizador carregue numa tecla. Por isso, a automatização completa mantém-se arreliadoramente fora do nosso alcance.

O USO DOS FICHEIROS

Talvez o leitor ainda se lembre de que a ideia impulsionadora deste esforço todo foi a de escrever um programa que tratasse dos pormenores da nossa colecção de discos. Contudo, essa ideia parece ter-se perdido na profusão de pormenores acerca das sub-rotinas para o tratamento do sistema de ficheiros. Seja como for, agora estamos de posse dessas sub-rotinas, e isso vai tornar muito mais fácil escrever o programa principal.

Qualquer sistema de ficheiro tem três funções básicas:

1. Criar o ficheiro.
2. Actualizá-lo através das adições e cortes necessários.
3. Procurar no ficheiro qualquer entrada que se queira.

Para ser mais fácil, vamos escrever para estas funções programas separados; no entanto, é fácil ligá-los usando um *menu* (consultar o nosso livro *Machine Code and Better Basic*).

A rotina de criação do ficheiro começa com bastante simplicidade. Há uma chamada a *initcfs,* na qual se estabelece que o nome do ficheiro de *input* vai ser «null» e o do ficheiro de *output* «reccol», por exemplo, que o número de *bytes* por registo vai ser 336 e o número de registos por bloco vai ser à volta de 5, para permitir que o programa disponha de um espaço adequado.

Chegamos agora ao único problema sério que nos é posto por esta rotina, e que consiste na forma de estabelecer cada registo de um modo que convenha ao utilizador. Por exemplo, sabemos que o campo para o «artista» tem um comprimento de 30 *bytes;* contudo, não queremos que o utilizador tenha de dactilografar «ABBA» seguido de 26 espaços. Por isso, vamos ter uma sub-rotina chamada *inrec* que trata do *input* de um único registo de uma maneira agradável para o utilizador.

O programa *create* (criar) tem então este aspecto:

```
100  GO SUB initcfs
110  GO SUB inrec
120  GO SUB write
130  INPUT "Any more? (y/n)"; q$
140  IF q$ = "y" THEN GO TO 110
150  GO SUB close          [partindo do princípio de que close
160  STOP                   foi implementada]
```

Vamos agora tratar de *inrec*. Estabeleçamos uma variável literal indexada, a$, que deve conter o registo à medida que ele vai sendo construído. Assim, se *inrec* começar na linha 8000, esta linha vai ser:

```
8000  DIM a$ (336)
```

Nesta altura é dada ao utilizador a indicação de que deve introduzir o primeiro dado requerido, que é colocado no local correcto:

```
8010  INPUT "Artist"; a$ (TO 30)
```

Seguidamente é a vez da data de compra:

```
8020  INPUT "Date of purchase"; a$ (31 TO 36)
```

Depois disto, queremos tratar de 12 pistas (Tracks) do mesmo modo:

```
8030  FOR t = 1 TO 12
8040  PRINT "track"; t
```

Tendo, na linha 8050, determinado os valores de *begin* (princípio) e *end* (fim), queremos escrever qualquer coisa do género de:

```
8060  INPUT "title"; a$ (begin TO end)
```

É evidente que *begin* e *end* mudam de acordo com a pista em que nos encontramos num dado momento. Vamos ver uma curta tabela dos seus valores, para ficarmos com uma ideia das relações envolvidas:

Pista (t)	Princípio	Fim
1	37	56
2	62	81
3	87	106

Posto isto, *begin* = 37 + 25 * (t − 1) e *end* = 56 + 25 * (t − 1).
Deste modo, temos:

```
8050  LET begin = 37 + 25 * (t - 1): LET end = 56 + 25 * (t - 1)
```

E, dado que a duração *(length)* de pista vai de *end* + 1 a *end* + 5, temos:

```
8070  INPUT "length"; a$ (end + 1 TO end + 5)
```

Seguidamente vem:

```
8080  NEXT t
```

O resultado é então passado para r$:

```
8090  LET r$ = a$
```

E chegamos ao fim da sub-rotina:

```
8100  RETURN
```

ACTUALIZAÇÃO

E quanto ao programa de actualização? Bom, começamos de novo com uma chamada a *initcfs,* onde, pela primeira vez, vamos querer definir tanto o ficheiro de *input* como o de *output*. O ficheiro de *input* chama-se «reccol», e é preciso que identifiquemos o ficheiro de *output* como sendo uma actualização deste. Podemos dar-lhe, por exemplo, o nome de «reccola», sendo as actualizações subsequentes chamadas «reccolb», «reccolc», etc., como se fossem matrículas de automóveis. Outra possibilidade, que pode agradar ao leitor, é fornecer informações acerca da data e chamar ao ficheiro, por exemplo, rc982, o que significaria «record collection as at September' 82» («colecção de discos em Setembro de 82»). Seja como for, vai ter de utilizar um sistema *qualquer;* de outra forma, é extremamente fácil enganar-se ao escolher o ficheiro e modificar a informação errada.

Para começar, vamos facilitar o problema, permitindo ao utilizador efectuar apenas uma adição ou um corte no ficheiro em cada passagem do programa.

Assim, vamos ter:

```
100  GO SUB initcfs
110  INPUT "add or delete (a/d)?"; q$
```

```
120  IF q$ = "a" THEN GO SUB add: STOP
130  IF q$ = "d" THEN GO SUB delete: STOP
140  PRINT "Enter a or d"
150  GO TO 110
```

CORTES

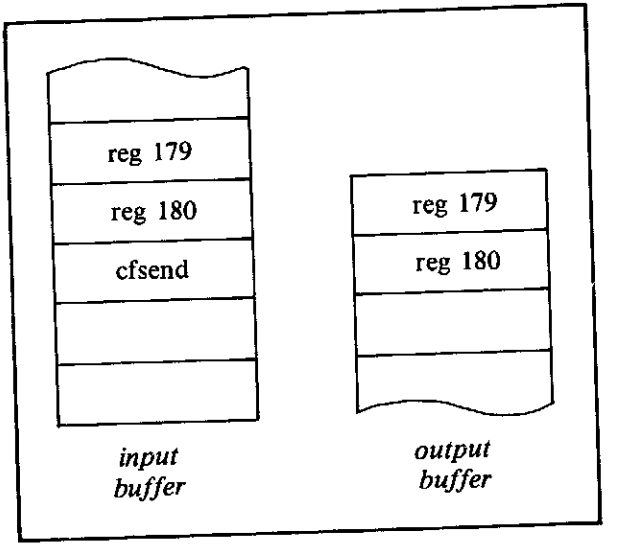
Vamos começar pela rotina *delete* (cortar), pois é mais fácil. Suponhamos que esta rotina deve encontrar-se a partir de 6000. Temos de dimensionar uma variável literal indexada para conter o nome do artista e a data de compra, de modo a identificar o disco. É claro que isto parte do princípio de que o leitor não comprou dois discos do mesmo autor no mesmo dia. Este tipo de ambiguidade causa muitas vezes problemas no planeamento de ficheiros. A maneira habitual de evitar isto consiste em acrescentar um campo suplementar, chamado uma *chave,* no início de cada registo, contendo esse campo um número que é usado apenas para esse registo específico. O número de uma conta bancária é um exemplo deste caso. Porém, partindo do princípio de que este problema não se vai pôr, precisamos do seguinte:

```
6000  DIM a$ (336)
6010  INPUT "Artist"; a$ (TO 30)
6020  INPUT "date purchased"; a$ (31 TO 36)
```

Estabelecemos assim a variável literal de que precisamos exactamente da mesma maneira que o fizemos na rotina *inrec* que acabámos de escrever. Até aqui, vai tudo muito bem; agora, só temos de ir buscar registos ao ficheiro de *input,* ver se coincidem com o registo que queremos cortar e, caso não coincidam, enviá-los para o ficheiro de *output.*

```
6030  GO SUB read
6040  IF r$ (TO 36) = a$ THEN GO TO 6030
6050  GO SUB write
6060  GO TO 6030
```

É fácil, não lhes parece? Infelizmente, é fácil de mais. Pensemos no que acontece perto do fim do ficheiro.



Suponhamos que acabámos de ler o registo 18Ø, vimos que não é para ser cortado, e o transferimos para o *output buffer* por intermédio da rotina *write*. É agora chamada *read* e encontra «cfsend». A rotina *read* diz que se ultrapassou o fim do ficheiro, e o programa pára com uma mensagem de erro. O leitor pode dizer que isso não constitui um verdadeiro problema, porque, de qualquer modo, já tínhamos acabado. Porém, isso não é bem verdade, dado que o *output buffer* ainda contém os registos 179 e 18Ø, que ainda não saíram. Além disso, como o nosso ficheiro de *output* não possui marcador de fim de ficheiro, é certo e sabido que vão acontecer coisas muito estranhas se tentarmos ler o ficheiro que acabámos de criar. Vem a propósito dizer que, quando este tipo de coisa acontece, é costume dizer-se que o *output buffer* não foi *flushed* (mangueirado), o que constitui um pitoresco termo da nossa gíria.

Há várias maneiras de evitar a armadilha em que acabámos — ou, se preferirem, acabei — de cair. A mais simples é, possivelmente, ter *dois* marcadores de fim de ficheiro, um para benefício do cfs e outro para benefício do leitor. Vamos usar "}}":

Vai ser, portanto, necessário voltarmos a escrever a rotina *close* para serem gerados ambos os marcadores de fim de ficheiro:

```
9740  LET r$ = "}}"
9750  GO SUB write
9760  LET r$ = "cfsend"
9770  GO SUB write
9780  RETURN
```

(É evidente que, por exigência do BASIC, *close* tem de ser identificada. Assim, podíamos acrescentar a linha 3

```
3 LET putblock = 9900: LET close = 9740        )
```

Agora, podemos modificar a rotina *delete* para testar o marcador de fim de ficheiro do utilizador e fechar o ficheiro de *output* quando esse marcador for encontrado:

```
6000 DIM a$ (336)
6010 INPUT "Artist"; a$ (1 TO 30)
6020 INPUT "date purchased"; a$ (31 TO 36)
6030 GO SUB read
6040 IF r$ (TO 2) = "}}" THEN GO SUB close: RETURN
6050 IF r$ (TO 36) = a$ THEN GO TO 6030
6060 GO SUB write
6070 GO TO 6030
```

Deve notar-se que, nas linhas 6040 e 6050, só são usados nas comparações os primeiros 2 e os primeiros 36 *bytes* de r$, respectivamente. Isto é importante, mas, se não for cuidadoso, vai esquecê-lo com facilidade. A razão para as comparações serem feitas desta forma está no facto de r$ ter um comprimento de 336 *bytes,* e, em termos de BASIC, ainda que a variável esteja vazia, "}}" não é o mesmo que "}} + 334 espaços".

ADIÇÕES

Tratemos agora da rotina de adições. Ainda não dissemos nada sobre a ordem em que os registos estão guardados na fita. De momento, partamos do princípio de que eles se encontram segundo a ordem alfabética do nome do artista, mas que, quando existir mais do que um disco do mesmo artista, a ordem é indefinida. Assim sendo, o nosso problema pode, em linhas gerais, ser apresentado da seguinte maneira:

1. Introduzir a adição.
2. Ler um registo.
3. Se for o fim do ficheiro, fechar o ficheiro: RETURN.
4. Se o nome do artista (registo) estiver antes do nome do artista (adição), escrever registo: saltar para 2.
5. Escrever adição.
6. Escrever registo.
7. Saltar para 2.

Por outras palavras, fazemos exactamente o que faríamos com um ficheiro: arranjamos um novo cartão, percorremos o ficheiro até encontrarmos o nicho alfabético correcto e colocamo-lo aí. A única diferença está em que, no caso do computador, os registos são *fisicamente* deslocados de um local para outro (*input* para *output*) de cada vez que os lemos. É como se tivéssemos duas caixas de ficheiro e, assim que acabássemos de ler um cartão da caixa de «*input*», tivéssemos de o transferir para a caixa de «*output*».

No entanto, há um subtil senão no nosso algoritmo. Suponhamos que o último disco da nossa colecção é da Suzi Quatro, e queremos acrescentar um álbum dos Yardbirds. O programa trabalha através do ficheiro, transferindo os registos à medida que vai passando, tratando finalmente do LP da Suzi Quatro. A seguir, vendo um marcador de fim de ficheiro, fecha o ficheiro, deixando os pormenores a serem adicionados ainda na memória! Assim sendo, precisamos de modificar o passo 3:

3. Se for o fim do ficheiro, verificar: fechar o ficheiro: acabar.

A verificação vai ser feita por uma rotina, *check,* que observa se o registo adicional já foi escrito, e, se não tiver sido, o escreve.

Vamos escrever a rotina *add* (adicionar) a partir da linha 5000. Antes de mais nada, temos de introduzir o registo adicional. Como já temos uma rotina que aceita todos os pormenores respeitantes a um disco, a rotina *inrec,* vamos chamá-la. Não tem interesse nenhum voltarmos a inventar a roda...

```
5000  GO SUB inrec: LET transfer = 0
5010  GO SUB read
5020  IF r$ (TO 2) = "}}" THEN GO SUB check: GO SUB close:
      RETURN
5030  IF r$ (TO 30) < a$ (TO 30) THEN GO SUB write: GO TO 5010
```

É necessário explicar um par de coisas antes de prosseguirmos. A primeira delas é a instrução "LET transfer = 0", na linha 5000. A rotina *check* vai precisar de ter uma maneira de saber se o registo adicional já foi transferido para o ficheiro de *output* ou não. Assim que a adição tenha sido aceite, «transfer» (transferir) é posta a zero. Quando a adição for transferida para o ficheiro de *output,* é atribuído a «transfer» o valor 1. Em consequência disto, é suficiente que *check* verifique se o valor de «transfer» é zero. Se for, quer dizer que a adição ainda tem de ser colocada no ficheiro de *output.*

A segunda coisa que precisa de ser explicada é a comparação, na linha 5030, de duas variáveis literais: r$, que acaba de ser devolvida

por *read,* e a$, que foi estabelecida por *inrec.* (r$ também foi estabelecida por *inrec,* mas foi imediatamente alterada por *read.*) O símbolo «menor que», usado nestas circunstâncias, tem o significado «precede alfabeticamente», de modo que ordenar alfabeticamente variáveis literais não apresenta qualquer problema.

Mas, continuando, temos:

```
5040  LET b$ = r$          [guardar o último registo lido]
5050  LET r$ = a$          [transferir a adição para r$ ...
5060  GO SUB write          para ser escrita... e escrevê-la]
5070  LET transfer = 1      [indicar que já foi escrita]
5080  LET r$ = b$
5090  GO SUB write          [voltar a pôr o último registo
5100  GO TO 5010            em r$ ... e escrevê-lo]
```

Vamos agora, finalmente, escrever *check* (a partir da linha 5200):

```
5200  IF transfer = 1 THEN RETURN  [não é preciso agir porque
                                    a adição já foi output]
5210  LET r$ = a$          [transferir a adição para r$
5220  GO SUB write          e escrevê-la]
5230  RETURN
```

[1] Trocadilho intraduzível, baseado na semelhança existente entre as palavras inglesas «files» (ficheiros) e «flies» (moscas). *(N. da T.)*

Projecto

Tudo aquilo que temos estado a fazer parte do princípio de que apenas vai ser feita uma alteração no ficheiro. Se, num dia de euforia,

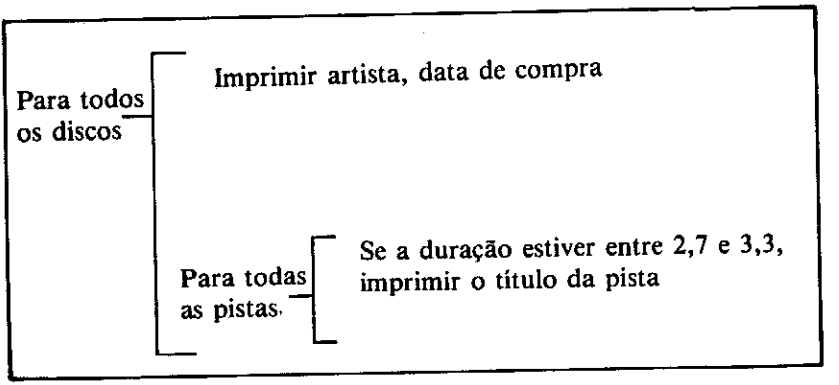
lhe der para comprar quinze discos novos e doar quatro dos seus discos antigos à Oxfam, vai precisar de passar o programa de actualização de ficheiros dezanove vezes para actualizar o ficheiro da sua colecção!

Tente escrever um programa de actualização que aceite, no início, todas as alterações. Vai ter de ler todas as adições para uma variável literal indexada e todos os cortes para outra. Em seguida, vai ter de comparar cada registo lido com cada um dos elementos das duas variáveis, para decidir se deve ser cortado, copiado ou se deve ser feita alguma inserção. Deve notar-se que cada adição vai precisar da sua própria *flag* de transferência (ou seja, vai ser necessário usar uma *variável indexada* de transferência em vez de uma variável simples). Deve também notar-se que a ordem de entrada das adições é irrelevante, pois o programa procura na variável indexada qualquer adição que deva ser inserida. Não se esqueça de contar com a possibilidade de ter de ser feita mais do que uma adição entre dois discos previamente existentes. Uma ideia interessante é criar o ficheiro apenas com um par de registos e, seguidamente, utilizar este programa de actualização para acrescentar os outros, de modo a que o ficheiro seja automaticamente gerado por ordem alfabética!

A PROCURA DE REGISTOS NO ARQUIVO

Já que gastámos uma quantidade de tempo considerável na criação deste ficheiro, devíamos encontrar alguma utilidade para ele. Suponhamos que queremos arranjar uma fita de música de fundo para uma festa. Queremos um conjunto de pistas com uma duração de cerca de 3 minutos. Assim, gostaríamos de ter uma lista das pistas cuja duração esteja entre 2,7 e 3,3 minutos.

Queremos portanto um programa que efectue as seguintes operações:



Isto é bastante fácil de conseguir:

```
100 GO SUB initcfs
110 GO SUB read
120 IF r$ (TO 2) = "}}" THEN STOP
130 PRINT r$ (TO 30)
140 PRINT r$ (31 TO 36)
150 FOR t = 1 TO 12
160 LET begin = 57 + (t - 1) * 25
170 LET d = VAL r$ (begin TO begin + 4)
180 IF d < 2.7 OR d > 3.3 THEN GO TO 200
190 PRINT r$ (begin -20 TO begin -1)
200 NEXT t
210 GO TO 110
```

Deve notar-se que, uma vez que estamos apenas a *ler* um ficheiro, não é necessária uma operação de *close* quando o fim do ficheiro é detectado, na linha 120.

Neste programa, a única parte que pode ser considerada um pouco traiçoeira é a avaliação da duração de cada pista como um número. Primeiro, temos de encontrar a porção de r$ que é relevante (linha 160). Seguidamente, é preciso criar um valor numérico a partir da variável literal (linha 170), de forma que seja possível efectuar comparações entre números na linha 180. É evidente que o 2,7 e o 3,3 podiam com igual facilidade ser variáveis cujo valor fosse introduzido no início do programa. Isso permitiria que fossem feitos outros pedidos, como, por exemplo:

> «Fazer uma lista de todas as pistas com uma duração mínima de 4,5 minutos» (dando como limite mínimo 4,5 e como limite máximo um valor despropositadamente alto, como por exemplo, 9999).

Ou então:

> «Fazer uma lista de todas as pistas cuja duração seja exactamente 3 minutos» (dando a ambos os limites o valor 3).

Projectos

Também é fácil identificar outras características específicas do ficheiro. Tente com as seguintes:

1. Fazer uma lista de todas as pistas interpretadas por um determinado artista.
2. O mesmo que (1), mas para discos comprados entre duas datas determinadas. (É ligeiramente mais difícil do que parece, por causa das comparações entre datas.)
3. Fazer uma lista de todas as pistas cujo título inclua a palavra «*rock*», ou qualquer outra palavra introduzida por meio de uma declaração INPUT.

REGISTOS DE COMPRIMENTO VARIÁVEL

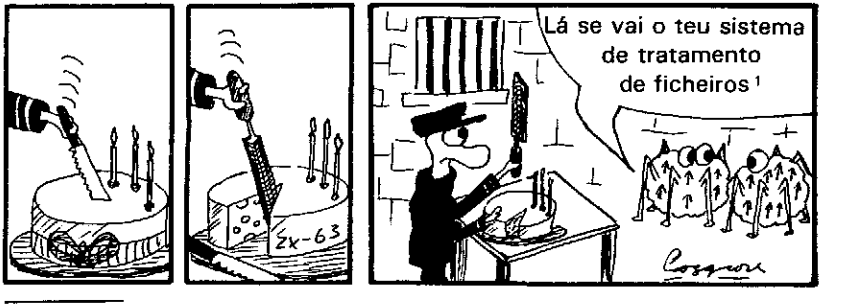
Foi dito no princípio deste capítulo que íamos partir do princípio de que todos os discos tinham 12 pistas, que cada título de pista ocupava exactamente 20 *bytes*, e por aí fora. Assim, criei um ficheiro cujos registos têm de certeza um comprimento de 336 *bytes*. Duvido muito que o leitor fique surpreendido ao saber que se diz que um ficheiro deste tipo tem registos de *comprimento fixo*. Além disso, cada registo contém campos de comprimento fixo, de forma que, por exemplo, cada título de pista tem de ser completado com espaços (ou abreviado) para preencher exactamente 20 *bytes*. Em circunstâncias ideais, era agradável permitir que os campos tivessem comprimentos variáveis (terminando cada um deles com um delimitador de campo qualquer) e os registos também (terminando com outro delimitador). Dessa forma, não desperdiçaríamos 27 *bytes* com cada álbum dos «Yes», nem haveria 8 pistas nulas para cada sinfonia clássica (se tratássemos cada movimento como sendo uma pista).

No entanto, o que se ganha por um lado perde-se pelo outro. Em primeiro lugar, os programas para extrair campos de um registo tornam-se mais complicados. Temos de verificar todos os *bytes*, procurando os delimitadores de campo, e já não podemos falar do «quinto campo», pois pode não existir. Em segundo lugar, as rotinas utilitárias para ler e escrever e para o tratamento de blocos de ficheiro tornam-se bastante rebarbativas. No fim de contas, com o nossos sistema fixo, era fácil definir eficazmente um bloco como uma variável indexada bidimensional — número de *bytes* por registo × número de registos por bloco. No caso de o número de *bytes* por registo mudar, já não podemos pensar desta maneira, que dá tanto jeito. Por outro lado, se mantivermos fixo o tamanho dos *buffers*, o número de registos por bloco muda, de acordo com o tamanho dos registos!

Não é minha intenção dar a ideia de que o tratamento de registos de comprimento variável excede as capacidades humanas, mas apenas de que usá-los pode não ser tão sensato como à primeira vista parece. A razão disso reside no facto de o tamanho dos programas utilitários aumentar à medida que estes se complexificam, o que, correspondentemente, diminui a quantidade de memória principal que sobra para o programa do utilizador e os *buffers* de ficheiros. Além disso, a não ser que esteja a ser desperdiçado muito espaço com o formato de comprimento fixo, a introdução de delimitadores extra pode gastar a maior parte do espaço de ficheiro que se tinha poupado. É frequente que a escolha criteriosa de comprimentos de campo num sistema de comprimento fixo seja perfeitamente adequada às necessidades. Devemos pensar cuidadosamente nos conteúdos do ficheiro. Será que queremos realmente misturar no mesmo ficheiro discos de música clássica e de música *pop?* Não seria melhor termos dois ficheiros, de modo a cada um deles poder ser definido de uma maneira sensata, sem ter de tomar em conta as características do outro?

Seja como for, não nos devemos esquecer nunca de que estamos a trocar espaço de *cassette,* que é barato, por RAM, que o não é. De facto, 5 *Kilobytes* (KB) de dados custam, mais ou menos 1 péni em fita de *cassette.* (É fácil de calcular: basta lembrar-se de que, a uma velocidade de transferência de cerca de 200 *bytes* por segundo, leva 25 segundos a guardar 5 KB, e seguidamente calcular 25 ÷ (60 × 60) do preço que tiver pago pela sua última *cassette* de 60 minutos. É provável que chegue à conclusão de que o valor que apresentei acima é pessimista.) No entanto, mesmo nos nossos dias, em que os preços de memória descem vertiginosamente, 5 KB em memória principal vão-lhe custar cerca de 10 libras. Creio que este facto é incontestável.

Se os raciocínios desenvolvidos acima parecem ao leitor os rodeios plausíveis de alguém que não quer ter de pensar demasiado a não ser que seja de facto obrigado a isso, significa que percebeu exactamente onde eu queria chegar. Em princípio, os computadores devem servir para tornar a vida mais fácil, não para a complicar.



[1] Trocadilho intraduzível, baseado no facto de a palavra inglesa «file» significar, simultaneamente, «ficheiro» e «lima». *(N. da T.)*

Uma modesta proposta

Tal como está, o cfs tem uma característica enervante: *initcfs* pede ao utilizador pormenores sobre o tamanho dos registos e dos blocos, mesmo para ficheiros que já existem.

Modifique cfs de modo que, precedendo cada ficheiro de *output*, seja escrito um bloco «cabeçalho» contendo os pormenores de tamanho de registos e de blocos. Seguidamente, se for especificado para *initcfs* o nome de um ficheiro de *input*, essa rotina já não vai pedir os pormenores de tamanho do *buffer*, lendo-os em vez disso no bloco cabeçalho do ficheiro de *input*.

(Se quiser uma proposta *menos* modesta, consulte o capítulo 17.)

Há mentiras, mentiras grandes, mentiras enormes
e resultados de computador...

10

A ESTATÍSTICA SEM ESFORÇO

A National Wealth Servers Ltd. (NWS) pagava aos seus 24 empregados 5Ø libras por semana; simultaneamente, o seu director recebia 1Ø4 ØØØ libras por ano. Quando a organização sindical correspondente, a Regional Organization of Wealth Serving Employees (ROWSE), entrou em greve com vista ao aumento dos salários, a gerência reservou uma página de anúncios inteira no *Gardener,* sendo aí o público informado de que o salário *médio* nessa companhia era de 128 libras por semana. Segundo as palavras do director, aqueles preguiçosos deviam voltar ao trabalho imediatamente e acabar com a agitação, pois 128 libras eram um salário semanal perfeitamente justo.

Há quem use a estatística, mas também há quem abuse dela. As médias *podem,* por vezes, ser uma medida justa da «norma», o «valor típico»; contudo, também podem ser distorcidas por uma excepção absolutamente discrepante, como acontece no caso presente. O *cálculo* está correcto:

$$\text{Média} = \frac{\text{Total dos salários de uma semana}}{\text{Número de empregados}}$$

$$= \frac{24 \times 5Ø + 2ØØØ}{25}$$

$$= 24 \times 2 + 8Ø = 128$$

No entanto, a *interpretação* segundo a qual a maioria dos trabalhadores ganha cerca de 128 libras por semana está errada.

Este episódio mostra bem que vale a pena ter alguns conhecimentos básicos de estatística. Neste caso, a ROWSE publicou, por sua vez, um anúncio dizendo que a *moda,* o valor do salário mais comum — aqui 5Ø libras —, era uma medida mais apropriada. Reconhecendo a lógica deste argumento, a NWS despediu o director, entregando a gestão a uma comissão de empregados, e subiu os salários para 128 libras por semana, poupando assim 4Ø56 libras por ano, o que, nos dez anos seguintes, quase amortizou o custo da campanha publicitária original.

Mas é claro que as coisas nem sempre se passam desta maneira...

APRESENTAÇÃO DE DADOS: HISTOGRAMAS

Este livro não é o local apropriado para ensinar teoria estatística. Vou apenas escrever alguns programas que permitam ao leitor explorar ideias estatísticas sem que tenha de se preocupar com o modo de funcionamento. Assim, ser-lhe-á possível compreender o que elas representam na prática.

Uma parte importante da estatística diz respeito à forma de apresentação de dados. Como o Spectrum é um animal visual, vou concentrar-me em dois tipos-padrão de apresentação gráfica: o *histograma* e a *tabela circular*.

Um histograma mostra a frequência com que ocorreu um dado «acontecimento». Suponhamos, por exemplo, que deito um dado vinte e oito vezes, obtendo os seguintes resultados:

o 1 saiu 3 vezes
o 2 saiu 6 vezes
o 3 saiu 5 vezes
o 4 saiu 5 vezes
o 6 saiu 4 vezes
o 6 saiu 5 vezes

Um histograma que represente estes resultados será constituído por seis barras verticais, referenciadas de 1 a 6, tendo cada uma delas uma altura correspondente à frequência com que o número correspondente saiu. Assim, a barra 1 tem altura 3, a barra 2 tem altura 6, e por aí fora, como se vê na figura 10.1.

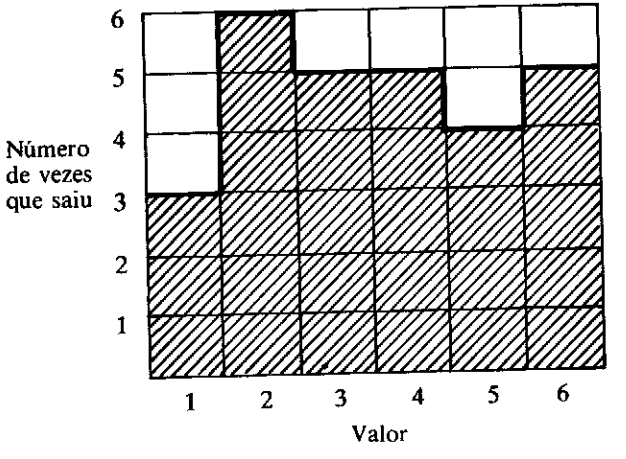


Fig. 10.1 — *Histograma mostrando o número de vezes que um determinado valor ocorreu ao ser lançado um dado.*

A *média* ou valor médio deste lançamento é dada por

$$\frac{1\times3+2\times6+3\times5+4\times5+5\times4+6\times5}{28}=3{,}57$$

Deve notar-se que cada valor é multiplicado pelo tamanho da barra correspondente. Se um dado que não esteja viciado for lançado um número de vezes suficientemente grande, deve verificar-se um número de ocorrência aproximadamente igual para cada um dos valres 1-6, sendo o valor médio

$$\frac{1\times2+3\times4+5\times6}{6}=3{,}5$$

Assim sendo, esta estatística particular está bastante de acordo com a teoria. Deve, no entanto, notar-se que o valor mais comum nesta experiência — a *moda* — é o número 2, que ocorre seis vezes.

A moda diz-nos onde se encontra o ponto mais alto do histograma. A média indica-nos mais ou menos onde desenhar uma linha vertical que divida ao meio a área total. A média e a moda não coincidem, necessariamente. A média só dá uma ideia do valor «típico» se os valores não forem demasiado dispersos; neste caso, são *muito* dispersos.

Mais tarde vou falar da maneira de medir o grau de dispersão, mas de momento apenas quero deixar claro o que é um histograma. Para que o leitor se familiarize com esta noção, aqui está um programa que apresenta um histograma para lançamentos de um dado, sendo o resultado de cada lançamento introduzido à medida que o dado vai sendo lançado. Além disso, o programa informa ainda sobre o valor médio dos lançamentos a cada momento.

```
10  LET init = 500
20  LET valin = 1000
30  LET hist = 1500
40  LET mean = 2000
50  LET wtot = 0: LET num = 0
100 DIM d (6)
200 GO SUB init
210 GO SUB valin
220 GO SUB hist
230 GO SUB mean
240 GO TO 210
500 REM init
510 PLOT 103, 175: DRAW 0, -160: DRAW 48, 0
```

```
520 PRINT AT 21, 13; "123456"
530 FOR i = 1 TO 6
540 PLOT 99 + 8 * i, 15: DRAW 0, -5
550 NEXT i
560 FOR i = 0 TO 20
570 PLOT 103, 15 + 8 * i: DRAW -5 -5 * (i = 10 OR i = 20), 0
580 NEXT i
590 PRINT AT 0, 9; "20"; AT 10, 9; "10"
600 RETURN
1000 REM valin
1010 IF INKEY$ < > " " THEN GO TO 1010
1020 IF INKEY$ = " " THEN GO TO 1020
1030 LET c = CODE INKEY$ - 48
1040 IF c < 0 OR c > 6 THEN GO TO valin
1050 IF c = 0 THEN STOP
1060 LET d (c) = d (c) + 1
1070 RETURN
1500 REM hist
1510 IF d (c) = 21 THEN INPUT "No room for display"; x$: STOP
1520 PRINT AT 20 - d (c), c + 12; PAPER c; "□"
1530 RETURN
2000 REM mean
2010 LET num = num + 1: LET wtot = wtot + c
2020 LET a = .01 * INT (100 * wtot/num)
2030 PRINT AT 2, 22; "Mean ="
2040 PRINT AT 4, 23; a; "□ □ □"
2050 RETURN
```

Dactilografe esta listagem no computador e RUN o programa. Lance um dado e prima a tecla correspondente ao resultado, ou seja, se saiu um «4», por exemplo, carregue na tecla 4, procedendo de forma idêntica para os outros números; não é necessário usar ENTER. No visor vai sendo construído um histograma colorido, que mostra, para cada um dos números 1-6, quantas foram as vezes é que esse número saiu. À medida que vão sendo registados mais lançamentos, o valor da média é actualizado. No caso de um número sair mais do que 20 vezes, o programa transmite uma mensagem e pára (na realidade, é preci-

so premir uma tecla para que ele pare). Para terminar antes de este caso se ter verificado, carregue na tecla Ø.

O leitor não vai ter dificuldade para compreender o funcionamento deste programa. O programa principal encontra-se nas linhas 2ØØ--24Ø; *init* estabelece os eixos e as escalas; *valin* lê a tecla; *hist* marca o gráfico; *mean* calcula a média e imprime-a.

O uso de INT na linha 2Ø2Ø serve apenas para nos certificarmos de que só vão ser impressas duas casas decimais (ou menos). É uma artimanha útil. (Para obter três casas decimais, use LET a = .ØØ1 * INT(1ØØØ*wtot/num), e assim sucessivamente, aumentando um zero em ambos os lados por cada casa decimal suplementar requerida.)

APRESENTAÇÃO DE DADOS: TABELAS CIRCULARES

Estes gráficos mostram bem como é que o bolo se encontra dividido entre os vários beneficiários: um círculo é dividido em fatias, sendo estas tanto mais grossas quanto maior for a parcela que representam. O leitor sabe com certeza a que é que me estou a referir.

O programa que se segue é um divisor automático de tabelas circulares: aceita como *input* uma série de itens (de despesas, por exemplo), cada um com seu nome, e apresenta a tabela circular correspondente. É mais eficiente com um número de itens igual ou inferior a 1Ø, sendo preferível que nenhum dos itens corresponda a menos do que 3% do total. O programa funciona mesmo que estes critérios não sejam respeitados; contudo, as tabelas circulares não são muito úteis quando são compostas por dimensionadas fatias, ou quando estas são tão finas que não se conseguem ver.

```
2ØØ  REM data in
21Ø  INPUT "Number of items?"; n
22Ø  DIM i$ (n, 9): DIM v (n)
23Ø  DIM a (n + 1)
24Ø  FOR i = 1 TO n
25Ø  INPUT "Name of item?"; i$ (i)
26Ø  INPUT "Value of item?"; v (i)
27Ø  PRINT AT i, 5; i$ (i), v (i); "□ □ □"
28Ø  INPUT "Is this correct? y/n"; q$
29Ø  IF q$ = "n" THEN GO TO 25Ø
3ØØ  LPRINT i$ (i), v (i): REM only type this line if you have a printer
31Ø  NEXT i
32Ø  LET tot = Ø
```

```
330  FOR i = 1 TO n
340  LET tot = tot + v (i)
350  NEXT i
500  REM piechart
510  CLS: CIRCLE 84, 84, 75
520  LET ang = 0
530  FOR i = 1 TO n
540  LET ang = ang + v (i) * 2 * PI / tot
550  LET a (i + 1) = ang
560  PLOT 84, 84
570  DRAW 75 * COS ang, 75 * SIN ang
600  REM label
610  LET u = .5 * (a (i) + a (i + 1) )
620  PRINT AT 11 - 7 * SIN u, 10 + 7 * COS u; i
630  NEXT i
700  REM table
710  FOR i = 1 TO n
720  PRINT AT i, 21; i; ":"; i$ (i)
730  NEXT i
740  COPY: REM if you have a printer
```

As linhas 300 e 740 devem ser omitidas, excepto se o leitor tiver uma impressora e quiser ter um registo dos resultados. As linhas 280 e 290 facultam ao utilizador uma maneira de corrigir erros se der por eles imediatamente; no entanto, se esta característica o aborrecer, pode retirar estas linhas. (A propósito, para «sim» pode premir qualquer tecla menos o «n». A tecla óbvia é ENTER.)

Por exemplo, introduza os números do quadro seguinte, que dão o produto nacional bruto *(per capita)* dos países da CEE em 1978.

Número de itens: 9 Nome do item	Valor do item	s/h
Bélgica	129	n para corrigir
Dinamarca	144	»
França	116	»
Alemanha	137	»
Holânda	123	»
Irlanda	50	»
Itália	60	»
Luxemburgo	126	»
Reino Unido	73	»

A figura 10.2 mostra a tabela circular resultante deste conjunto de dados. Experimente utilizar outros conjuntos de números, quer reais, quer imaginários.

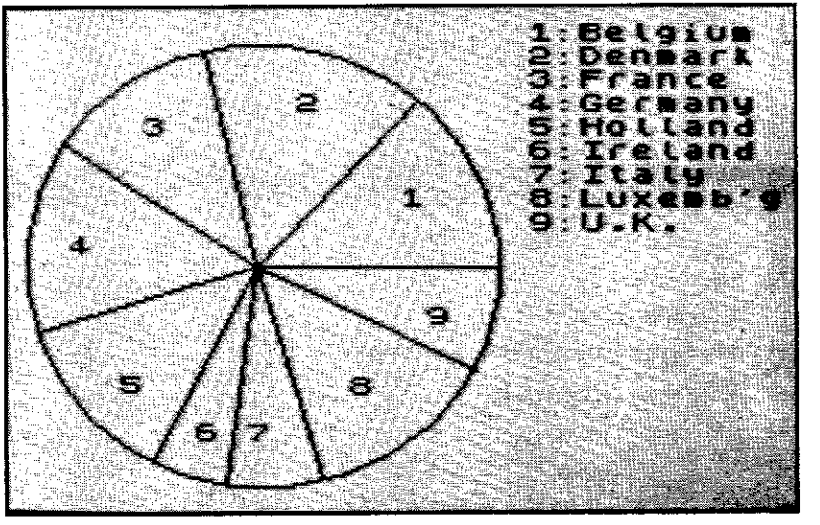


Fig. 10.2 — *Eis como o bolo da CEE está dividido...*

MÉDIAS, MODAS E POR AÍ FORA

Já expliquei o que são médias e modas (existe ainda outra criatura, chamada *mediana,* mas não lancemos a confusão). A média é o valor «médio», a moda é um (pode haver mais do que um) valor «mais comum».

Também já mencionei que a média é um valor razoavelmente «típico» desde que os dados não sejam demasiado dispersos. Para medir a dispersão, os estatísticos usam o *desvio-padrão.* Se quer saber como se calcula, é a raiz quadrada da média dos quadrados do desvio da média.

Os estatísticos têm também uma curva favorita, chamada *curva normal,* que é usada para aproximar histogramas. Essa curva tem sempre a mesma média e o mesmo desvio-padrão e tem a forma de uma bossa de camelo.

Em vez de presentear o leitor com uma data de matemática (se estiver interessado, tem sempre a possibilidade de consultar um livro de estatística — qualquer serve), resolvi escrever um programa que, a partir de um histograma previamente definido pelo utilizador, lhe dá a

média, a moda mais baixa, o desvio-padrão e ainda, como bonificação, marca a aproximação à curva normal.

Se o leitor (ou um filho seu) estiver a estudar estatística na escola, este programa vai ajudá-lo a compreender bem o que estes conceitos representam.

```
10  LET init = 500
20  LET draw = 1000
30  LET stats = 2000
40  LET normal = 3000
50  DIM a (24)
60  LET step = 3
100 GO SUB init
110 GO SUB draw
120 GO SUB stats
130 GO SUB normal
140 STOP
500 REM init
510 CLS
520 PLOT 15, 165
530 DRAW 0, -150
540 DRAW 240, 0
550 FOR t = 1 TO 24
560 PLOT 15 + 10 * t, 15: DRAW 0, -3 -3 * (t = 10 OR t = 20)
570 NEXT t
580 FOR t = 1 TO 15
590 PLOT 15, 15 + 10 * t: DRAW -3 -3 * (t = 10), 0
600 NEXT t
610 PRINT OVER 1; AT 21, 0; "[13 spaces] 10 [11 spaces] 20"
620 PRINT AT 7, 0; "1"
630 PRINT AT 8, 0; "0"
640 RETURN
1000 REM draw
1005 LET h = 15
1010 FOR i = 1 TO 24
1020 IF INKEY$ < > " " THEN GO TO 1020
```

```
1030  IF INKEY$ = " " THEN GO TO 1030
1040  LET k$ = INKEY$
1045  LET h = h + 10 * (CODE k$ - 53)
1050  IF h < 15 THEN LET h = 15
1055  IF h > 165 THEN LET h = 165
1060  LET a (i) = (h - 15) / 10
1070  PLOT 15 + 10 * i - 10, 15: DRAW 0, h - 15:
      DRAW 10, 0: DRAW 0, 15 - h
1080  NEXT i
1090  RETURN
2000  REM stats
2005  LET tot = 0: LET norm = 0: LET m = 0: LET mv = 0
2010  FOR i = 1 TO 24
2015  IF a (i) > mv THEN LET mv = a (i): LET m = i
2020  LET tot = tot + i * a (i)
2025  LET norm = norm + a (i)
2030  NEXT i
2040  LET av = tot/norm
2050  PRINT AT 0, 3; "Mean = □"; av
2055  PRINT AT 1, 3; "Mode = □"; m
2060  LET std = 0
2070  FOR i = 1 TO 24
2080  LET std = std + a (i) * (i - av) * (i - av)
2090  NEXT i
2100  LET std = SQR (std/norm)
2105  LET std = .01 * INT (100 * std)
2110  PRINT AT 2, 3; "Standard deviation = □"; std
2120  RETURN
3000  REM normal
3010  FOR i = 0 TO 240 STEP step
3020  LET y = EXP (- (.5 + i / 10 - av) * (.5 + i / 10 - av) /
      (2 * std * std) ) / (std * SQR (2 * PI) )
3030  LET y = y * tot
3035  IF y > = 150 THEN GO TO 3060
3040  IF i = 0 THEN PLOT i + 15, y + 15
```

```
3050 IF i > 0 THEN DRAW step, y + 15 - PEEK 23678
3060 NEXT i
3070 RETURN
```

O USO DO PROGRAMA ANTERIOR

Este programa foi planeado de modo a tornar muito simples o estabelecimento de histogramas experimentais. Em consequência disso, a forma como os histogramas são introduzidos é diabolicamente pouco ortodoxa e extremamente irritante até nos habituarmos a ela. Vejamos, pois, como se processa. As barras nas posições 1-24 são introduzidas sequencialmente. Inicialmente, a altura é 0 (e mantém-se sempre entre 0 e 16). A altura de barra *seguinte* é determinado pelo número k da tecla da carreira de cima em que carregarmos e vai ser igual à *soma* de k-5 com o anterior valor da altura. Isto quer dizer que a *mudança* de altura da barra entre uma coluna e a seguinte é controlada pelas teclas dos números de acordo com o quadro abaixo.

Tecla	Efeito sobre a altura da barra
1	4 mais baixa.
2	3 mais baixa.
3	2 mais baixa.
4	1 mais baixa.
5	Igual.
6	1 mais alta.
7	2 mais alta.
8	3 mais alta.
9	4 mais alta.

Por exemplo, para obter o resultado que se encontra na figura 10.3, RUN o programa e depois carregue (não precisa de usar ENTER) nas teclas

5 5 6 6 6 7 8 5 2 3 4 4 4 5 5 5 5 5 5 5 5 5 5

Invente os seus próprios histogramas e verifique que:

1. A média é um valor «médio» razoável. Se cortasse um histograma em folha de metal, ele equilibrar-se-ia quando apoiado sobre a linda da média.

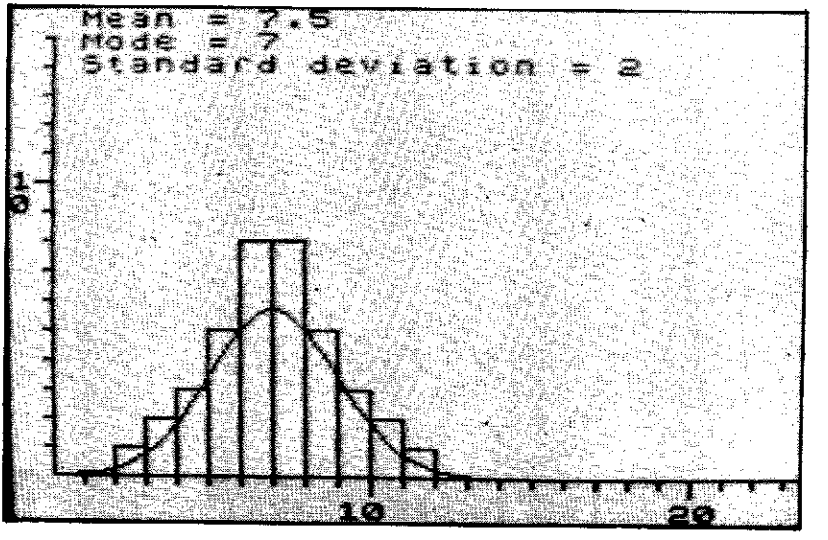


Fig. 10.3 — *Curva normal aproximando relativamente bem o histograma...*

2. A moda é o valor do primeiro «pico». (Os valores dos outros pontos que se encontram à mesma altura também são modas, mas o programa ignora-os. No entanto, é fácil alterar esta característica.)
3. Os histogramas que se encontram mais dispersos, como o que se obtém carregando nas teçlas

   5 5 6 6 6 7 8 5 5 5 5 5 2 3 4 4 4 5 5 5 5 5 5

   têm um desvio-padrão maior; os que são mais compactos, como o que se obtém carregando nas teclas

   5 5 5 5 5 5 9 9 1 1 5 5 5 5 5 5 5 5 5 5 5 5 5

   têm um desvio-padrão menor.
4. A curva é uma aproximação razoável para histogramas que tenham apenas um pico, não se encontrem demasiado dispersos e sejam mais ou menos simétricos...

Contudo, a aproximação à normal pode ser bastante inadequada para outros histogramas, por exemplo no caso de terem duas bossas, como acontece com o que se obtém carregando nas teclas

5 5 5 7 7 7 9 1 1 4 4 5 5 5 8 8 3 3 3 5 5 5 5 5

que dá o resultado que se vê na figura 10.4.

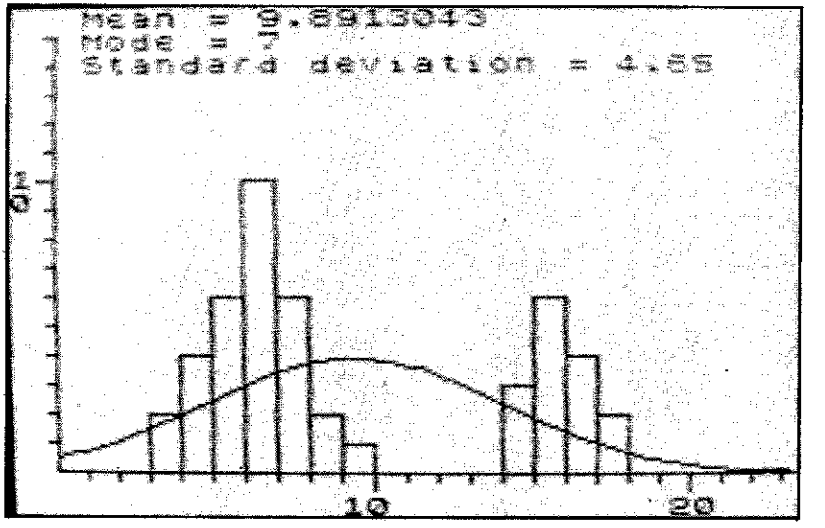


Fig. 10.4 — *...porém, às vezes, a aproximação é muito má.*

Projecto

Modifique as linhas 1045-1060, de modo a poder introduzir *directamente* a altura das barras. (Não é nada difícil de conseguir. A única razão por que não utilizei esse processo é que, tal como está, o programa apresenta os histogramas com muita rapidez e, desde que se esteja habituado ao método, sem ser necessário pensar muito, o que o torna ideal para experimentação.)

Experiência

Lance quatro dados e tome nota do valor total obtido. Execute esta operação cem vezes. Introduza o histograma resultante no programa de que temos estado a tratar (através do projecto) e veja quais são os resultados obtidos. A curva normal é uma é uma boa aproximação ou não?

Simulação

Utilize os números estocásticos do Spectrum para simular o lançamento de quatro dados e repita a análise anterior com os resultados assim obtidos.

Um pouco de atenção dispensada aos pormenores pode fazer milagres — como, por exemplo, traduzir Spectrês para álgebra...

11

MELHORAMENTO DA APRESENTAÇÃO

Há muitos programa que podem ser tornados muito mais atraentes através do uso de caracteres definidos pelo utilizador visando obter uma representação mais precisa do efeito desejado. Ao longo destas linhas vou considerar um projecto que pega em *polinómios,* em várias variáveis x, y e z, na forma que é utilizada pelo Spectrum e lhes dá o aspecto de álgebra normal. (Se o leitor não gostar de álgebra, tente ter paciência: o que é mais importante neste caso é a computação.)

Em álgebra normal, os polinómios são escritos da seguinte forma:

$ax^2 + bx + c$

$2x^3 + 5y^7$

$a^3 + b^3 + c^3 - 3abc$

O Spectrum, contudo, utiliza "*" para multiplicação, em vez de justapor simplesmente os símbolos, e utiliza "↑" para potenciação, em vez de elevar o símbolo em relação à linha, como podemos ver abaixo:

```
a*x↑2+b*x+c
2*x↑3+5*y↑7
a↑3+b↑3+c↑3-3*a*b*c
```

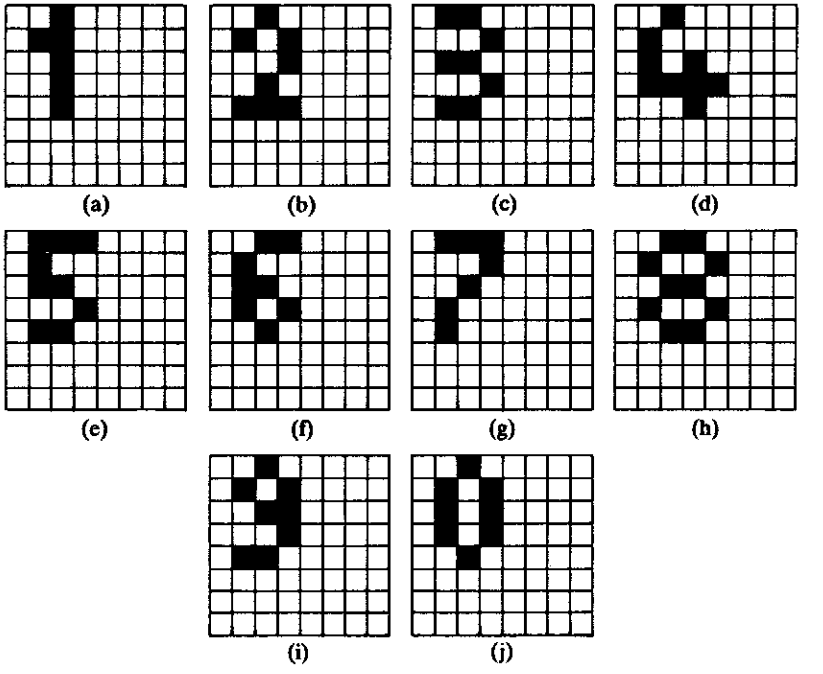


Fig. 11.1 — *Dados gráficos para os caracteres representando os expoentes.*

O primeiro passo consiste em desenvolver caracteres definidos pelo utilizador para representarem os expoentes elevados, 1, 2, ... 9, Ø. A figura 11.1 mostra um arranjo possível. Estes caracteres devem ser introduzidos como sendo os caracteres gráficos "a" – "j", da forma habitual (*Easy Programming*, p. 49).

A seguir, dactilografe o programa abaixo. Este programa aceita a expressão em Spectrês e percorre-o efectuando três tarefas:

1. Retirar todos os *.
2. Substituir os *números* que se seguem a ↑ por expoentes gráficos definidos pelo utilizador.
3. Retirar todas as ↑ quando (2) tiver sido executado.

```
 5 LET pow = 2ØØ
 6 LET prod = 6ØØ
 7 LET print = 1ØØØ
1Ø INPUT "Expression to be tidied?"; e$
2Ø LET l = LEN e$
3Ø LET f$ = " "
```

```
 40  LET i = 1
 50  IF e$ (i) = "↑" THEN GO TO pow
 60  IF e$ (i) = "*" THEN GO TO prod
 70  LET f$ = f$ + e$ (i)
 80  LET i = i + 1
 90  IF i > l THEN GO TO print
100  GO TO 50
200  REM powers
210  LET j = i + 1
212  IF j > l THEN GO TO print
215  LET c = CODE e$ (j)
220  IF c < 48 OR c > 57 THEN GO TO 500
230  LET f$ = f$ + CHR$ (c + 95 + 10 * (c = 48) )
240  LET j = j + 1
245  IF j > l THEN GO TO print
250  GO TO 212
500  LET i = j
510  GO TO 50
600  REM products
610  LET i = i + 1
620  GO TO 50
1000 REM print result
1010 PRINT "Old: □"; e$ ' ' "New: □"; f$
```

TESTAR E GUARDAR

Experimente utilizar as expressões apresentadas acima, na sua versão para o Spectrum, como os *inputs* e$, e verifique se aparecem os resultados correctos. Experimente agora usar outras expressões, como por exemplo:

```
2 * x * y * z − 17 * a ↑ 552
m ↑ 77 + n ↑ 88
a* b * c * d * e * f * g + 45 * h ↑ 999 + 32
```

É evidente que pode usar quaisquer outras expressões deste tipo. No entanto, o programa não resolve *todos* os problemas: veja, por exemplo, o que acontece com 2 * 2! Seja como for, serve para ilustrar a ideia.

Para guardar (SAVE) este programa numa forma útil, temos de trabalhar um bocado mais, porque não é possível guardar automaticamente as imagens definidas pelo utilizador; é necessário descobrir primeiro onde elas se encontram.

No Spectrum de 16K começam no endereço 32600, mas isso pode ter sido alterado através da variável do sistema UDG, que se situa nos endereços 23675-6. Assim sendo, o leitor pode preferir encontrar o endereço, usando a instrução:

```
PEEK 23675 + 256 * PEEK 23676
```

No entanto, existe uma maneira mais fácil, porque o carácter gráfico «a» tem de iniciar a área de UDG. Dado isto, também vai servir a instrução:

```
USR "a"
```

(PRINT os valores e veja.)

Enrole a fita até ao local onde quer começar e acrescente mais uma linha para preparação daquilo que se vai seguir, por exemplo:

```
1 LOAD "exp" CODE USR "a", 80
```

Agora, guarde todo o programa, usando a instrução:

```
SAVE "algebra" LINE 1
```

Assim, vai começar automaticamente na linha 1.

Mas ainda não está tudo feito. Agora, tem de usar armazenamento de *bytes* para guardar as imagens:

```
SAVE "exp" CODE USR "a", 80
```

Esta instrução pega nos 80 *bytes* que começam em USR "a", a área das imagens definidas pelo utilizador, e guarda-os em fita. O número 80 é mencionado na instrução porque cada carácter tem 8 *bytes* e são 10 caracteres, de modo que o número de *bytes* total é 80 × 10 = 80. Deve notar-se que a linha 1, que acrescentamos, inverte este processo para que estes *bytes* voltem a ser introduzidos (LOAD) no computador.

Se tanto o programa como os caracteres já estiverem guardados em segurança na fita, enrole-a até voltar à posição inicial e dactilografe:

```
LOAD "algebra"
```

Vai ouvir os zumbidos e apitos habituais, vão aparecer as riscas vermelhas, amarelas e azuis do costume, e por aí fora, até aparecer uma mensagem:

Program: algebra

Deixe a fita correr. O programa «algebra» vai começar automaticamente na linha 1, que passa automaticamente para o computador as imagens definidas pelo utilizador. Depois de mais zumbidos, mais apitos e mais riscas, aparece a mensagem:

Bytes: exp

Após esta mensagem, o programa «algebra» segue para a próxima linha e continua a passar como de costume. Pare a fita, e pronto.

O que acabamos de fazer é um exemplo de *ligação* entre programas gravados. Para esta ligação, tirámos partido do facto de poderem ser acrescentadas instruções LOAD a programas já escritos. Para outro exemplo, ver o capítulo 15: «Alterações no conjunto de caracteres».

Projectos

1. Modificar «algebra» de modo que o programa resolva quaisquer produtos de *números*, como por exemplo 23 * 45, em vez de se limitar a justapô-los de modo a dar 2345 (o que está errado), que é o que acontece na versão de que dispomos.
2. Modificar o programa de modo que seja feita a substituição de expressões como x * x por x^2, x * x * x e x * x por x^3, e assim sucessivamente.
3. Modificá-lo de modo que as variáveis sejam ordenadas alfabeticamente, transformando abcbab em aabbbc ou, ainda melhor, em a^2b^3c.
4. Modificá-lo de modo que sejam removidos quaisquer termos multiplicados por Ø.
5. Encontrar uma expressão que não seja correctamente tratada pela sua versão mais completa do programa e modificá-lo de modo que o faça.
6. GO TO 5.

Há programas que executam tarefas cuja finalidade é independente e programas que ajudam a escrever outros. Chamam-se a estes programas utilitários.

12

RENUMERAÇÃO DE LINHAS

A ordenação de números de linhas pode ser uma corveia terrível — tanto assim que ninguém a faz, *a não ser* que tenha um programa utilitário que evite fazê-la directamente. É possível comprar programas de renumeração de linhas muito imaginativos; também é possível escrever um desses programas para uso próprio, poupando assim entre 5 e 10 libras, mas, muito provavelmente, perdendo alguma da versatilidade do programa de compra.

O programa que se segue é um compromisso entre esses dois. Uma vez que este livro não versa sobre Linguagem Máquina, o programa tem de ser em BASIC; mas, apesar de esta linguagem ser lenta, o programa tem de ser suficientemente rápido para que valha a pena usá-lo. Daqui se retira que o programa tem de ser bastante rudimentar. Especificamente, *apenas* são renumeradas as linhas: os números de GO TO ou GO SUB *não* são automaticamente renumerados. Após a apresentação da listagem do programa, voltarei a referir-me a este assunto.

OS NÚMEROS DE LINHAS EM PROGRAMAS

Como acontece com a maior parte dos programas utilitários, este programa necessita de estar na posse de alguns conhecimentos acerca do que se passa no interior do Spectrum quando se carrega nos botões. O leitor deve estar lembrado (*Easy Programming,* p. 93) de que o programa é guardado na RAM sob a forma de uma série de *bytes* de caracteres começando no endereço contido na variável do sistema PROG e acabando imediatamente antes de o endereço contido em VARS. Se consultar o *Manual,* verá que estes endereços podem ser encontrados através do uso das instruções:

PROG: PEEK 23635 + 256 * PEEK 23636
VARS: PEEK 23627 + 256 * PEEK 23628

Também poderá aí descobrir que cada linha do programa está guardada sob o formato seguinte:

NS	NJ	LJ	LS	Código linha	ENTER

Neste quadro, NS e NJ são os *bytes* sénior e júnior do número da linha e LJ e LS são os *bytes* júnior e sénior do número de caracteres total da linha.

Concretizando, se o número da linha for n, temos:

NS = INT (n/256)

NJ = n − 256 * NS

Para LJ e LS procede-se da mesma forma. Assim, para a linha 700, por exemplo, temos:

NS = INT (700 / 256) = 2

NJ = 700 − 256 * 2 = 188

São estes os dois primeiros *bytes* da secção da RAM onde está guardada a linha 700.

Se mudássemos NJ para, digamos, 198, podíamos enganar o Sistema Operador, levando-o a pensar que se tratava, de facto, da linha 710 (= 2 * 256 + 198). Esta circunstância sugere uma maneira de renumerar as linhas:

Percorremos o programa, fazendo PEEK para encontrar ocorrências do carácter ENTER (cujo código é 13). Quando as encontramos, sabemos que os dois *bytes* seguintes são um número de linha. Vamos então POKE estes *bytes* com o novo valor desejado.

PRIMEIRA TENTATIVA

Se tentar escrever uma rotina que faça exactamente o que acabamos de descrever, vai encontrar algumas dificuldades inesperadas. Em primeiro lugar, a primeira linha não é renumerada, pois não é precedida por um carácter ENTER; em segundo lugar (e por uma razão menos clara), se se der o caso de um dos *bytes* LS ou LJ ser 13, há problemas.

Estas falhas remedeiam-se com facilidade: renumera-se a primeira linha de acordo com o que se quiser e saltam-se os LJ e LS.

De momento, vamos partir do princípio de que queremos que os

novos números comecem em 10 e vão subindo de 10 em 10. Assim sendo, chegamos ao resultado que se segue:

```
1000  LET prog = PEEK 23635 + 256 * PEEK 23636
1010  LET vars = PEEK 23627 + 256 * PEEK 23628
1020  LET ns = 0: LET nj = 10
1030  FOR i = prog TO vars - 1
1040  IF i = prog THEN POKE i, ns: POKE i + 1, nj:
      LET i = i + 4: LET nj = nj + 10: IF nj >= 256
      THEN LET nj = nj - 256: LET ns = ns + 1
1050  IF PEEK i = 13 THEN POKE i + 1, ns: POKE i + 2, nj:
      LET i = i + 5: LET nj = nj + 10: IF nj >= 256
      THEN LET nj = nj - 256: LET ns = ns + 1
1060  NEXT i
```

Dactilografe esta rotina no seu aparelho, fazendo-a preceder por algumas linhas para serem renumeradas:

```
 1 REM
 2 REM xxx
17 REM
```

Continue por aí fora. Seguidamente, carregue em GO TO 1000. Valha-nos Deus! Pára tudo com uma mensagem de erro:

```
N  Statement Lost, 1060: 1
```

Por que é que isto aconteceu?
O que há a fazer é LIST, e pensar bastante.

SEGUNDA TENTATIVA

O que aconteceu é óbvio: a tonta da rotina acaba por se pôr a renumerar-se *a ela própria.* Quando a linha 1030 tiver sido renumerada, o NEXT i em 1060 envia de novo o aparelho para a linha 1060, que já não existe...

Bom, isto é fácil de remediar; pára-se a renumeração *antes* de ser atingida a própria rotina de renumeração. A maneira mais simples de o conseguir é alterar a linha 1030, substituindo vars-1 por vars-len, em que len é o comprimento em *bytes* da rotina de renumeração. (Descobre-se qual é usando vars-prog.) Para esta rotina particular, len = 385, de modo que a linha 1030 deve passar a ser:

```
1030  FOR i = prog TO vars - 385
```

Agora, tudo funciona bem, e, para BASIC, com uma relativa rapidez. Um programa com cinquenta linhas leva cerca de vinte segundos a ser renumerado, o que não é instantâneo, mas é suficientemente rápido para poupar uma data de trabalho. (Se quiser uma rotina simples e tão limitada como esta, mas que *é* instantânea, em Linguagem Máquina, veja *Machine Code and Better Basic,* p. 159, ou consulte o capítulo 16 de *Spectrum Machine Code.*)

TERCEIRA TENTATIVA

Talvez já tenha ocorrido ao leitor que é possível que estejamos a ser um bocado parvos. Podemos tirar partido dos *bytes* de comprimento de linha, LS e LJ: em vez de andarmos laboriosamente a PEEK ao longo do programa à procura de ENTER, devíamos ser capazes de saltar imediatamente para os próximos *bytes* de número de linha através da adição do comprimento de linha (mais *byte* menos *byte*).

Também lhe pode ter ocorrido que é provável que esta modificação acabe por não poupar muito tempo por causa do processamento requerido pela adição, etc. A única maneira de descobrirmos se poupamos tempo ou não é experimentarmos.

Aqui está um programa escrito de acordo com a nova ideia e que tem, pelo menos, uma vantagem muito evidente: é mais curto.

```
1010  LET i = PEEK 23635 + 256 * PEEK 23636:
      LET ns = 0: LET nj = 10: LET fin =
      PEEK 23627 + 256 * PEEK 23628 - 285
1020  IF i >= fin THEN STOP
1030  POKE i, ns: POKE i + 1, nj: LET nj = nj + 10:
      IF nj >= 256 THEN LET nj = nj - 256:
      LET ns = ns + 1
1040  LET i = i + PEEK (i + 2) + 256 * PEEK (i + 3) + 4:
      GO TO 1020
```

É óbvio que a variável *fin* é a antiga variável *vars* menos o comprimento (285) desta rotina.

Para ver qual é o método mais rápido, introduzimos em cada um, usando MERGE, um programa razoavelmente longo e cronometramos o tempo de execução das rotinas de renumeração. (Vai ter de utilizar programas cujos números de linhas sejam inferiores a 1000, para evitar sobreposições com as rotinas — ver abaixo.) Faça isto antes de conti-

nuar a leitura... ou, pelo menos, tente adivinhar fundamentadamente...

Com um programa de 5Ø linhas, os resultados do meu teste foram:

Primeira rotina:	2Ø segundos.
Segunda rotina:	1 segundo.

Este impressionante melhoramento mostra bem até que ponto é importante que não se deixe de pensar num programa pelo simples facto de ele já *funcionar*.

VERSÃO FINAL (?)

No entanto, ainda não acabámos. A tarefa final é afinar a rotina para maximizar a sua utilidade. Quais serão os critérios a ser satisfeitos?

1. A rotina deve ocupar o mínimo de memória possível.
2. Deve ser escrita no menor número de linhas possível.
3. Essas linhas devem encontrar-se em áreas nunca, ou quase nunca, usadas por um programa normal. Na falta de um truque batoteiro em Linguagem Máquina e de um abaixamento de RAMTOP, o melhor local para as colocar é nas linhas 9995-9998. (Reserve 9999 para o caso de querer acrescentar mais um bocado de programa, usando 9999 GO TO seja lá onde for...)
4. Os nomes escolhidos para as variáveis devem ser nomes que o leitor não utilize habitualmente, de forma a ser seguro fundir (MERGE) a rotina e usá-la com um programa que tenha variáveis introduzidas directamente.
5. Era agradável podermos começar e acabar com linhas escolhidas arbitrariamente e ter incrementos e valores de base arbitrários para os novos números. (A primeira destas sugestões obriga a verificações extra e gasta mais tempo, pelo que não a vou incluir; a segunda é essencial, pois o leitor pode querer MERGE uma das suas rotinas favoritas com qualquer programa cujos números de linhas se lhe sobreponham, precisando portanto de renumerar a rotina de forma apropriada.)

Tomando em consideração estes critérios (mas estando consciente de que não é possível fazer tudo ao mesmo tempo, pois certas coisas são antagónicas, sendo necessário um compromisso), acabei por chegar a esta versão final:

```
9994 INPUT "Start, inc?"; o1, o2: LET o3 =
     INT (o1/256): LET o4 = o1 - 256 * o3
9995 LET o = PEEK 23635 + 256 * PEEK 23636:
     LET oo = PEEK 23627 + 256 * PEEK 23628 - 315
9996 IF o >= oo THEN STOP
9997 POKE o, o3: POKE o + 1, o4: LET o4 = o4 + o2:
     IF o4 >= 256 THEN LET o4 = o4 - 256: LET
     o3 = o3 + 1
9998 LET o = o + PEEK (o + 2) + 256 * PEEK (o + 3) + 4:
     GO TO 9996
```

Esta rotina é um pouco mais longa — cerca de 3 segundos para um programa com 5Ø linhas. É este o preço que se paga por um pouco mais de flexibilidade.

A razão de ser dos «oo» que aparecem nesta rotina está, obviamente, no facto de ser muito raro o uso de «o» para variável em programas normais, dado o perigo de serem confundidos com «zero».

Esta rotina é genuinamente útil. O que se deve fazer é guardá-la sob um nome como, por exemplo, «ren». Antes de escrevermos um programa, introduzimo-la (LOAD) na parte superior da área de programas (os seus elevados números de linhas permitem-no). Escrevemos então o programa. Ordenamos as linhas chamando *ren*, através do uso de GO TO 9994; editamos os GO TO e os GO SUB para os seus valores correctos; finalmente, quando estivermos satisfeitos, editamos as linhas 9994-9998 e guardamos o programa final.

No caso de se ter esquecido de introduzir *ren* no princípio, é sempre possível MERGE esta rotina mais tarde.

RENUMERAÇÃO DE BLOCOS

Este programa é óptimo se se quiser fazer listagens cujos números de linhas sejam inexoravelmente 1Ø, 2Ø, 3Ø, ..., sem haver quebras — ou, se se quiser variar, 1Ø2, 1Ø4, 1Ø6, ... —; contudo, isto nem sempre corresponde ao que se quer. A listagem fica ordenada, mas pode tornar-se menos útil.

Se leu com atenção as pp. 87-92 de *Easy Programming,* deve saber que uma forma de fazer programas civilizados e que funcionem é dividi-los em blocos, constituindo cada bloco uma sub-rotina; usar números de linhas como nomes para os blocos (como, por exemplo, LET block = 1ØØØ); e fazer cada bloco começar num belo número redondo (1ØØØ, 2ØØØ, etc., dependendo do número de blocos com que vamos ter de trabalhar).

A renumeração de uma ponta à outra vai de alguma forma subverter esta estrutura tão cuidadosamente planeada. Por outro lado, quando estiver a desenvolver o bloco 1000, vai rapidamente chegar à conclusão de que a correcção dos gatos dá linhas como 1035, um pouco mais tarde como 1032 e 1033, e por aí fora; no fim, vai estar tudo completamente desordenado, ou pode mesmo acontecer que precise de inserir uma linha entre a 1046 e a 1047. O intérprete de BASIC não vai gostar da ideia da existência de uma linha 1046½ ou 1046.5.

Assim, era agradável podermos efectuar renumerações *no interior de um bloco* (onde, a propósito, os problemas de GO SUB e GO TO são muito menos frequentes e, num programa bem estruturado, praticamente inexistentes). A rotina que se segue efectua esta operação: o utilizador *input* o início e o fim do bloco, o novo número onde ele deve começar e o novo incremento, e a rotina faz o resto.

Se pedir à rotina que renumere um bloco a partir de um número de linha mais baixo do que o actual, ela apita e pede-lhe que volte a introduzir os dados. (Isto é feito porque o sistema BASIC não gosta que as linhas se desorganizem na RAM.) Do mesmo modo, se acabar a renumeração do bloco com um número de linha maior do que o que se lhe segue, a rotina apita e continua a renumeração até chegar ao fim.

Tal como a escrevi, a rotina encontra-se na linha 9000. Se introduzir por comando directo LET ren = 9000, pode ter-lhe acesso dactilografando GO TO ren. É evidente que pode ser preferível um número de linha mais alto, como, por exemplo, 9990, e as variáveis podem, como foi feito acima, ser mudadas por outras menos comuns. No entanto, preferi aqui evitar esses estratagemas por uma questão de clareza.

```
8999  STOP
9000  INPUT "Start/end of block"; stt, end:
      INPUT "Start/inc of new numbers"; nst, inc:
      IF nst < stt THEN BEEP .1, 0: GO TO 9000
9010  IF end > = 8999 THEN LET end = 8998
9020  LET ns = INT (nst/256): LET nj = nst - 256 * ns
9030  LET i = PEEK 23635 + 256 * PEEK 23636
9040  IF 256 * PEEK i + PEEK (i + 1) < stt THEN
      GO SUB 9080: GO TO 9040
9050  IF 256 * PEEK i + PEEK (i + 1) > end THEN
      GO TO 9090
9060  POKE i, ns: POKE i + 1, nj: LET nj = nj + inc:
      IF nj > = 256 THEN LET nj = nj - 256: LET ns = ns + 1
```

```
9070  GO SUB 9080: GO TO 9050
9080  LET i = i + PEEK (i + 2) + 256 * PEEK (i + 3) + 4: RETURN
9090  IF 256 * ns + nj - inc > 256 * PEEK i + PEEK (i + 1)
      THEN BEEP .1, 20: LET end = 8998: GO TO 9060
```

Este é outro programa utilitário verdadeiramente prático, especialmente se for usado em conjunção com MERGE.

Projecto

Pense em tratar automaticamente da renumeração de GO TO e de GO SUB. (Não tratei disso neste capítulo em parte por preguiça, em parte por o resultado ser muito mais lento e em parte porque o uso de *sub-rotinas com nome,* que muitas vezes constitui uma boa prática de programação, requer que se mate ainda mais a cabeça. Por exemplo, em SUPERFILLER a sub-rotina *test* encontra-se na linha 2000, sendo isto estabelecido por inicialização da variável test = 2000, na linha 20. Quando a linha 2000 for renumerada, não vai ser necessário alterar GO SUB test (linha 250): o que vai *mesmo* ser preciso mudar é a linha 20, onde está o valor atribuído a *test.* Mas, mesmo que dê atenção a isto, existem particularidades de GO SUB que são baralhadas pela renumeração.)

As rotinas não se encontram protegidas contra a subida excessiva dos números de linhas (acima de 9999). Modifique este aspecto. (No entanto, esta modificação vai tornar a rotina um pouco mais lenta. Será que vale a pena?)

Não só é simplesmente admirável como é admiravelmente simples.

13

POLÍGONOS

Esta ideia é essencialmente muito fácil, mas só é possível realizá-la sem ajuda se se estiver à vontade em trigonometria (SIN, COS, TAN, e o resto). O objectivo consiste em desenvolver as imagens do Spectrum de modo a permitir a construção de polígonos — *e* seres mais estranhos do mesmo jaez.

Como sem dúvida o leitor se lembra, um polígono é uma figura formada por segmentos de recta. Dum certo ponto de vista, o Spectrum não pode desenhar mais do que isso; contudo, se os segmentos forem suficientemente pequenos, os resultados aproximam-se de curvas. É por esta razão que se numa imagem de alta resolução da Bo Derek, a sua silhueta não *parece* ser feita de segmentos de recta...

Bom, mas voltemos ao Spectrum. Um polígono é *regular* se tiver todos os lados e todos os ângulos iguais entre si, de modo a ser agradavelmente simétrico. Para desenhar um polígono regular com n lados inscrito numa circunferência de raio r, centrada no ponto x, y, use o seguinte programa:

```
10 INPUT "Number of sides?"; n
20 FOR i = 0 TO n
30 LET a = x + r * COS (2 * i * PI / n):
   LET b = y + r * SIN (2 * i * PI / n)
40 IF i = 0 THEN PLOT a, b
50 DRAW a - PEEK 23677, b - PEEK 23678
60 NEXT i
```

Só como está, há o perigo de a imagem ultrapassar os limites do visor; por isso, vamos acrescentar:

```
5 IF r > x OR r > y OR r > 255 - x OR r > 175 - y
  THEN RETURN
```

O «RETURN» foi posto aqui porque estou a pensar em usar tudo isto numa sub-rotina. Assim, é evidente que também vou precisar de:

```
70 RETURN
```

Desta forma, já fica tudo arranjado. Perfeito, mas até ao presente sem a possibilidade de ser usado. Vamos portanto acrescentar:

```
100 INPUT "Radius?"; r
110 INPUT "Centre?"; x, y
120 CLS
130 GO SUB 5
```

Experimente este programa, mas não se esqueça de começar com GO TO 100.

Está tudo muito bem, mas torna-se aborrecido depressa. No entanto, podemos tornar os resultados muito mais bonitos se usarmos um *loop*. Por exemplo, remova a linha 10 e introduza:

```
200 LET n = 5: LET x = 127: LET y = 87
202 CLS
210 FOR r = 5 TO 85 STEP 5
220 GO SUB 5
230 NEXT r
```

O resultado é o que nos é apresentado pela figura 13.1.

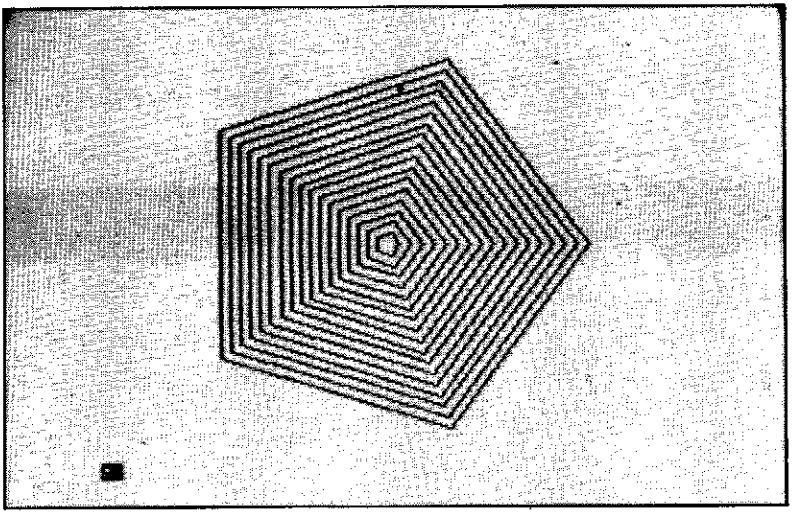
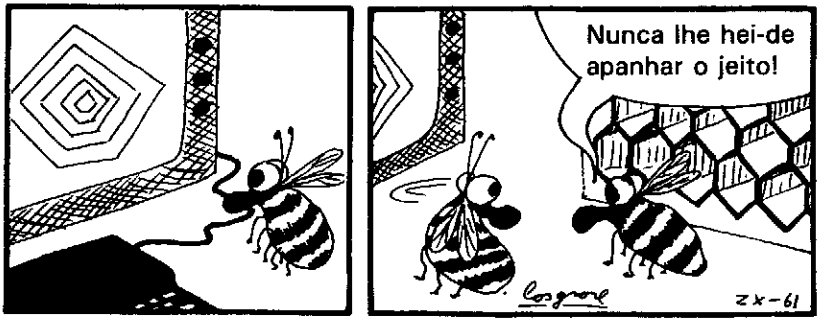


Fig. 13.1 — *Pentágonos concêntricos.*



No caso de gostar desta imagem, acabe com o "LETn = 5", na linha 200, e acrescente:

```
205  INPUT "Number of sides?"; n
```

Desta forma, pode experimentar usar diferentes números. Agora, GO TO 200.

Experimente com n = 50: aparecem círculos bastante aceitáveis (embora mais lentamente do que usando CIRCLE). Era precisamente aqui que eu queria chegar com aquela observação acerca da Bo Derek...

ROTAÇÕES

Passado algum tempo, isto também começa a ser aborrecido. Os maçadores dos polígonos estão sempre virados para o mesmo lado; por isso, vamos acrescentar uma rotação:

```
206  INPUT "Rotate?"; ro
```

Agora, rodamos a figura ro graus, desde que mudemos a linha 30 para:

```
30   LET a = x + r * COS (2 * i * PI / n + ro * PI / 180):
     LET b = y + r * SIN (2 * i * PI / n + ro * PI / 180)
```

Façamos coisas ainda mais fantasiosas:

```
300  LET n = 7: LET x = 127: LET y = 87
305  CLS
310  FOR r = 5 TO 85 STEP 5
320  LET ro = r * 2
330  GO SUB 5
340  NEXT r
```

Se usar GO TO 300, obterá o resultado que nos é mostrado pela figura 13.2. Para poder ter mais variedade, troque o n=7 por um INPUT livre. Na linha 32Ø pode mudar o r * 2 para r, r * 3 ou o que quer que lhe apeteça. Até mesmo r * r/5Ø dá um resultado bastante agradável.

Ao olhar para a linha 3Ø sob a forma por ela apresentada neste momento, não consigo deixar de pensar no que aconteceria se as posições de marcação a e b fossem rodadas de acordo com números de graus *diferentes*. Por outras palavras, vamos alterar essa linha da seguinte forma:

```
3Ø  LET a = r * COS (2 * i * PI / n + ro1 / 18Ø):
    LET b = r * SIN (2 * i * PI / n + ro2 / 18Ø)
```

Os valores de ro1 e ro2 são introduzidos através de:

```
26  INPUT "Rotate 1 and 2?"; ro1, ro2
```

Primeiro, experimente isto começando por GO TO 200. O resultado é que o polígono fica mais ou menos torcido. Seguidamente, adapte a linha 32Ø da rotina com início em 3ØØ, transformando-a em, por exemplo:

```
32Ø  LET ro1 = r * 2: LET ro2 = r * 3
```

As figuras estão a tornar-se mais complicadas, agora: além disso, os resultados são menos previsíveis.

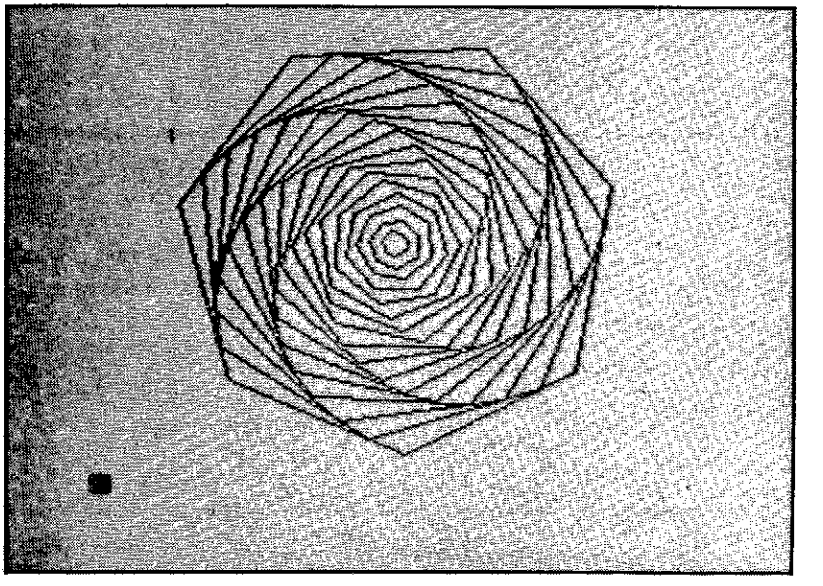


Fig. 13.2 — *Heptágonos rotativos.*

LADOS CURVOS

Que mais poderemos fazer? Bom, a instrução DRAW pode ser utilizada para desenhar tanto linhas direitas com curvas (*Easy Programming*, p. 34). Assim, podemos acrescentar isso como uma opção. Mudemos a linha 50 para:

```
50  DRAW a - PEEK 23677, b - PEEK 23678, bend
```

Tratemos de introduzir em qualquer lado a quantidade de curvatura *(bend)*. Por exemplo:

```
207  INPUT "bend?", bend
```

O leitor vai chegar à conclusão de que as curvaturas entre −4 e +4 são as que dão melhores resultados; pessoalmente, de entre estes valores, prefiro os negativos.

Experimente um bocado com GO TO 200. Seguidamente, coloque as curvaturas no *loop* da linha 300. Acrescente, por exemplo:

```
315  LET bend = -r/25
```

Além disso, mude a linha 320 para:

```
320  LET ro1 = r/2: LET ro2 = r/3
```

Agora, isto está a tornar-se bastante complexo...

ESTRELAS

Mude a linha 20 para, por exemplo:

```
20  FOR i = 0 TO 2 * n
```

Agora pode introduzir valores n = 5/2, 7/2, etc., para obter uma *estrela* de 5 ou 7 pontas.

Mude mais uma vez a linha 20, desta feita para:

```
20  FOR i = 0 TO 3 * n
```

Desta vez, pode experimentar com n = 5/3, 7/3, 8/3, e por aí fora. Em geral, valores do tipo n = (número inteiro)/k vão originar formas semelhantes a estrelas se a linha 20 for:

```
20  FOR i = 0 TO k * n
```

(O valor de k tem de ser introduzido ou atribuído.)

Seguidamente, o leitor pode acrescentar instruções de cor; marcar usando OVER 1... É um nunca mais acabar de possibilidades de variar.

Para acabar, aqui vai uma variação bastante engraçada: dactilografe-a no aparelho e indique GO TO 4ØØ. A figura 13.3 mostra o resultado.

Antes de mais nada, altere a linha 2Ø da sub-rotina de acordo com:

```
2Ø  FOR i = Ø TO 3 * n
```

Seguidamente, acrescente:

```
4ØØ  LET n = 11/3: LET x = 127: LET y = 87
4Ø5  PAPER Ø: BORDER Ø: OVER 1: CLS
41Ø  LET ro1 = Ø: LET ro2 = Ø
415  FOR r = 5 TO 75 STEP 1Ø
418  LET bend = -r/25
42Ø  INK 1 + r/16
43Ø  GO SUB 5
44Ø  NEXT r
```

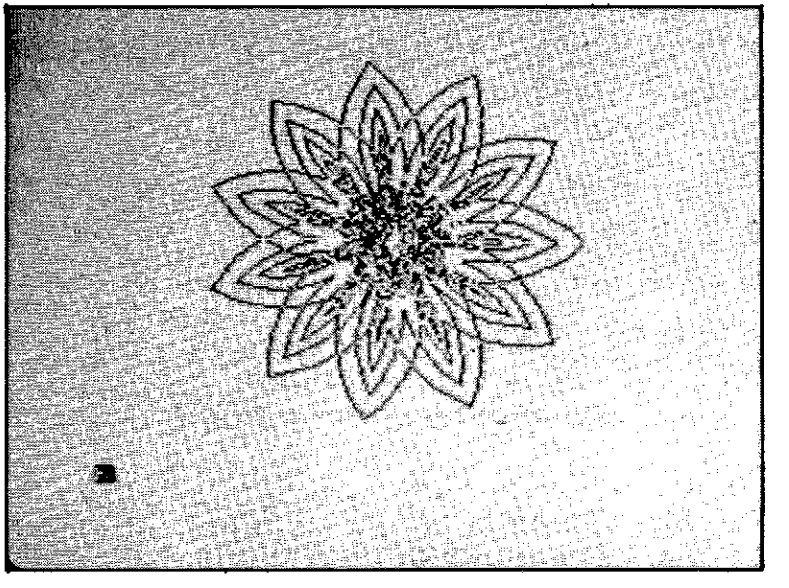


Fig. 13.3 — *5½-ágonos com lados curvos.*

É evidente que o leitor pode usar estas rotinas noutros programas. Por exemplo, ainda não tentámos nada para alterar o centro x, y. Veja se é capaz de desenhar uma disposição de 4×4 pentágonos; uma fila de heptágonos que se sobreponham parcialmente; uma fila curva de hexágonos que vão aumentando progressivamente *e* rodando; um arranjo estocástico de polígonos e estrelas com um número estocástico de lados e estocasticamente coloridos...

Contudo, se o leitor seguiu o que tenho vindo a dizer neste capítulo, não deve precisar de muitos incitamentos para se lançar por sua vez.

Esnes erom sekam ti sdrawkcab gnidaeh siht daer uoy fi...

14

CRIPTOGRAFIA E CRIPTANÁLISE

Estas palavras referem-se à arte de pôr mensagens em código e à descodificação dos resultados sem conhecimento prévio do código utilizado.

Um triste comentário sobre a condição humana é que o mais antigo caso de uso de mensagens em código de que há registo remonta aos Lacedónios, em 400 a. C.

O primeiro livro dedicado a este assunto foi a obra de Tácito *Sobre a Defesa das Fortificações,* escrita no século IV a. C.

O problema inverso — a descodificação de uma mensagem sem conhecimento do código utilizado — tem uma relevância que ultrapassa o aspecto meramente militar. Os historiadores precisam de saber qual o conteúdo de mensagens trocadas por comandantes durante a Guerra Civil Americana; os linguistas necessitam de entender escritas antigas, como os hieróglifos egípcios ou a escrita linear B micénica. Ainda que, *no seu tempo,* não tenham sido códigos, não há dúvida de que, *agora,* o são.

O computador pode ser uma poderosa arma para o criptanalista, pois pode realizar a alta velocidade tentativas por ensaio e erro «iluminadas». (Tentativas por ensaio e erro não iluminadas são demasiado lentas mesmo que se use um Cray-1; o seu Spectrum não teria a menor hipótese.)

Os códigos mais simples são os *códigos de substituição,* nos quais cada letra da mensagem original é codificada de acordo com um alfabeto baralhado. É neste tipo de código que me vou concentrar, para que o capítulo seja mantido dentro de certos limites.

CÓDIGOS DE SUBSTITUIÇÃO

Suponhamos, por exemplo, que a mensagem é:

«My dog has four legs»
(«O meu cão tem quatro pernas»)

Sendo o código definido por:

abcdefghijklmnopqrstuvwxyz
zvetrsnbjpkcuxdfgayohlqimw

A mensagem codificada vai ser:

«um tdn bzy sdha crny»

Isto, claro, lendo da fila de cima para a fila de baixo.

O programa que se segue recebe uma mensagem e codifica-a através de um código de substituição estocástico. Um utilizador inteligente desenvolverá a partir dele uma rotina para *descodificar* este tipo de mensagens.

```
100 LET a $ = "abcdefghijklmnopqrstuvwxyz"
105 PRINT a$
110 LET b$ = " "
120 FOR i = 1 TO 26
130 LET m = INT (1 + (27 - i) * RND)
140 LET b$ = b$ + a$ (m): LET a$ = a$ (TO m - 1) + a$ (m + 1 TO)
150 NEXT i
160 PRINT b$
```

Até este ponto, o programa não faz mais do que conferir ao alfabeto uma ordem estocástica, seleccionando estocasticamente a primeira letra, após o que, também estocasticamente, selecciona, entre as letras que ficaram, a segunda, e assim sucessivamente. Estude-o com atenção: as revistas estão cheias de rotinas de «estocastização» que pegam em duas letras ao acaso, as trocam, e repetem o processo uma data de vezes. É uma maneira inacreditavelmente longa de chegar ao resultado pretendido!

Vamos agora tratar da introdução e codificação da mensagem:

```
200 INPUT m$
210 PRINT m$
215 LET c$ = " "
220 FOR i = 1 TO LEN m$
225 LET c = CODE m$ (i)
230 IF c = 32 THEN GO TO 250
235 IF c < 97 THEN LET c = c + 32
```

```
240  IF c >= 97 AND c < 128 THEN LET c$ = c$ + b$ (c - 96)
250  NEXT i
260  PRINT c$
```

Os aspectos principais a serem notados neste caso são que a linha 235 converte maiúsculas em minúsculas, a linha 230 ignora os espaços, e o c-96, na linha 240, aparece porque o alfabeto corresponde aos caracteres 97-123.

ANÁLISE DE FREQUÊNCIA

De que modo deverá um criptanalista abordar um código deste tipo? (Na prática, este método de codificação desvenda-se com demasiada facilidade para que possa ter alguma utilidade efectiva. O que vamos fazer de seguida é descobrir a razão por que isso acontece.)

O facto essencial que devemos ter em conta é que, qualquer que seja a língua de que estamos a tratar, as letras não ocorrem todas com igual frequência. Em inglês, por exemplo, a letra «E» ocorre com muito mais frequência do que a letra «Z». Abaixo, encontra-se uma tabela da taxa média de ocorrência (por cem letras) de todas as letras do alfabeto na língua inglesa, determinada pela análise de telegramas governamentais.

a	.074	n	.079
b	.010	o	.075
c	.031	p	.027
d	.042	q	.003
e	.130	r	.076
f	.028	s	.061
g	.016	t	.092
h	.034	u	.026
i	.074	v	.015
j	.002	w	.016
k	.003	x	.005
l	.036	y	.019
m	.025	z	.001

Por outras palavras, a letra mais comum é o «e», que ocorre cerca de 13% das vezes; segue-se o «t», com 9,2%; vêm depois o «n», o «r», o «o», o «a», o «i», o «s» e o «d»; o resto das letras têm uma frequência de ocorrência inferior a 4%. Assim sendo, podemos deitar-nos a adivinhar, pensando que a letra mais frequente na versão codificada de uma mensagem razoavelmente longa deve representar o «e», e por aí fora.

Na mensagem acima (que é bastante curta), as frequências são as seguintes:

a	1	em 16
b	1	
c	1	
d	2	
h	1	
m	1	
n	2	
r	1	
s	1	
t	1	
u	1	
y	2	
z	1	

As letras mais comuns são o «n» e o «y», que representam, respectivamente, o «g» e o «s». Não é uma ajuda lá muito grande, mas o texto é muito curto. Suponhamos que tínhamos começado, com um texto mais longo, como por exemplo:

"PEEK, POKE, BYTE and RAM is an excellent computer book"
(«PEEK, POKE, BYTE e RAM é um óptimo livro sobre computadores»)

A versão codificada seria desta vez:

"frrk fdkr vmor zxt azu jy zx rierccrxo edufhora vddk"

(Na prática, os espaços seriam omitidos.) As frequências são, portanto:

a	2	em 43
c	2	
d	4	
e	2	
f	3	
h	1	
i	1	
j	1	
k	3	
m	1	
o	3	
r	8	
t	1	
u	2	
v	2	
x	3	
y	1	
z	3	

A letra mais comum nesta mensagem é o «r», que vamos partir do princípio de que representa o «e»; a seguir vêm o «d», o «f», o «k», o «o» e o «x». Se nos podemos fiar nas frequências, estas letras devem representar, numa ordem qualquer, o «t», o «n», o «r», o «o» e o «a». Na realidade, representam, respectivamente, o «o», o «p», o «k», o «t» e o «n». Três destas letras, o «n», o «o» e o «t», fazem parte do grupo anterior. É possível destrinçá-las com algumas tentativas por ensaio e erro.

Na realidade, o simples facto de sabermos onde vai ficar a letra «e» mostra-nos que a mensagem é:

.ee. ...e ...e ... ... e..e..e.. .....e. ....

Bom... Eis-nos lançados. Algumas palavras inglesas com quatro letras e que se enquadram na forma «.ee.» são: *beef, beer, been, beet, beep, bees, deed, deem, deep, deer, feed, feel, feet, heed, heel, jeep, jeer, keel, keen, keep, leek, leer, lees, leet, meed, meek, meet, need, neep, peek, peel, peep, peer, reed, reef, reek, reel, seed, seek, seem, seen, seep, seer, teed, teem, weed, week, weep.* Quarenta e oito palavras para experimentar... Por fim, havia de chegar à solução.

O PROGRAMA

Visto isto, um programa de criptanálise para códigos de substituição deve fornecer ao utilizador uma análise das frequências relativas das várias letras, e, seguidamente, deixá-lo experimentar palpites e ver qual é o resultado. Isto leva-nos ao programa abaixo:

```
500  REM Frequency analysis
510  DIM n (26)
515  LET col = 0
520  LET tot = LEN c$
525  FOR i = 1 TO 26
530  FOR j = 1 TO tot
540  IF CODE c$ (j) - 96 = i THEN LET n (i) = n (i) + 1
550  NEXT j
554  LET s$ = STR$ (.01 * INT (100 * n (i) / tot) ):
     IF s$ (1) = "." THEN LET s$ = "0" + s$
555  PRINT TAB col; CHR$ (i + 96); "□"; s$;
560  LET col = col + 8
570  IF col = 32 THEN LET col = 0
580  NEXT i
```

Aqui são determinadas as frequências das letras. As linhas 555--580 apresentam os resultados em quatro colunas impressas no visor. A linha 554 é uma tentativa (coroada de êxito) de apresentar apenas duas casas decimais. Se omitir a segunda parte desta linha, referente a s$(1), vai ver que alguns números são impressos como

```
.09
```

enquanto outros aparecem como

```
0.34
```

Ora, desta forma, os resultados ficam com um aspecto confuso e desarrumado. Se pusermos o zero inicial, já parece tudo certinho. Porém, na realidade, não passa de uma exibição de conhecimentos sem verdadeiro interesse.

Vamos agora tratar da descodificação por ensaio e erro:

```
1000  REM decode
1010  LET p$ = " ": FOR i = 1 TO tot:
      LET p$ = p$ + ".": NEXT i
1020  PRINT AT 15, 0; p$
1100  REM trial
1110  INPUT "Code letter?"; k$: IF LEN k$ < > 1
      THEN GO TO 1110
1120  INPUT "Guess at decode?"; g$: IF LEN g$ < > 1
      THEN GO TO 1120
1130  FOR i = 1 TO tot: IF c$ (i) = k$ THEN
      LET p$ (i) = g$
1135  NEXT i
1140  PRINT AT 15, 0; p$
1150  GO TO 1110
```

Não se pode dizer que esteja mal, mas aparecem problemas inesperados. Por exemplo, é possível, sem se dar por isso, tentar usar a mesma letra para descodificar letras diferentes na mensagem cifrada, o que é absolutamente desastroso! Era bom podermos verificar se isto está a acontecer. Para fazermos essa verificação, precisamos de manter um registo do estado actual dos nossos conhecimentos.

```
10  DIM d (26)
1125  GO TO 1500
```

```
1500  REM record of choice
1510  LET k = CODE k$ - 96: LET g = CODE g$ - 96
1520  FOR a = 1 TO 26
1525  IF d (a) < > g OR d (a) = k THEN GO TO 1560
1530  INPUT "You have used □"; CHR$ (g + 96);
      "□ for code letter □"; CHR$ (a + 96);
      "□ do you want to leave it that way?"; y$
1540  IF y$ = "y" THEN GO TO 1110
1550  LET d (a) = 0
1555  GO SUB 2000
1560  NEXT a
1570  NEXT i
1600  LET d (k) = g
1610  GO TO 1130

2000  FOR b = 1 TO tot
2010  IF p$ (b) = g$ THEN LET p$ (b) = "."
2020  NEXT b
2030  RETURN
```

Agora, apenas precisamos de uma forma de, quando acharmos que já descobrimos a mensagem, indicarmos que queremos sair da rotina.

```
1115  IF k$ = "0" THEN GO TO 2500
```

Graças a esta linha, se quando for perguntado *«code letter?»* («letra de código?»), introduzirmos 0, vamos para a apresentação dos resultados.

```
2500  REM wrap it all up
2510  CLS
2520  PRINT "Code message: " ' c$ ' '
      "Decoded message: " ' p$ ' ' '
2530  PRINT "Code known so far:"
2540  FOR i = 1 TO 26
2550  PRINT AT 12, i + 3; CHR$ (i + 96)
2560  IF d (i) < > 0 THEN PRINT AT 13, i + 3;
      CHR$ (d (i) + 96)
```

```
257Ø NEXT i
```

Para evitar fazer batota, o leitor tem de DELETE as linhas 1Ø5, 16Ø e 21Ø. Arranje alguém que introduza a mensagem por si. Senão...

PROBLEMA

Aqui vão quatro mensagens para o leitor descodificar (as soluções encontram-se no fim do livro, p. 205). Todas as mensagens correspondem a citações bastante conhecidas e cada uma delas vem num código de substituição diferente do das outras[1].

Projectos

1. Modificar o programa de modo a apresentar, se pedida, a tabela de frequência das letras na mensagem (por exemplo, através da introdução do número «1» como resposta à pergunta «code letter?»).
2. Acrescentar uma *bubble-sort (Easy Programming,* p. 65) para ordenar as letras usadas por ordem de freqüência, o que simplifica a tarefa de tentar adivinhar.
3. Acrescentar uma opção (introduzir «2» como resposta à pergunta «code letter?») para ser impressa a tabela das frequências normais das letras, para referência.
4. As combinações de duas letras (dígrafos) mais comuns em inglês são:

en	.111	on	.Ø77
re	.Ø98	in	.Ø75
er	.Ø87	te	.Ø71
nt	.Ø82	an	.Ø64
th	.Ø78	or	.Ø64

   Escreva uma rotina de contagem de dígrafos para aproveitamento destas novas informações.
5. Escreva programas para implementação de outros tipos de código (são boas referências sobre este assunto a *Enciclopédia Britânica* e *The Code Breakers,* de D. Kahn) e para a descodificação desses códigos.

[1] Como aconteceu com as outras mensagens cifradas apresentadas neste capítulo, estas estão em inglês, pois a ausência de uma tabela das frequências de ocorrência das várias letras no português torna a versão traduzida impossível de descodificar. *(N. da T.)*

Precisa de mais do que 23 caracteres gráficos definidos pelo utilizador? Agora, com um único POKE, tem à sua disposição 256.

15

ALTERAÇÕES NO CONJUNTO DE CARACTERES

No capítulo 6 mencionei o facto de ser possível estabelecer novos caracteres acima de RAMTOP e ter-lhes acesso através do uso de POKE sobre a variável do sistema CHARS. Neste capítulo, esse processo é descrito em pormenor.

Para simplificar, vamos partir do princípio de que queremos 64 caracteres novos. Assim sendo, vamos precisar de 64 * 8 = 512 *bytes* de espaço livre. Normalmente, num Spectrum de 16K, RAMTOP encontra-se no valor 32599, por isso tem de ser baixada para 32599 – 512 = 32087 para deixar um «sótão» com 512 *bytes*. Para o fazer, introduza (por comando directo)

```
CLEAR 32087
```

A área desimpedida começa no endereço seguinte, 32088. Este valor é 125 * 256 + 88, de modo que o seu *byte* júnior é 88 e o seu *byte* sénior é 125. Levamos CHARS a apontar para esta nova área diminuindo mais 1 ao *byte* sénior (não se esqueça de que CHARS contém 256 *a menos* do que o endereço do quadro de caracteres). A instrução a introduzir vai, pois, ser:

```
POKE 23606, 88: POKE 23607, 124
```

Contudo, não a introduza ainda.

O programa que se segue permite ao leitor estabelecer 64 caracteres novos, e teste-os para ver se tudo ficou bem.

```
10 FOR i = 32 TO 96
20 GO SUB 400
30 PRINT i – 31,
```

```
40  GO SUB 200
50  PRINT CHR$ i
60  PRINT '
70  NEXT i
80  GO SUB 400
90  STOP
200 REM new address for CHARS
210 POKE 23606, 88: POKE 23607, 124
220 RETURN
400 REM usual address for CHARS
410 POKE 23606, 0: POKE 23607, 60
420 RETURN
1000 REM input routine for new character set
1010 LET i = 32088
1020 INPUT j
1030 PRINT j; "□";
1040 POKE i, j
1050 LET i = i + 1: GO TO 1020
```

Neste programa, 200 muda CHARS para a nova área; 400 volta a pô-la na sua posição habitual; 1000 é uma rotina de *input*.

Comece com GO TO 1000. Para experimentar, vamos usar apenas seis caracteres. Tal como acontece com as imagens definidas pelo utilizador, vai ser necessário desenhar uma réplica do carácter numa quadrícula de 8 × 8; converter as filas em 0s e 1s binários; seguidamente, *input* o valor obtido (consultar *Easy Programming*, p. 49). Nesta experiência vou usar os caracteres apresentados pela figura 15.1. Isso significa que vou ter de *input*, por ordem:

255	255	255	255	255	255	255	255	(para ■)
255	129	129	129	129	129	129	255	(para □)
1	3	7	15	31	63	127	255	(para ◢)
255	255	195	195	195	195	255	255	(para ▣)
1	2	4	8	16	32	64	128	(para /)
24	126	126	255	255	126	126	24	(para ●)

É quanto basta. Introduza STOP e, seguidamente, RUN. Devem aparecer no seu visor os seis caracteres classificados como 1, 2, 3, 4, 5 e 6. Se isso não acontecer, verifique cuidadosamente se o seu trabalho está correcto.

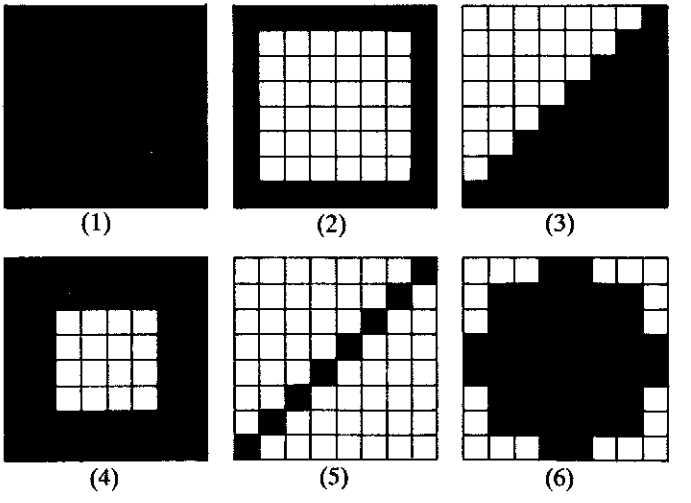


Fig. 15.1 — *Um conjunto de seis caracteres experimentais.*

Agora, que já verificámos que o método dá resultado, vamos à tarefa final, que consiste em *input* os tais 64 caracteres. Essa tarefa vai ser realizada em várias etapas:

1. *Planear* o desenho dos caracteres. O leitor pode utilizar CHARACTER-BUILDER, uma rotina utilitária de *Easy Programming.*
2. Determinar os lados referentes às diferentes filas. (Idem.)
3. *Input* os dados na linha 1000, como foi feito acima.

Tenho a certeza de que o leitor está já a pensar numa data de truques imaginativos que tornem este processo mais fácil, tal como a *combinação* de CHARACTER-BUILDER com o programa acima. Vou mencionar apenas um desses truques. Para introduzir as filas directamente em *binário,* evitando a maçadora conversão para decimal (que é um aborrecimento inútil, pois o Spectrum volta a converter tudo em binário imediatamente), mude a linha 1020 para:

```
1020 INPUT b$
1025 LET j = VAL ("BIN" + b$)
```

Agora, pode *input* as filas como sequências de 0s e 1s, lidos a partir da quadrícula de 8 × 8: 0 para um quadrado vazio; 1 para um quadrado preto.

Que caracteres deve o leitor escolher? Acima, são sugeridas algumas possibilidades. A figura 15.2 apresenta um alfabeto grego.

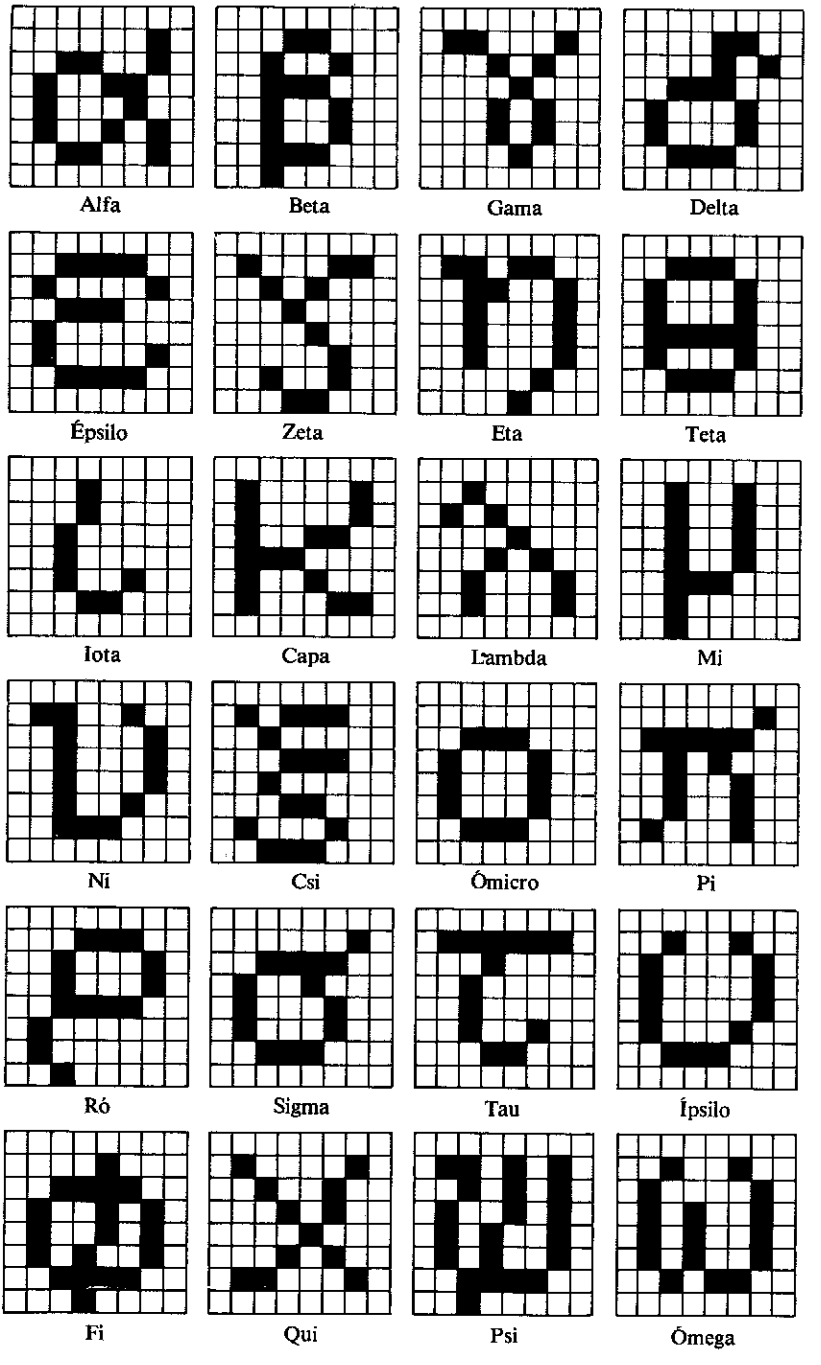


Fig. 15.2 — *Plano para um alfabeto grego.*

COMO USAR O NOVO CONJUNTO

A partir do momento em que os novos caracteres se encontram em memória, para os usar basta mudar CHARS para a nova área e chamar os caracteres pelo número respectivo. Se CHARS for mudada de acordo com a linha **200**, deve chamar CHARS 31 + n, ou, simplesmente, chamar *directamente* o carácter habitual com código 31 + n, para obter o carácter n.º n da sua lista. Para obter os caracteres habituais, volte a mudar CHARS para a sua posição normal, como vem na linha **400**.

COMO GUARDAR O NOVO CONJUNTO

Para SAVE em fita o seu árduo trabalho, o leitor vai precisar de utilizar armazenamento de *bytes*. Dactilografe no seu aparelho a seguinte instrução:

```
SAVE "newset" CODE 32088, 512
```

Desta forma, os 512 *bytes* a partir do endereço 32088 são passados para a fita, sendo-lhes dado o nome de «newset» («novo conjunto»). É evidente que a fita deve ser passada da forma habitual para SAVE.

Para voltar a introduzir no aparelho os novos caracteres, use:

```
LOAD "newset" CODE 32088, 512
```

No entanto, existe uma maneira ainda melhor: escrever um programa que se ocupe da operação de LOAD. Em primeiro lugar, intruduza o programa:

```
10  LOAD "newset" CODE 32088, 512
```

Agora, deve SAVE este programa, sob o nome de «newload», por exemplo. Para isso, use SAVE «newload» LINE 10. Logo a seguir, guarde, em fita, os novos caracteres usando SAVE... CODE como foi feito acima. Assim, a fita passa agora a conter dois pedaços de material.

Volte a enrolar a fita, dactilografe LOAD «newload», carregue nos botões e observe o visor.

A primeira mensagem a ser apresentada vai ser:

```
Program: newload
```

Imediatamente após ter sido introduzido, este programa vai *passar* a partir da linha 1Ø, devido à instrução LINE 1Ø que se encontra no SAVE. *Se deixar a fita a correr,* o programa «newload» introduzirá automaticamente os *bytes* para «newset» (aparece a mensagem Bytes: newset). Desta forma, não precisa de se lembrar de todos os números de endereços, etc.

Este método abre possibilidades completamente novas. O leitor passa a poder encadear topo a topo vários programas, de tal forma que cada um deles chama o seguinte. Se for rápido a controlar a fita e não quiser voltar atrás, pode usar esta ideia de uma série de formas para aumentar realmente o poder do seu aparelho através do pleno uso da capacidade da memória extra existente *na fita.* Os capítulos 9 e 17, sobre Arquivos em *Cassette* e Sistemas de Gestão de Dados, exploram esta ideia num contexto útil.

Um pequeno problema que nos aparece inesperadamente quando usamos as instruções PLOT e DRAW do Spectrum é o aparecimento de mensagens de erro quando os pontos a ser marcados se encontram fora dos limites do visor. O que há a fazer é planear uma rotina utilitária que permita...

16

MARCAÇÃO DE CURVAS À PROVA DE ENGANOS

A maneira mais fácil de desenhar curvas é escrever um *loop* que, depois de marcar (PLOT) um ponto de partida, desenhe (DRAW) sucessivamente em direcção a outros pontos gerados a partir quer de uma lista de dados, quer de uma fórmula. No entanto, este processo causa problemas se os pontos saírem do visor. O que se vai seguir é uma descrição pormenorizada do desenvolvimento de um método para superar este problema. Por vezes, torna-se um tudo nada matemático; no entanto, se a matemática não for o seu ponto forte, ignore a álgebra e concentre-se na estrutura geral.

Deve estar lembrado de que, para as imagens de alta resolução, o Spectrum utiliza uma quadrícula coordenada com 176 linhas e 256 colunas (numerada de Ø-175 e de Ø-225), com início no canto inferior esquerdo do visor. O limite do visor forma portanto um rectângulo com as dimensões de 176 × 256 *pixels*. A chave do problema está em inventar uma sub-rotina que, ao ser alimentada com as coordenadas (x1, y1) e (x2, y2) de dois pontos que *não se encontram necessariamente no visor*, seja capaz de descobrir onde é que a linha que os une toca nos limites desse rectângulo (fig. 16.1).

Alguns conhecimentos da velha geometria analítica permitem-nos computar estes pontos. A expressão algébrica

$$\frac{(a - b)*(d - e)}{(c - b)} + e$$

vai aparecendo sucessivamente sob diversos disfarces, o que indica que talvez valha a pena utilizar uma função definida pelo utilizador (consultar o capítulo 3).

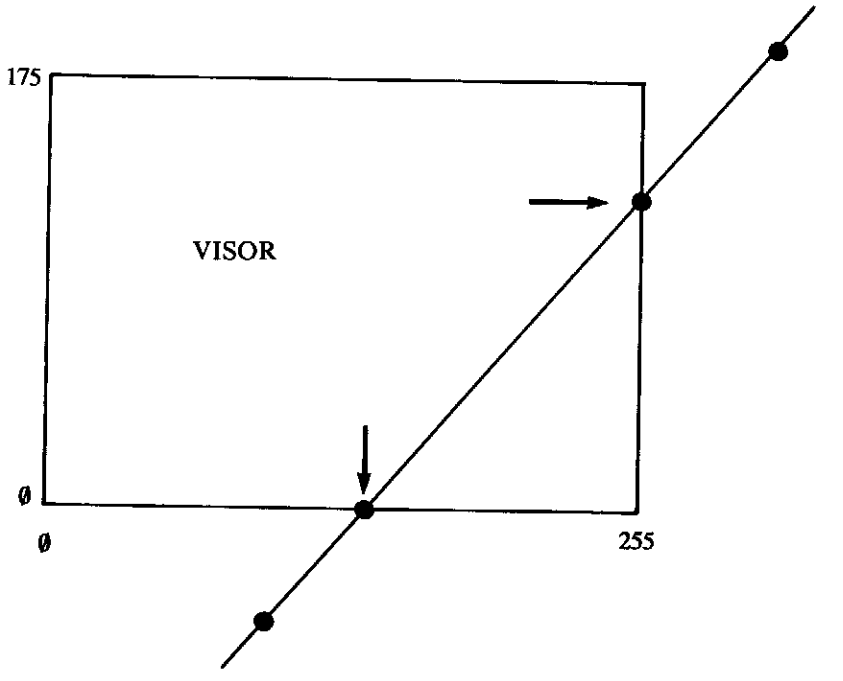


Fig. 16.1 — *Onde é que a linha que une dois pontos intersecta os limites do visor?*

Assim, chegamos à rotina que se vai seguir. (Os números de linhas podem parecer um pouco irregulares, mas a minha intenção é que o leitor possa seguir o capítulo e *input* todas as linhas do programa à medida que vai avançando, de modo a acabar com o programa completo; as descrições, no entanto, são dadas sub-rotina a sub-rotina. Esta forma de proceder facilita sempre o planeamento e a introdução no aparelho de programas.)

```
    2  DEF FN a (a, b, c, d, e) = (a - b) * (d - e) / (c - b) + e
1000  REM seg
1010  DIM x (2): DIM y (2)
1020  IF x1 = x2 THEN LET xt = x1:
      LET xb = x1: LET yl = -1: LET yr = -1
1025  IF y1 = y2 THEN LET xt = -1: LET
      xb = -1: LET yl = y1: LET yr = y1
1030  IF x1 < > x2 AND y1 < > y2 THEN LET
      xt = FN a (175, y1, y2, x2, x1): LET
```

```
      xb = FN a (0, y1, y2, x2, x1): LET
      yl = FN a (0, x1, x2, y2, y1): LET
      yr = FN a (255, x1, x2, y2, y1)
1090  LET q = 1
1100  IF xt > = 0 AND xt < = 255 THEN LET
      x (q) = xt: LET y (q) = 175: LET q = q + 1
1110  IF xb > = 0 AND xb < = 255 THEN LET
      x (q) = xb: LET y (q) = 0: LET q = q + 1
1120  IF yl > = 0 AND yl < = 175 THEN LET
      x(q) = 0: LET y (q) = yl: LET q = q + 1
1130  IF yr > = 0 AND yr < = 175 THEN LET
      x(q) = 255: LET y (q) = yr
1140  RETURN
```

Num programa é necessário fornecer a esta rotina os quatro números x1, y1, x2 e y2 (as coordenadas dos dois pontos). A rotina coloca as coordenadas dos pontos de intersecção da linha que une os pontos dados com o rectângulo limite em

```
x (1), y (1)
x (2), y (2)
```

Para chamar esta sub-rotina podemos usar GO SUB 1000, mas, se quisermos escrever num estilo mais civilizado, inicializamos:

```
22  LET seg = 1000
```

Se tivermos feito isto, podemos usar GO SUB seg. Vamos pois acrescentar a linha 22.

É mais prudente *testar* esta rotina antes de avançarmos mais: a parte algébrica é suficientemente confusa para que eventuais erros cometidos nesta etapa não sejam detectados, vindo mais tarde a mostrar-se desastrosos. Para este efeito, acrescente as seguintes linhas *temporárias:*

```
100  LET x1 = 127: LET y1 = 87
110  FOR t = 1 TO 50
120  LET x2 = 500 * SIN (t * PI / 50):
     LET y2 = 500 * COS (t * PI / 50)
130  GO SUB seg
```

```
140 PLOT x (1), y (1): DRAW x (2) − x (1), y (2) − y (1)
150 NEXT t
160 STOP
```

Agora, RUN o programa. Se obtiver um conjunto de linhas radiais que atravessam o meio do visor e terminam nos seus limites, é porque tudo vai bem; se não, verifique uma vez mais a sua listagem. Quando a rotina estiver correcta, retire as linhas 100-160.

O trabalho que exigia pensamento já está feito. Agora, o que falta fazer é mera rotina... ou, pelo menos, sub-rotina...

COMO DESENHAR CURVAS

A primeira tarefa consiste em decidir qual vai ser a estrutura geral. Há duas formas principais de desenhar uma curva:

1. Um *gráfico*. Marca-se FN (t) contra t, sendo FN uma qualquer função. Para mais pormenores, consulte *Easy Programming*, p. 79.
2. Uma *curva parametrizada*. Marca-se FNb (t) contra FNa (t), sendo FNa e FNb *duas* funções. A t chama-se um *parâmetro*.

Era bom que a nossa rotina nos permitisse optar por qualquer destas soluções. (O que não é difícil, pois um gráfico é de facto um tipo especial de curva parametrizada, na qual FNa (t) = t; porém, geralmente não encaramos da mesma forma estes dois tipos de curva.)

O utilizador vai ter de *estabelecer* estas funções. Podíamos ter no programa uma linha como a que se segue:

```
7777 DEF FNb (t) = uma coisa qualquer
```

Seria então pedido ao utilizador que *editasse* esta linha para a função que lhe apetecesse. Não seria, porém, mais agradável que lhe fosse permitido *input* a função desejada durante a passagem do programa?

A instrução VAL (*Easy Programming*, p. 71) parece feita por medida para este tipo de aplicação. Se o utilizador introduzir FNb (t) sob a forma de, por exemplo, uma *variável literal* f$, podemos calcular o seu valor se pedirmos VAL f$. Por exemplo, se t = 71 e f$ = "t * t", temos:

```
VAL f$ = VAL "t * t" = 71 * 71 = 5041
```

É pois este o valor da função «quadrado» quando t = 71.

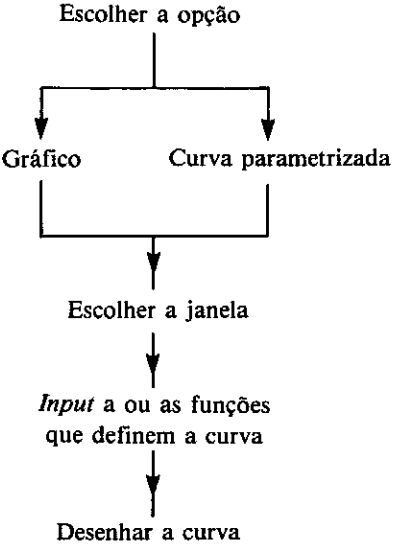
Seria também bastante aborrecido se apenas pudéssemos marcar curvas cujas coordenadas estivessem compreendidas respectivamente

entre Ø e 155 e entre Ø e 175 (trata-se do problema habitual de deslocar e esticar os eixos — consultar *Easy Programming*, p. 81). Uma técnica estandardizada para ultrapassar esta dificuldade consiste em estabelecer uma *janela* que contenha a área desejada e, durante a rotina de desenho, transformar as coordenadas de acordo com o que se pretender (fig. 16.2).

Estamos agora em condições de estruturar o programa, sendo a estrutura apresentada, esquematicamente, a seguir à listagem.

Deixando, por agora, de parte os pormenores (programação *top-down*, consultar *Easy Programming*, p. 87), vamos tratar de escrever o programa principal:

```
100 PRINT "SPECTROGRAPH"
110 PRINT ' "OPTIONS:" ' ' "□ □ 1. Parametrized
    curve [11 spaces] 2. Graph"
120 INPUT "Select option number"; opt
130 GO SUB window
140 CLS
150 IF opt = 1 THEN GO SUB param
160 IF opt = 2 THEN GO SUB graph
170 STOP
```



Estrutura esquemática para o estabelecimento de uma janela

Assim, vamos ter de escrever três sub-rotinas: *window* (janela), *param* e *graph*.

A sub-rotina *window* é extremamente simples:

```
 20  LET window = 500
500  REM window
510  INPUT "Window coordinates: left, right,
     bottom, top"; wl, wr, wb, wt
520  RETURN
```

Como já foi dito, *graph* é um caso especial de *param,* de modo que o que parece mais lógico é escrever primeiro *param* e esperar que *graph* se lhe ajuste com facilidade.

Para marcar uma curva parametrizada, com parâmetro t, precisamos de saber duas coisas: o âmbito dos valores possíveis de t e o intervalo que deve existir entre os pontos a serem computados. Por isso, é necessário que *param* peça estes valores, para seguidamente desenhar a curva.

```
 26  LET param = 2000
2000 REM param
2010 INPUT "Parameter range: left, right"; tl, tr
2020 INPUT "Number of steps?"; ns
2030 INPUT "Specify x and y as functions of t"; x$, y$
2040 LET stepsize = (tr - tl) / ns
2050 GO SUB draw
2060 RETURN
```

Conseguimos assim deixar a maior parte do trabalho para a rotina *draw* (desenhar), que ainda não escrevemos. Óptimo! A rotina *graph* tem um funcionamento semelhante ao da rotina anterior e salta para *draw* na altura conveniente:

```
 24  LET graph = 1500
1500 REM graph
1510 LET tl = 0: LET tr = 255
1520 LET ns = 255
1530 INPUT "Specify function of t"; y$:
     LET x$ = "t"
```

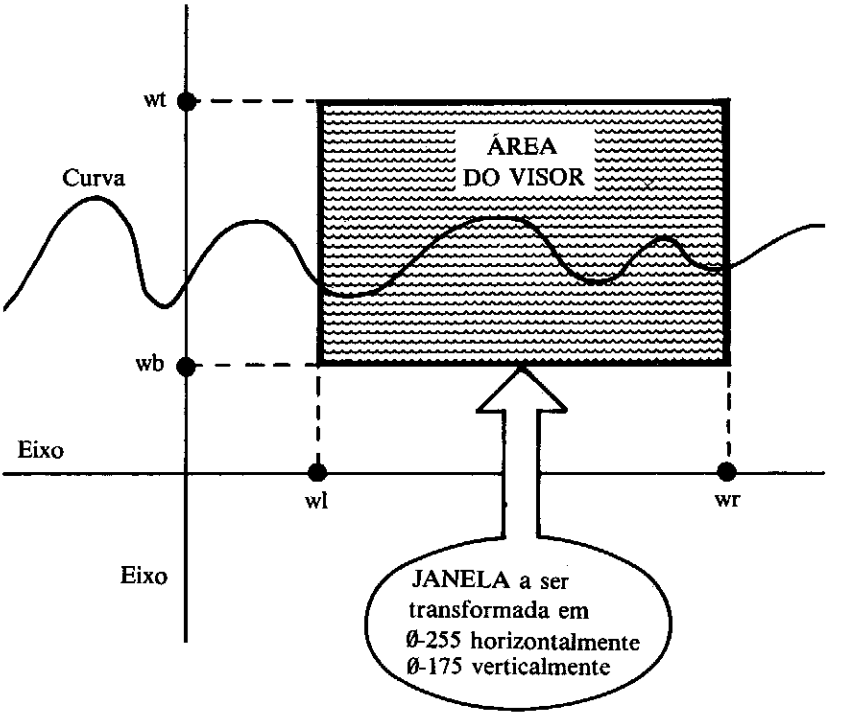


Fig. 16.2 — *A área do visor usada como uma janela.*

```
1540  LET stepsize = 1
1550  GO SUB draw
1560  RETURN
```

Esta rotina foi deliberamente escrita de modo a realçar a analogia com *param: graph* estabelece automaticamente todas as variáveis que se encontram em *param* com a excepção de y$, que é a função introduzida, e, seguidamente, chama *draw* de forma precisamente idêntica. (Era *possível* substituir as últimas duas linhas por GO TO 2050, mas, por uma questão de estilo e dado que não se desperdiça com isso nem muita memória nem muito tempo, resolvemos não o fazer.)

Agora é que não podemos adiar por mais tempo o momento das dificuldades...

```
  28  LET draw = 2500
2500  REM draw
2510  LET t = tl
```

```
2520  LET u = VAL x$: LET v = VAL y$
2530  GO SUB transf
2540  IF FN o (u, v) THEN PLOT u, v
2550  FOR t = tl + stepsize TO tr STEP stepsize
2560  LET u = VAL x$: LET v = VAL y$
2570  GO SUB transf
2580  GO SUB flag
2590  GO SUB d (fl)
2600  NEXT t
2610  RETURN
```

...Bom, se calhar, afinal de contas, ainda podemos. Arranjamos maneira de inventar três novas sub-rotinas (sendo uma delas constituída por quatro partes):

A rotina *transf,* que transforma as variáveis de modo que a janela escolhida se ajuste exactamente à área do visor;

A rotina *flag,* que estabelece uma bandeira para nos informar se os pontos unidos em *draw* se encontram ambos dentro do visor, ambos fora dele ou um dentro e um fora. Esta rotina atribui a uma variável fl o valor 1, 2, 3 ou 4, consoante a combinação precisa das posições dos pontos;

A rotina *d(fl),* que se trata, na realidade, de quatro rotinas *d(1) – d(4),* uma para cada valor da bandeira, pois as acções requeridas diferem consideravelmente entre os vários casos.

Incluímos também uma nova função definida pelo utilizador FN o (consultar o capítulo 3). Esta variável pretende ser uma bandeira «sobre o visor». Ou seja, se definirmos

```
1  DEF FN o (x, y) = x > = 0 AND x < = 255
   AND y > = 0 AND y < = 175
```

então FN o (x, y) = 1 se (x, y) estiver *dentro* do visor e FN o (x, y) = 0 se (x, y) estiver *fora* dele. Não acha que é bonito? É evidente que era possível fazer isto de outras maneiras: também me ocorreu a possibilidade de definir uma bandeira chamada *onscreen* (sobre o visor) e escrever 2540 IF onscreen THEN... . De facto, esta possibilidade resulta com onscreen = FN o (x, y). As coisas que se fazem para que as listagens se assemelhem mais ao inglês real do que à lei da gravidade de Einstein!

O leitor pode estar a pensar que por este andar não chegamos a parte nenhuma, pois ainda não abordámos o problema fulcral de desenharmos *realmente* alguma coisa. Mantenha a fé: não *sente* que o pro-

blema se vai tornando progressivamente mais fácil à medida que o vamos desembaraçando de certas partes periféricas e o dividimos em pedaços mais pequenos?

A transformação das variáveis para uma janela adequada é simples para quem, como eu, teve seis anos de treino matemático:

```
30   LET transf = 3000
3000 REM transf
3010 LET u = (u - wl) / (wr - wl) * 255:
     LET v = (v - wb) / (wt - wb) * 175
3020 RETURN
```

O leitor pode verificar que tenho razão. O que queremos fazer é transformar as coordenadas wl e wr em 0 e 255 e as coordenadas wb e wt em 0 e 175. Se fizermos, à direita do sinal "=", u = wl, vem, à esquerda, u = 0; se fizermos u = wr, vem u = (wr − wl)/(wr − wl) * 255 = = 1 * 225 = 225. Ufa!... Passa-se exactamente o mesmo com wt e wb.

Mas retomemos o que estávamos a fazer:

```
32   LET flag = 3200
3200 REM flag
3210 LET fl = FN o (xo, yo) + 2 * FN o (u, v) + 1
3220 RETURN
```

O funcionamento desta parte é o seguinte: suponhamos que queremos unir o ponto *antigo* (xo, yo) ao ponto *novo* (u, v). Temos então quatro casos possíveis:

Ponto antigo	Ponto novo	Valor de *fl*
Dentro	Dentro	4
Dentro	Fora	2
Fora	Dentro	3
Fora	Fora	1

Neste quadro, dentro/fora refere-se à situação do ponto relativamente ao visor, servindo o valor de *fl* para distinguir os vários casos.

Mas distingui-los para quê? Bom, a acção requerida é diferente consoante os casos:

Valor de f1	Ação requerida
1	Desenhar, se for caso disso, a porção da linha entre o ponto antigo e o novo que ficar dentro do visor.
2	Unir o ponto antigo ao limite do visor, seguindo a linha que leva ao ponto novo.
3	Unir o limite do visor ao ponto novo, seguindo a linha que vem do ponto antigo.
4	Unir o ponto antigo ao novo.

A razão da inclusão da acção (1) está na *possibilidade* de, apesar de tanto o ponto antigo como o novo estarem fora do visor, uma porção da linha entre eles ficar *dentro* dele, devendo portanto ser desenhada (ver a figura 16.1).

ROTINAS GRÁFICAS

Seguem-se todas as rotinas deste tipo, apresentadas pela ordem que, na altura em que foram escritas, pareceu mais simples ao que ainda me restava do cérebro.

```
  34 DIM d (4)
  37 LET d (1) = 3800
  38 LET d (2) = 3600
  39 LET d (3) = 3700
  40 LET d (4) = 3500
3500 REM d (4)-both on
3510 DRAW u - PEEK 23677, v - PEEK 23678
3520 RETURN
3600 REM d (2)-old on, new off
3610 LET x1 = xo: LET y1 = yo:
     LET x2 = u: LET y2 = v
3620 GO SUB seg
3630 LET z = 1
3640 IF u < > xo AND SGN (u - x (1) ) < > SGN (x (1) - xo)
     THEN LET z = 2
```

```
3650 IF u = xo AND SGN (v - y (1) ) < > SGN (y (1) - yo)
     THEN LET z = 2
3660 DRAW x (z) - PEEK 23677, y (z) - PEEK 23678
3670 RETURN
3700 REM d (3)-old off, new on
3710 LET x1 = xo: LET y1 = yo:
     LET x2 = u: LET y2 = v
3720 GO SUB seg
3730 LET z = 1
3740 IF u < > xo AND SGN (u - x (1) ) < > SGN (x (1) - xo)
     THEN LET z = 2
3750 IF u = xo AND SGN (v - y (1) ) < > SGN (y (1) - yo)
     THEN LET z = 2
3760 PLOT x (z), y (z): DRAW u - x (z), v - y (z)
3770 RETURN
3800 REM d (1)-both off
3810 LET x1 = xo: LET y1 = yo: LET x2 = u: LET y2 = v
3820 GO SUB seg
3830 PLOT x (1), y (1): DRAW x (2) - x (1), y (2) - y (1)
3840 RETURN
```

Aquela trapalhada toda com SNG serve para descobrir qual dos pontos x(1), y(1) ou x(2), y(2) é que deve ser usado. Se detestar matemática, faça de conta que não viu nada.

Já está quase. Contudo, as variáveis xo e yo, referentes às coordenadas do ponto antigo, ainda não foram estabelecidas em parte nenhuma. O lugar correcto para o fazer é na linha:

```
2535 LET xo = u: LET yo = v
```

Este estabelecimento deve ser renovado nas linhas:

```
2575 LET ur = u: LET vr = v
2595 LET xo = ur: LET yo = vr
```

TESTE

Estamos agora em condições de testar o fruto do nosso trabalho. RUN o programa e dactilografe no seu trabalho:

Opção (Option)	1			
Janela (Window)	−5	5	−5	5
Âmbito do parâmetro (Parameter range)	−5	5		
Número de saltos (Number of steps)	1ØØ			
Funções de t (Functions of t)	t	t		

Tudo vai bem. Uma linha, que não é totalmente recta, sobe a partir da parte inferior esquerda do visor até ao cimo, à direita. (Não foi este o resultado que obteve? Então é porque engatou alguma coisa!)

Quer experimentar com qualquer coisa um bocado mais complicada? Vamos tentar uma parábola, da qual uma parte se encontra fora do visor. [O primeiro teste não chegou a utilizar as sub-rotinas d(1)-d(3), que são, afinal de contas, a parte mais importante de todo o exercício!]

Opção	1			
Janela	−5	5	−1	2Ø
Âmbito do parâmetro	−5	5		
Número de saltos	1ØØ			
Funções de t	t	t*t		

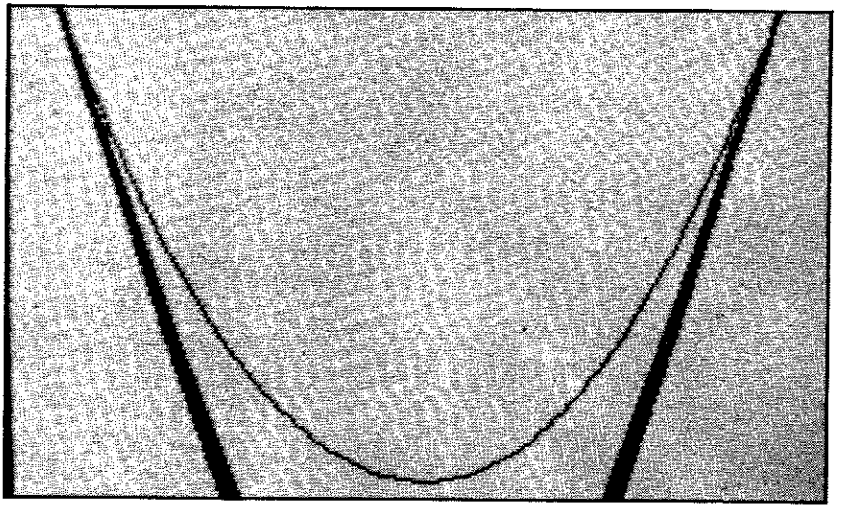


Fig. 16.3 — *Uma parábola! Bem, quase...*

A figura 16.3 mostra-nos o resultado. É bonito, mas é uma parábola um bocado esquisita...

Pode ser que fique surpreendido ao saber o tempo que levei a dar com o gato, ou pode ser que não fique. Seja como for, levei tempo de mais — já devia estar a ficar cansado. Um traçado LPRINT mostrou que a culpa era da sub-rotina d(1), aquela que une dois pontos que se encontram ambos fora do visor. Mas, afinal de contas, que é que estava errado nessa rotina?

Finalmente, a evidência do erro impôs-se. No caso em que os dois pontos a ser unidos se encontram ambos fora do visor e do *mesmo* lado em relação a ele, a *recta* definida por eles pode entrar no visor, ainda que isso não aconteça com o *segmento* de recta que os une (ver a figura 16.4).

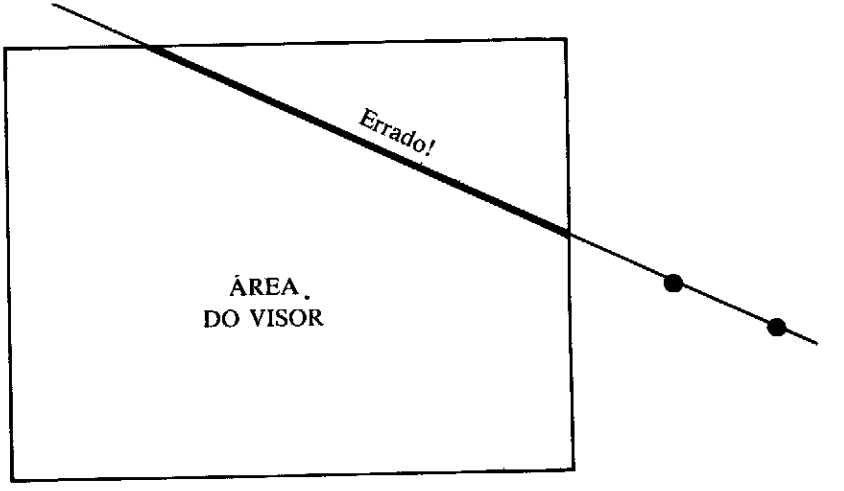


Fig. 16.4 — *A origem do gato.*

Este erro remedeia-se com facilidade recorrendo ao uso de mais uma sub-rotina:

```
  33  LET check = 4000
4000  REM check
4010  LET segon = 0
4020  IF x1 = x2 AND SGN (y1 - y (1) ) =
      SGN (y2 - y (1) ) THEN LET segon = 1
```

```
4030  IF x1 < > x2 AND SGN (x1 - x (1) ) =
      SGN (x2 - x (1) ) THEN LET segon = 1
4040  RETURN
```

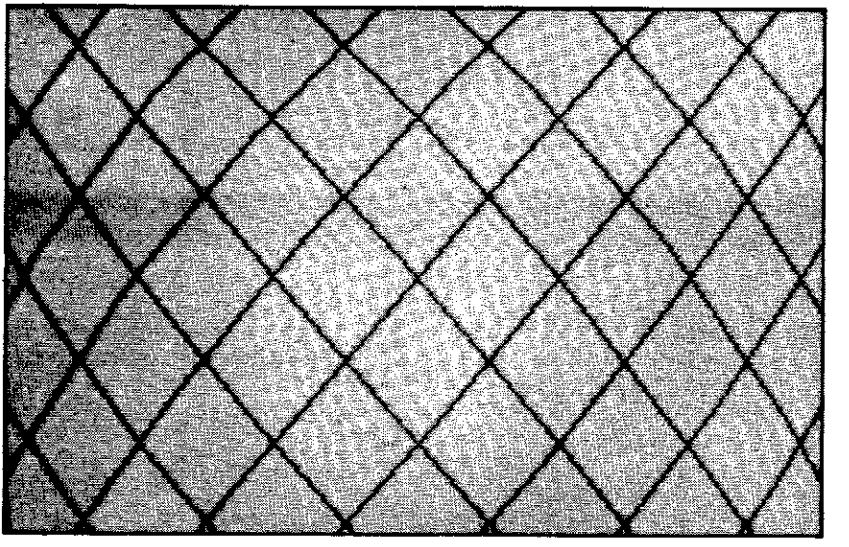
Esta sub-rotina estabelece uma bandeira, *segon,* que, no caso de a situação ilustrada pela figura 16.4 se repetir, toma o valor 1. Para poder ter-lhe acesso, acrescente:

```
3825  GO SUB check: IF segon THEN RETURN
```

Vamos repetir o teste com a parábola... Deu resultado! Para acabar, vamos fazer um teste mais rigoroso:

Opção	1
Janela	-.7 .7 -.3 .6
Âmbito do parâmetro	Ø 2.1
Número de saltos	300
Funções de t	SIN (11 * PI * t) COS (13 * PI * t)

Estes dados originam uma figura de Lissajous (consultar *Easy Programming,* p. 111), mas a janela foi escolhida de forma que a figura saia do visor e volte a entrar nele repetidamente. O resultado obtido é o que é apresentado pela figura 16.5, sendo bastante evidente que o programa funciona como deve ser.



F.ig. 16.5 — *Uma figura de Lissajous saindo e entrando no visor muitas vezes constitui um teste excelente.*

Projectos

A utilidade deste programa é real. Experimente usá-lo com as suas próprias escolhas de opções, janelas e funções, começando com as sugestões fornecidas abaixo. Se tiver dúvidas, utilize as funções indicadas acima, fazendo variar apenas os outros números, um de cada vez. O livro *A Book of Curves*, de E. H. Lockwood, Cambridge University Press, constitui uma boa fonte para ideias.

É evidente que ainda é possível efectuar neste programa alguns melhoramentos. Em vários locais encontram-se pedaços de código muito semelhantes que são usados mais do que uma vez — a listagem seria com certeza encurtada pelo uso de uma sub-rotina. Uma opção que permitisse voltar a RUN o programa alterando apenas uma das variáveis escolhidas também era útil. O leitor pode até ser capaz de pensar numa forma mais inteligente de abordar o problema: um dos defeitos da programação *top-down* é que, quando se começa com uma má estratégia, se tem tendência para ficar agarrado a ela até ao fim.

O leitor também pode acrescentar rotinas adicionais. Quer que os eixos sejam desenhados? Que as escalas sejam marcadas? Que sejam sobrepostas várias curvas? Eis algumas ideias de projectos, que ficam para aqueles que se sentirem inclinados para os pôr em prática.

SUGESTÕES PARA CURVAS

Opção 2: Gráfico

(*a*) Catenária: EXP t + EXP (− t)	janela	− 5	5	− 1Ø	1ØØ
(*b*) Cissóide: t/SQR (10 − t)	janela	− 5	5	− 2	2
(*c*) Parábola de Neile: (t*t)↑(1/3)	janela	− 1Ø	1Ø	− 1	1Ø
(*d*) Serpentina: 2*t/(4 + t*t)	janela	− 1Ø	1Ø	− 1	1
(*e*) Estrofóide: t*SQR((2 − t)/(2 + t))	janela	− 1.5	2	− 5	1

Opção 1: Curva parametrizada

(*f*) Coclióide:	janela	− 2Ø	2Ø	− 2Ø 2Ø
	âmbito do parâmetro	− 3Ø	3Ø	
	número de saltos	5ØØ		
t*SIN*COS t				
t*SIN t*SIN t				

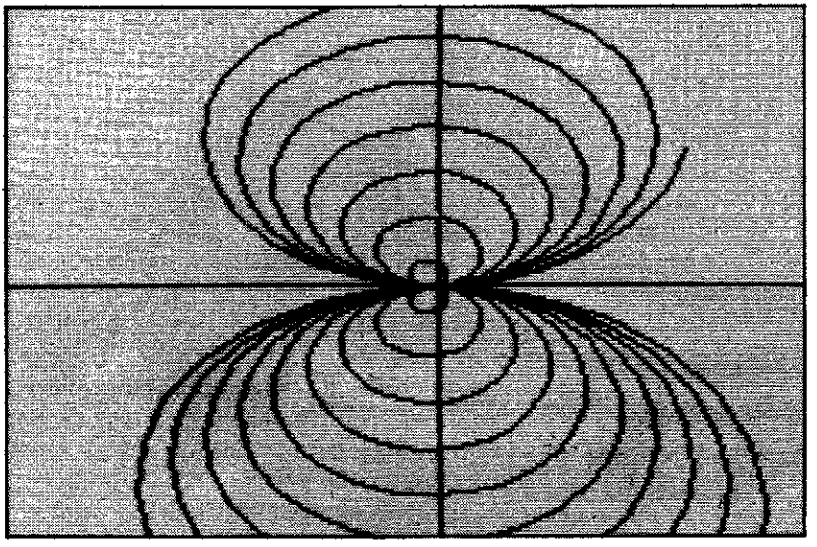


Fig. 16.6 — *A coclióide*

(g) Espiral:	janela	− 1Ø	1Ø	− 1Ø	1Ø
	âmbito	− PI	PI		
	saltos	1ØØ			
(3 + 5 * COS t) * COS t					
(3 + 5 * COS t) * SIN t					
(h) Rosácea;	janela	− 1.2	1.2	− 1.2	1.2
	âmbito	Ø	PI		
	saltos	5ØØ			
COS (11 * t) * COS t					
COS (11 * t) * SIN t					
(i) Pseudo-Lissajous:	janela	− 1Ø	1Ø	− 1Ø	1Ø
	âmbito	− 5	5		
	saltos	5ØØ			
10/(1 + t * t) * SIN (3 * t)					
10/(1 + t * t) * SIN (5 * t)					

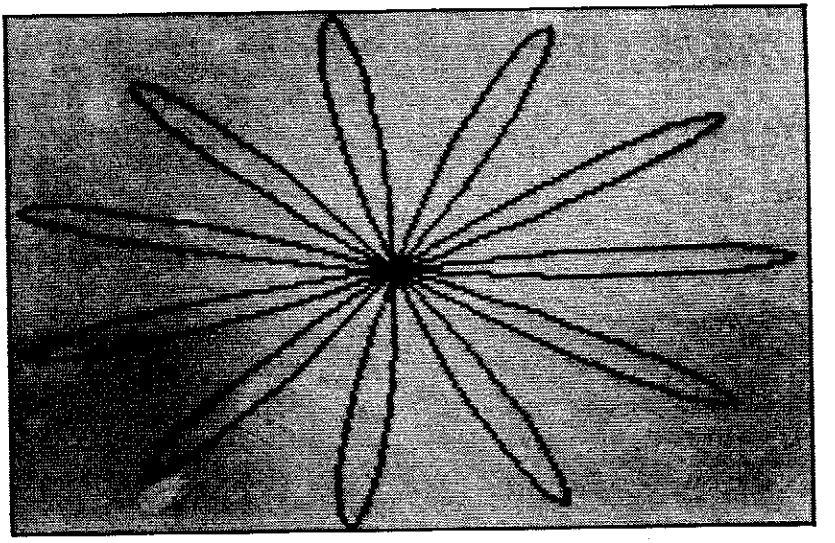


Fig. 16.7 — *Uma rosácea de onze pétalas*

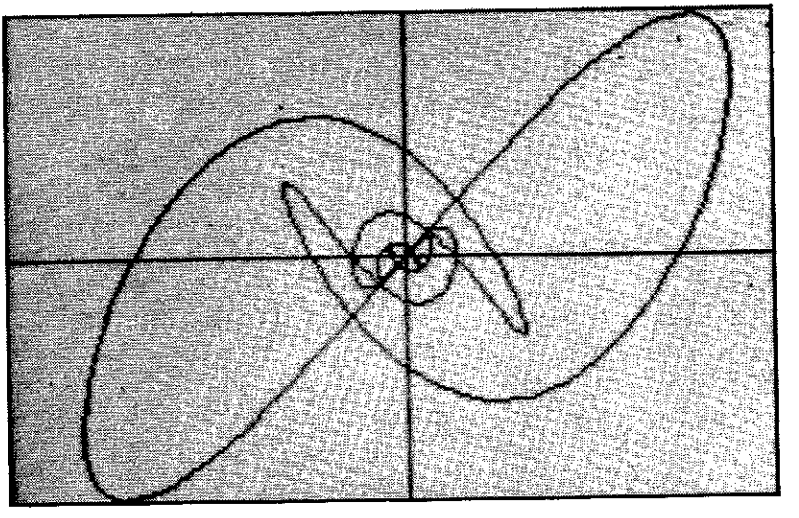


Fig. 16.8 — *Uma elegante figura de pseudo-Lissajous*

A maioria dos ficheiros têm muito em comum. Por que é que não os havemos de tratar todos da mesma maneira?

17

SISTEMAS DE GESTÃO DE DADOS

Vamos agora desenvolver um pouco mais as ideias sobre ficheiros apresentadas no capítulo 9. Nessa altura, partimos do princípio de que, antes de escrevermos os programas destinados a criar, manter e procurar entradas num determinado finheiro, tínhamos de conhecer as características exactas desse ficheiro. Contudo, todos os ficheiros vão acabar por ficar bastante semelhantes: as variações encontram-se apenas no número de campos por registo e nos comprimentos respectivos. Assim sendo, não deve ser difícil modificar *inrec*, por exemplo, substituindo as constantes todas (30, 36, etc.) por variáveis. Como fez notar um eminente cientista de computadores irlandês: «Todas as constantes que usou deviam ser variáveis.»

No entanto, é melhor esperar um bocado antes de se pôr a fazer modificações. Como é que *inrec* vai saber quais são os comprimentos dos campos num dado caso? Bom, porque não faz que *initcfs* os peça ao utilizador, durante a criação do ficheiro, ou os leia a partir do bloco cabeçalho? Como a forma do bloco cabeçalho começa a tornar-se mais complicada, vamos observá-lo mais em pormenor.

O BLOCO CABEÇALHO

Este bloco vai ser constituído basicamente por duas variáveis indexadas. Uma delas contém os nomes dados pelo utilizador a cada campo (trata-se, portanto, de uma variável indexada literal); a outra contém o número de *bytes* atribuído a cada campo. Já não precisamos de saber qual o número de *bytes* por registo, pois ele é a soma dos comprimentos de todos os campos, mas ainda precisamos do número de registos por bloco, pelo que vamos tratar de colocar essa informação na variável indexada numérica. Há ainda outro aspecto a tomar em consideração. Os campos podem conter informação tanto numérica como

literal, e é provável que o seu tratamento tenha de ser diferente consoante estejam num caso ou no outro. Dada esta circunstância, é lógico assegurarmo-nos de que esta distinção é feita quando os campos são estabelecidos. Podíamos arranjar outra variável indexada contendo esta informação, mas eu prefiro apor a cada nome de campo um «c» ou um «n», situado no primeiro *byte*, para indicar se pertence ao tipo literal (em inglês, *character*) ou ao tipo numérico. As nossas variáveis indexadas podiam portanto ter um aspecto semelhante às que são apresentadas na figura 17.1.

Neste caso, vou partir do princípio de que o ficheiro vai conter um conjunto de transacções de contas bancárias. Temos uma data constituída por seis caracteres (tratada como literal, pois não a vamos utilizar para efectuar operações aritméticas; no caso de a querer utilizar para esse fim, o melhor era ter três campos numéricos separados, um para o dia, um para o mês e um para o ano, respectivamente chamados, por exemplo, nday, nmonth e nyear), um número de cheque e uma quantia de dinheiro (ambos números) e um campo de descrição (literal), no qual pode ser colocado um número de caracteres não superior a 25 para descrever a transacção. Os números de cheques são sem-

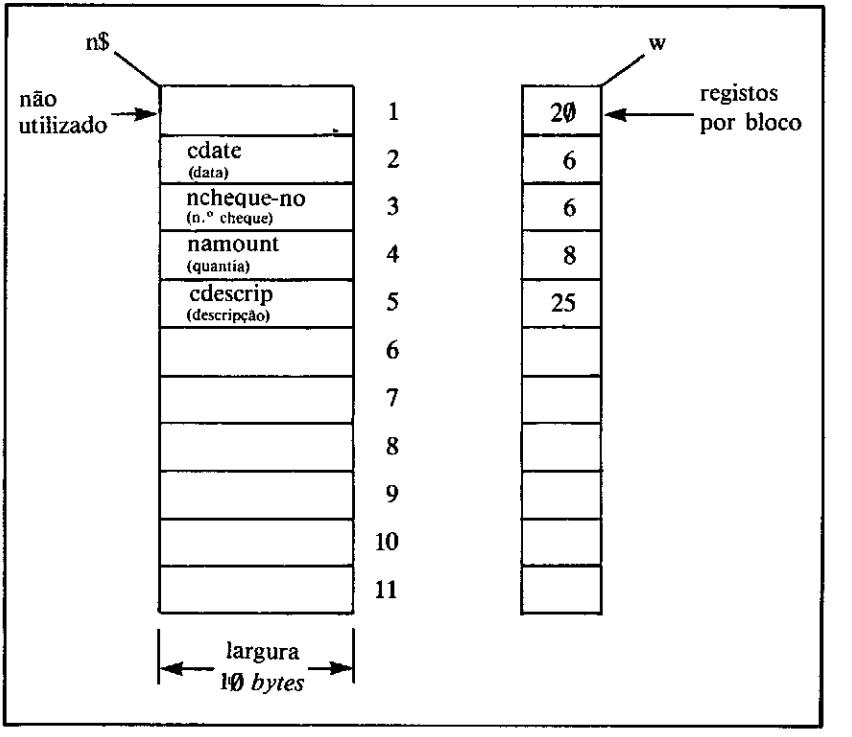


Fig. 17.1 — *Plano para o bloco cabeçalho*

pre constituídos por 6 dígitos (pelo menos, os meus são assim), e um campo para a quantia com lugar para 8 caracteres permite introduzir valores até 99999.99 libras, o que, para o *meu* balanço bancário, é muitíssimo optimista. (Repare que o ponto decimal ocupa o espaço de um carácter.) O número de registos por bloco é 2Ø; n$ (1) é deixado em branco, de modo que os índices dos nomes de campo e dos seus comprimentos se ajustem. Pessoalmente, gosto sempre que exista um pedaço de espaço desocupado, pois isso permite uma certa liberdade de manobra no caso de me ter esquecido de alguma coisa. Dado que ambas as variáveis indexadas têm 11 elementos, há lugar para 1Ø campos por registo. É evidente que, se o leitor achar esta organização um bocado restritiva, a pode alterar sem qualquer problema.

COMO ESTABELECER UM FICHEIRO

Agora, a rotina *initcfs* toma este aspecto:

```
95ØØ  DIM n$ (11, 1Ø): DIM w (11): LET ip = Ø: LET op = Ø:
      LET inbc = Ø: LET outbc = Ø: LET bpr = Ø
95Ø5  INPUT "Enter input filename:"; f$
951Ø  IF f$ = "null" THEN GO SUB set up: GO TO 9525
9514  PRINT "Start PLAY recorder"
9515  LOAD f$ + "h1" DATA n$ ( )
952Ø  LOAD f$ + "h2" DATA w ( )
9521  PRINT "Stop PLAY recorder"
9525  INPUT "Enter output filename:"; g$
953Ø  FOR p = 2 TO 11
9535  LET bpr = bpr + w (p)
954Ø  NEXT p
9545  DIM i$ (w (1), bpr): DIM o$ (w (1), bpr)
9547  IF g$ = "null" THEN RETURN
955Ø  SAVE g$ + "h1" DATA n$ ( )
9555  SAVE g$ + "h2" DATA w ( )
9557  PRINT "Stop recording": PAUSE 12Ø
956Ø  RETURN
```

Não está muito diferente do que era, exceptuando que, se introduzirmos «null» como nome do ficheiro de *input*, a rotina chama *setup*, que vai gerar uma nova descrição de ficheiro. Nos outros casos, a roti-

na vai buscar uma descrição do ficheiro aos dois primeiros blocos do ficheiro de *input* (que, para um ficheiro chamado «fred», seriam respectivamente «fredh1» e «fredh2»), vê qual é o número de *bytes* por registo (linhas 9530 a 9540) e estabelece a partir desta informação os *buffers* de entrada e de saída. Finalmente, quando o ficheiro de *output* não é «null», a rotina escreve o bloco cabeçalho.

A rotina *setup* vai ter início na linha 9400:

```
9400  INPUT "Enter no. of fields:"; nf
9405  CLS
9410  FOR p = 2 TO nf + 1
9415  PRINT AT 10, 2; "field □"; p - 1
9420  INPUT "Name of field (1st char:c/n):"; n$ (p)
9425  INPUT "No. of bytes:"; w (p)
9430  NEXT p
9435  CLS
9440  INPUT "No. of records per block:"; w (1)
9445  RETURN
```

Não há aqui grande coisa que valha a pena comentar. A rotina *setup* limita-se a executar um *loop* FOR uma vez para cada campo, indo buscar o nome de campo e o comprimento dos elementos apropriados da variável indexada. É preciso fazer uma ligeira batota para remediar o facto de o campo 1 ocupar, na realidade, o elemento 2 da variável indexada. No fim, o número de registos por bloco é especificado e colocado em w (1).

Agora, estamos no bom caminho. Podemos voltar a escrever *inrec* usando a versão para formato fixo como fonte de pistas sobre a forma de abordar esta rotina de objectivo mais geral.

O começo da rotina antiga era:

```
8000  DIM a$ (336)
```

O número 336 referia-se ao comprimento de um registo. No caso presente, esse número foi calculado por *initcfs* e colocado em bpr. Por isso, vamos ter:

```
8000  DIM a$ (bpr)
```

A linha seguinte era:

```
8010  INPUT "Artist"; a$ (TO 30)
```

Assim, em termos mais gerais, o que nos convinha era ter um *loop* no qual a linha correspondente à apresentada acima dissesse qualquer coisa do tipo:

INPUT «next field name»; a$ (beginning of field TO end of field)

Esta operação seria repetida para todos os campos.
O leitor não vai poder escrever um par de linhas semelhante a:

```
10 LET p$ = "next value"
20 INPUT p$; nv
```

A razão desta impossibilidade reside no facto de, ainda que do ponto de vista da codificação em BASIC estejam correctas, o seu significado é diferente do que nós pretendemos. A linha 20 não quer dizer: «Apresente uma mensagem cujo teor seja o que quer que esteja contido em p$ e, seguidamente, aceite um valor em nv». O que ela significa é: «Aceite um grupo de caracteres em p$ e, seguidamente, um valor em nv». Contudo, se escrever p$ *entre parênteses,* obterá o efeito desejado.

APRESENTAÇÃO NO VISOR

Apesar de a simplicidade desta organização ser muito conveniente, o utilizador corre o risco de ser confundido por ela, pois vê apenas um campo de um registo de cada vez, quando seria desejável que visse o registo *inteiro* a ser construído. Além disso, não lhe é fornecida qualquer indicação sobre o *comprimento* de campo que pode utilizar.

A solução mais fácil para este problema é construir um registo no visor através do uso de PRINT para as mensagens e copiar cada valor de INPUT para o visor de modo a ajustar-se à mensagem referente a ele. Esta técnica deve ser-lhe familiar se o seu treino foi feito num ZX81, que não permitia a utilização de mensagens literais em instruções INPUT!

Assim, neste exemplo com a conta bancária, o aspecto que queríamos que o visor apresentasse devia ser semelhante ao que apresentamos de seguida:

c:	date	060482
n:	cheque-no	317462
n:	amount	– 71.37
c:	description	Cadeira escritório: d. imp.

Aqui, os itens sublinhados são introduzidos como valores de INPUT, sendo logo de seguida apresentados como é mostrado acima. O sublinhado aparece de facto no visor, mostrando o tamanho máximo que pode ser ocupado por cada campo.

Há várias coisas que vale a pena salientar. Em primeiro lugar, o tipo de campo foi separado do nome de campo, mantendo-se apenas como uma achega à memória do utilizador. Em segundo lugar, a quantia é apresentada como sendo negativa: mais abaixo, o leitor vai compreender a utilidade desta convenção. Em terceiro lugar, o utilizador acrescentou uma nota ao campo de descrição, dizendo que se trata de um item susceptível de ser deduzido nos impostos. Esta utilização do sistema não revela muita sensatez, pois no fim do ano, quando quiser uma lista dos itens dedutíveis, vai ter de efectuar uma busca sobre parte do ficheiro. Devia ter pensado neste ponto previamente e usar um quinto campo para identificar itens dedutíveis.

Seja como for, voltemos ao nosso problema:

```
8010  CLS: LET begin = 1          [apagar o que estiver no visor de modo que a apresentação do registo não fique confusa, estabelecer um indicador para escolhermos onde começar em a$... e entrar no loop]

8020  FOR p = 2 TO 11
8025  IF n$ (p, 1) = "□" THEN GO TO 8120      [teste para o último campo]

8030  PRINT AT p, 0; n$ (p, 1); ":";
      AT p, 4; n$ (p) (2 TO)
8040  FOR c = 1 TO w (p)
8050  PRINT AT p, 14 + c; "_"
8060  NEXT c
```

[molde para impressão de campo]

Precisamos agora de saber onde começa *(begin)* e onde acaba *(end)* a parte de a$ ocupada pelo campo de que estamos a tratar:

```
8070  LET end = begin + w (p) - 1
```

Da primeira vez temos begin = 1, por isso foi estabelecido na linha 8010, e end = 6. Podemos portanto escrever:

```
8080  INPUT (n$ (p) (2 TO) ); a$ (begin TO end)
```

O que daqui resulta é uma mensagem consistindo na palavra «date» (data), sendo o valor que lhe for atribuído transferido para a$ (1 TO 6). Seguidamente, é apresentado no visor:

```
8090  PRINT AT p, 15; a$ (begin TO end)
```

Depois disto, temos de estabelecer a nova posição de «begin»:

```
8100  LET begin = end + 1
```

Agora, fechamos o *loop:*

```
8110  NEXT p
```

Finalmente, o resultado deve ser passado para r$, após o que saímos da sub-rotina:

```
8120  LET r$ = a$
8130  RETURN
```

TESTE DO PROGRAMA

Nesta altura, podemos escrever um par de rotinas de teste para vermos se está tudo a funcionar como deve ser. Antes de mais nada, precisamos de criar um ficheiro. A rotina que usamos para criar o ficheiro da colecção de discos pode servir para este efeito, sem qualquer modificação. Para refrescar a memória do leitor, aqui vai:

```
100  GO SUB initcfs
110  GO SUB inrec
120  GO SUB write
130  INPUT "Any more? (y/n)"; q$
140  IF q$ = "y" THEN GO TO 110
```

```
150 GO SUB close
160 STOP
```

Quando o leitor fizer o programa RUN, *initcfs* vai pedir-lhe um nome para o ficheiro de *input,* que o leitor vai, evidentemente, indicar ser «null». Seguidamente ser-lhe-á pedida uma descrição de registo: a da conta bancária pode ser usada, por constituir um exemplo bastante simples. O tamanho dos blocos deve ser pequeno, 5, por exemplo, de forma que o leitor não tenha de introduzir muitos registos antes de o mecanismo de bloco ser activado. Se seguir estes conselhos, conseguirá verificar tudo sem ter de dactilografar muito. Para acabar, introduza 10 ou 15 registos para criar o ficheiro de teste.

Agora precisamos de saber se os dados foram correctamente guardados. Substitua as linhas 110-160 por:

```
110 GO SUB read
120 IF r$ (TO 2) = "}" THEN STOP
130 PRINT r$
140 GO TO 110
```

Se tiver usado o formato de registo do exemplo com a conta bancária, vai obter registos do género de:

```
120882123459̄21.76 □ □ □ ananases
```

Sabemos que os primeiros 6 *bytes* representam uma data: vai portanto ser 12/08/82; a data é seguida pelo número de cheque (123459), por uma quantia (21.76 — repare nos três espaços que ficam livres, devido a, na definição de registo, haver espaço para 8 *bytes*), e, finalmente, pela descrição. Contudo, este tipo de *output* não dá jeito nenhum ao utilizador. Por isso, aquilo de que precisamos é de uma rotina que clarifique r$, apresentando os campos separados. Uma vez que esta rotina vai desempenhar uma função oposta a *inrec,* vamos chamar-lhe *outrec;* armazenamo-la a partir de 8200:

```
8200 LET begin = 1
8210 FOR p = 2 TO 11
8220 IF n$ (p) = "□" THEN PRINT: RETURN
8230 PRINT n$ (p, 1); ":"; TAB 4; n$ (p) (2 TO);
8240 LET end = begin + w (p) − 1
8250 PRINT TAB 15; r$ (begin TO end)
8260 LET begin = end + 1
```

```
8270 NEXT p
8280 PRINT: RETURN
```

Não é para admirar que esta rotina apresente uma acentuada semelhança com *inrec*. Contudo, existe entre estas rotinas uma diferença importante: desta vez, queremos que os registos vão rodando continuamente pelo visor, em vez de serem apresentados sucessivamente numa posição fixa. Se tivéssemos optado pela última solução, o leitor teria de ser extremamente rápido para conseguir entender o que quer que fosse! Assim sendo, não vamos obter o resultado que pretendemos usando «PRINT AT». Para conseguir a disposição horizontal, que anteriormente, numa instrução «PRINT AT», era indicada por uma coordenada, usei neste caso a função «TAB». Se o leitor ainda nunca usou esta função, pode pensar nela como sendo o equivalente de «PRINT AT» sem a especificação da linha. Por outras palavras, se escrever

```
PRINT AT 2, 15; . . .
```

está a indicar especificamente a linha 2; se escrever

```
PRINT TAB 15; . . .
```

não altera a sua indicação relativamente à coluna; porém, a impressão não vai situar-se na linha disponível seguinte, *qualquer que seja* a posição dessa linha no visor.

Agora, resta-nos apenas alterar a linha 130:

```
130 GO SUB outrec
```

Quando passar o programa resultante destas modificações, cada registo vai ser apresentado, de forma legível, no visor. É evidente que é pouco provável que o leitor queira que seja apresentada no visor uma lista completa do ficheiro: era mais lógico mandá-la para a impressora. Assim sendo, devíamos ter uma rotina chamada *outrec* que envie um registo para ser apresentado através da impressora. Essa rotina deve ser igual a *outrec,* excepto que todas as instruções PRINT devem ser substituídas por LPRINT.

CONCRETIZAÇÃO DO GRANDIOSO OBJECTIVO

Estamos agora de posse de todos os instrumentos de que precisamos para construir um verdadeiro sistema de gestão de dados. Vamos pois fazer uma pausa na codificação, recuar um pouco e pensar no nosso grandioso objectivo.

Implementámos três rotinas pertencentes ao nosso sistema de tratamento de arquivos original: as rotinas de criação *(create)* e de actualização *(maintain)* do ficheiro e a rotina de procura *(search)* de registos no ficheiro. Vamos ter necessidade de utilizar algumas outras rotinas para além destas; porém, desta vez, vamos ligá-las por intermédio de um *menu* que irá funcionar a partir do mesmo programa principal. Não faz muito sentido ter «*maintain*» como uma das opções, se nos vai ser imediatamente perguntado se o que queremos é «*add*» (acrescentar) ou «*delete*» (retirar) um registo. É mais prático termos directamente, como opções principais, *add* e *delete*.

Assim sendo, o nosso programa principal vai ter o seguinte aspecto:

```
10  CLS
20  PRINT AT 0, 5; "Spectrum Data Manager": GO SUB initcfs
26  CLS
30  PRINT AT 2, 0; "Options are:"
40  PRINT AT 3, 11; "1) create"
41  PRINT AT 4, 11; "2) add"
42  PRINT AT 5, 11; "3) delete"
43  PRINT AT 6, 11; "4) search"
100 INPUT "Enter option no.:"; opt
110 GO SUB 200* opt
120 GO TO 26
```

Será que estou a ouvir sussurros de descontentamento? Por acaso estará alguém a murmurar que eu ainda agora disse que íamos precisar de implementar mais algumas rotinas e afinal de contas só incluí aquelas em que já tínhamos pensado? Realmente, o leitor não deixa escapar nada... Contudo, repare como tornei fácil alinhavar aqui quaisquer rotinas novas. Basta acrescentar mais uma linha à lista de opções:

```
44 PRINT AT 7, 11; "5) o que lhe apetecer"
```

Seguidamente, colocamos a sub-rotina a partir da linha 1000, pois o mecanismo que conduz o programa principal até à sub-rotina correcta limita-se a multiplicar o número da opção por 200.

Dado isto,

create	começa em 200
add	começa em 400
delete	começa em 600

search	começa em 800
aquilo que lhe apetecer	começa em 1000

e assim sucessivamente.

Esta abordagem é bem melhor do que estar a construir o sistema de um modo completamente fixo que não permitisse nenhuma expansão. Na realidade, só nos começamos a aperceber das limitações de um dado sistema depois de o termos utilizado algumas vezes. É nessa altura que nos começa a apetecer implementar uma rotina para dominar o mundo ou seja lá para o que for. Se agir de acordo com este processo, pode modificar o programa de cada vez que se encontrar nas garras da inspiração (ou, com maior probabilidade, nas da frustração).

INICIALIZAÇÃO

É preciso chamar a atenção do leitor para mais um ponto: *initcfs* é chamada antes de qualquer das opções ser invocada. Desta forma, não temos de a inserir em cada uma das rotinas, nem de lhe responder mais do que uma vez se quisermos efectuar diversas acções sobre os mesmos ficheiros. É claro que, para serem estabelecidos ficheiros diferentes, o programa tem de ser passado de novo. Infelizmente, vai ser necessário repensar um pouco da codificação, dado que, para além da definição de nomes de ficheiros, *initcfs* faz mais algumas coisas. Esta rotina também estabelece indicadores e contadores de blocos, que *têm mesmo* de voltar a ser inicializados no princípio de uma segunda leitura do ficheiro. Assim, vamos introduzir em *cfs* uma nova sub-rotina, chamada *reset* (reestabelecer) cuja acção consiste apenas em pôr a zero estes indicadores.

```
9970  LET ip = 0: LET op = 0: LET inbc = 0: LET outbc = 0
9980  RETURN
```

Agora, a primeira linha de *initcfs* pode ser substituída por:

```
9500  DIM n$(11, 10): DIM w(11): LET bpr = 0: GO SUB reset
```

Além disso, vamos acrescentar à linha 1:

```
LET reset = 9970
```

Para acabar, podemos chamar *reset*, a partir do programa principal, antes de voltarmos à apresentação do *menu:*

```
120  GO SUB reset
130  GO TO 26
```

AS ROTINAS

Já temos a rotina *create*. Basta que a renumeremos e substituamos, no fim, STOP por RETURN.

```
200  GO SUB inrec
210  GO SUB write
220  INPUT "any more? (y/n)"; q$
230  IF q$ = "y" THEN GO TO 200
240  GO SUB close
250  RETURN
```

A rotina *add* põe-nos mais problemas. Por um lado, não foi, até agora, implementada separadamente; por outro lado, existe a sua natureza bastante primitiva, a que já fiz referência. Dado isto, vamos aproveitar esta oportunidade para repensarmos todo o problema. Para começar, precisamos de colocar todas as adições numa variável indexada; seguidamente, à medida que o ficheiro vai sendo lido, temos de comparar cada uma delas com o registo actual para ser decidido se deve ou não ser feita uma inserção. Além disso, temos de referenciar cada inserção potencial para indicar se já foi escrita. De momento, vamos ignorar esse problema. Posto isto, temos:

```
400  INPUT "how many records?"; nr
410  DIM b$ (nr, bpr)
420  FOR q = 1 TO nr
430  GO SUB inrec
440  LET b$ (q) = r$
450  NEXT q
```

estabelecimento da variável indexada

colocar em b$ os registos adicionais

A parte que se segue agora tem um aspecto um pouco rebarbativo: trata-se de identificar a parte do registo a ser usada como chave. Usando, por exemplo, o caso da conta bancária, podíamos ordenar os registos segundo a data, segundo o número do cheque ou ainda segundo a quantia. Como regra geral, queremos que o utilizador disponha da maior flexibilidade possível. Para que, neste caso concreto, isso aconteça, chamamos uma sub-rotina denominada *extractkey* (extrair chave). Mais tarde, vamos preocupar-nos com os pormenores desta sub-rotina; de momento, vamos limitar-nos a dizer aquilo que gostaríamos que *extractkey* fizesse. Assim, a acção desta rotina deve consistir em pedir ao utilizador o nome do campo chave e fornecer à rotina que a chamou três valores:

begin (princípio):	o *byte* de r$ ou b$ (p) no princípio do campo chave
end (fim):	o *byte* de r$ ou b$ (p) no fim do campo chave
ctype (tipo):	este valor vai ser 0 se a chave for numérica e 1 se a chave for literal

O facto de definirmos *extractkey* como uma sub-rotina traz a vantagem habitual de nos dar a sensação de não termos de trabalhar muito, pois, a partir do momento em que determinamos o que a rotina vai fazer, apesar de ainda não termos pensado em como é que o vai fazer, podemos começar a servir-nos dela. Contudo, se pensarmos no que se vai passar um pouco mais adiante, chegamos à conclusão de que aparece outra indicação típica de que *extractkey* deve ser uma sub-rotina: é que *delete* e *search* também vão precisar de a utilizar!

Seja como for, de momento vamos partir do princípio de que *extractkey* se encontra à nossa disposição e ver como é que a vamos usar:

```
460  GO SUB extractkey
```

ORDENAÇÃO

Seguidamente, vamos tratar de colocar pela ordem conveniente os registos adicionais. Esta ordenação não pôde ser feita antes dado que, até *extractkey* ter feito o seu trabalho, não sabíamos qual era a chave segundo a qual devíamos ordenar.

```
465  GO SUB sort
```

Podemos agora estabelecer um indicador para o primeiro registo de b$ a ser inserido no ficheiro:

```
470  LET ap = 1
```

A partir deste ponto, tudo se apresenta sem grandes dificuldades. Apenas precisamos de comparar b$ (ap) com o próximo registo do ficheiro (r$). Se r$ tiver a chave mais pequena, deve ser esse o registo enviado para *output;* caso contrário, deve ser enviado b$ (ap). Contudo, existe uma particularidade para a qual convém chamar a atenção do leitor. No caso de r$ ser *output*, precisamos de ir buscar o registo seguinte do ficheiro; no entanto, se for um elemento de b$ a ser escrito, basta-nos aumentar o valor de ap uma unidade. Em ambos os casos, temos de impedir que a leitura ultrapasse quer o fim do ficheiro, quer o fim da variável indexada. O que acontecerá quando o ficheiro de *input* ou a variável indexada que contém os registos adicionais se esgota-

rem? Infelizmente, acontecem coisas diferentes. Por isso, vamos chamar uma sub-rotina denominada *loosends* (pontas soltas) para tratar dos remates.

```
480 GO SUB read
490 IF ap > nr OR r$ (TO 2) = "}" THEN GO SUB loosends:
    RETURN
```

Agora queremos comparar as chaves de r$ e b$ (ap). Antes de mais nada, vamos extraí-las e colocá-las, respectivamente, em r$ e t$:

```
500 LET s$ = r$ (begin TO end): LET t$ = b$ (ap) (begin TO end)
```

Vamos então agora compará-las. Mas aparece-nos um problema: s$ e t$ podem ser chaves numéricas ou literais. No caso de se tratar de chaves numéricas mas encaradas como se fossem literais, 123 não vai ser o mesmo que 123.0; e, o que ainda é pior, 5 vai ser considerado maior do que □□12!

Por isso, vamos precisar de mais duas sub-rotinas, *compn* e *compc*, que executam, respectivamente, comparações numéricas e literais. Tanto uma como outra vão apresentar um valor chamado comp que identifica o resultado da comparação de acordo com o quadro abaixo.

Valor de comp	*Significado*
−1	s$ < t$
0	s$ = t$
1	s$ > t$

O leitor pode estar a magicar na razão que me levou a especificar condições de "=" e ">", quando apenas precisamos de "<". Bom a razão por que o fiz é que estava a pensar também nas rotinas *delete*, *search* e *sort* (ordenar), que vão igualmente utilizar comparações, embora, provavelmente, o façam de formas diferentes desta. É, portanto, melhor tornar as rotinas tão gerais quanto for possível.

Podemos agora escrever:

```
510 IF ctype THEN GO SUB compc
520 IF NOT ctype THEN GO SUB compn
```

É possível que o leitor nunca tenha encontrado este tipo de construção (a não ser que tenha lido o capítulo 3). Não parece que o «IF» vá testar seja lá o que for. O que está em jogo aqui é que o valor atri-

buído a uma condição incluída numa declaração «IF» vai ser Ø ou 1, consoante a condição for falsa ou verdadeira. Para concretizar, escrevemos, por exemplo:

```
75  IF a = b THEN . . .
```

Neste caso, a = b vai ser substituído por 1 se os valores forem iguais; caso contrário, será substituído por Ø. Uma vez que os valores atribuídos a ctype são precisamente Ø ou 1, poupamos desta forma um passo na avaliação, e o resultado ten um aspecto mais agradável do que "IF ctype = Ø THEN GO SUB compn".

Vamos agora testar comp. Se o seu valor for menor do que zero, queremos que r$ seja enviado para *output,* sendo substituído pelo registo seguinte do ficheiro:

```
53Ø  IF comp < Ø THEN GO SUB write: GO TO 48Ø
```

Se o valor de comp não preencher a condição referida, queremos que seja enviado para *output* o registo adicional de que estivermos a tratar na altura. Porém, não nos podemos esquecer de que só é possível *output* o que se encontra em r$, tendo-o seu conteúdo actual de ser guardado e chamado de novo antes e depois desse processo:

```
54Ø  LET t$ = r$: LET r$ = b$ (ap): GO SUB write: LET r$ = t$:
     LET ap = ap + 1: GO TO 49Ø
```

Repare, no fim da linha, em "GO TO 49Ø". Não voltamos á linha 48Ø porque ainda não mandámos embora o registo do ficheiro de que temos estado a tratar; é evidente que não precisamos de outro!

MAIS SUB-ROTINAS

Agora vem a parte mais difícil. As sub-rotinas condescendentemente chamadas a partir da rotina *add* vão ter de ser escritas. Vamos atribuir aos seus pontos de partida os números de linhas que se seguem, e que, como é óbvio, têm de ser inicializados no princípio do programa:

extractkey	78ØØ
loosends	76ØØ
compc	74ØØ
compn	72ØØ
sort	7ØØØ

Antes de mais nada, *extractkey* tem de perguntar ao utilizador qual é o campo chave:

```
7800  DIM k$ (9): INPUT "Enter key field name"; k$
```

A variável literal k$ vai conter um nome de campo a partir do segundo *byte*. Por outras palavras, k$ *não vai* incluir o carácter do tipo. Parece mais natural introduzir «amount» do que «namount»; de qualquer forma, o «n» não é essencial, uma vez que já temos essa informação em n$.

Agora, vamos ter de percorrer a variável indexada n$ em busca de k$, e, além disso, temos também de saber a cada momento quantos *bytes* é que já andámos. Para isso, vamos adoptar o seguinte procedimento: partimos do princípio de que o campo que estamos a procurar é o primeiro, sendo portanto begin = 1; no caso de o campo chave não ser este, incrementamos begin w(2) unidades, de modo que passe a indicar o início do segundo campo; se ainda não for este o campo que queremos, voltamos a incrementar begin, desta vez w(3); e prosseguimos desta forma até o campo chave ser encontrado:

```
7810  LET begin = 1
7820  FOR p = 2 TO 11
7830  IF n$ (p) (2 TO) = k$ THEN GO TO 7860
7840  LET begin = begin + w (p)
7850  NEXT p
7860  LET end = begin + w (p) - 1
```

(nota: o comprimento de k$ tem de ser 9 *bytes*, pois vai ser feita uma comparação com os últimos 9 *bytes* de cada elemento de n$)

Podemos agora identificar o tipo do campo vendo o teor do primeiro *byte* do elemento de n$ indicado por p:

```
7870  IF n$ (p, 1) = "c" THEN LET ctype = 1
7880  IF n$ (p, 1) = "n" THEN LET ctype = 0
7890  RETURN
```

Na verdade, foi bastante fácil, não acha?

Agora vamos tratar de *loosends*. Se chegarmos primeiro ao fim do ficheiro, temos de esvaziar o *buffer* das adições; caso contrário, temos de copiar o resto do ficheiro. Portanto, vai ser bastante simples:

```
7600  IF ap > nr THEN GO SUB cpyfile: RETURN
7610  FOR p = ap TO nr
7620  LET r$ = b$ (p)
7630  GO SUB write
```

```
7640 NEXT p
7650 GO SUB close
7660 RETURN
```

Colocando *cpyfile* a partir de 7700, temos:

```
7700 GO SUB write
7710 GO SUB read
7720 IF r$ (TO 2) = "}}" THEN GO SUB close: RETURN
7730 GO TO 7700
```

As sub-rotinas *compc* e *compn* são muito fáceis:

```
7400 IF s$ < t$ THEN LET comp = -1
7410 IF s$ = t$ THEN LET comp = 0
7420 IF s$ > t$ THEN LET comp = 1
7430 RETURN

7200 IF VAL s$ < VAL t$ THEN LET comp = -1
7210 IF VAL s$ = VAL t$ THEN LET comp = 0
7220 IF VAL s$ > VAL t$ THEN LET comp = 1
7230 RETURN
```

A ROTINA DE ORDENAÇÃO

Por fim, precisamos da rotina *sort*. Se leu o livro *Easy Programming*, deve lembrar-se de que introduzi nos capítulos sobre a correcção de gatos um algoritmo de ordenação. Aqui, vamos utilizá-lo de novo. Não se pode dizer que seja o processo de ordenação mais elegante que jamais foi inventado (muito pelo contrário, aliás); contudo, tem a vantagem de *ser simples*, e, como b$ não é uma variável indexada muito grande, não vai levar demasiado tempo a executar.

```
7000 LET inc = 1
7005 LET flag = 0
7010 FOR p = 1 TO nr - inc
7020 LET s$ = b$ (p) (begin TO end): LET t$ = b$ (p + 1)
     (begin TO end)
7030 IF ctype THEN GO SUB compc
7040 IF NOT ctype THEN GO SUB compn
```

```
7050  IF comp > 0 THEN LET t$ = b$ (p): LET b$ (p) = b$ (p + 1):
      LET b$ (p + 1) = t$: LET flag = 1
7060  NEXT p
7070  IF flag > 0 THEN LET inc = inc + 1: GO TO 7005
7080  RETURN
```

Bom, em retrospectiva, vemos agora que *add* nos deu bastante trabalho. No entanto, é capaz de ser a mais difícil das rotinas do *menu*.

«DELETE»

Escrever a rotina *delete* vai permitir-nos descansar um bocado:

```
600  INPUT "How many keys"; nr
610  GO SUB extractkey
620  DIM d$ (nr, end - begin + 1)
630  FOR p = 1 TO nr
640  INPUT "Enter deletion (key only):"; d$ (p)
650  NEXT p
660  GO SUB read
670  IF r$ (TO 2 ) = "}" THEN GO SUB close:
     RETURN
680  FOR p = 1 TO nr
690  IF r$ (begin TO end) = d$ (p)
     THEN GO TO 660
700  NEXT p
710  GO SUB write
720  GO TO 660
```

(Encontrar os limites do campo chave e usá-los para dimensionar uma variável indexada de tamanho adequado.)

Estabelecer a variável indexada de chaves a serem retiradas de r$.

Procurar uma convergência entre a chave de r$ e um elemento de d$; se for encontrada, ir buscar o próprio registo.

Como não foi encontrada nenhuma convergência, o registo é escrito... seguidamente, vamos buscar outro.

Era bom que fosse sempre assim...

«SEARCH»

Antes de nos aventurarmos por caminhos perigosos, devíamos pensar seriamente na forma como a rotina *search* deve actuar. O mais fácil seria permitir que o utilizador introduzisse uma única chave, sendo subsequentemente lido o ficheiro e apresentados no visor todos os

registos que contivessem essa chave. Para ver se este método é adequado, tente colocar-se na posição de um utilizador e imaginar o tipo de perguntas que ele gostaria de pôr. Para termos um exemplo conveniente, vamos voltar ao ficheiro da conta bancária. No caso de existir um campo que indique se um determinado item pode ser deduzido nos impostos, o utilizador pode perfeitamente querer que sejam apresentados todos os registos que tenham esse campo afirmativamente preenchido. Por outro lado, pode querer que sejam apresentados todos os registos referentes a cheques passados para uma quantia excedendo 200 libras, ou a cheques sujo número seja superior a 318472, ou ainda a cheques passados entre 8 de Julho de 1981 e 5 de Setembro de 1981. Por outras palavras, é muito provável que lhe interesse considerar um *espectro* de chaves preferencialmente a uma única. Devemos pois permitir-lhe ambas as possibilidades. Além disso, será que o utilizador vai *sempre* querer que sejam apresentados no visor os registos encontrados por esta rotina? Parece pelo menos igualmente provável que queira que sejam impressos. No entanto, existe ainda outra possibilidade que vai aumentar imenso a utilidade do nosso sistema sem que seja necessário esforçarmo-nos significativamente (bom, sem que seja necessário esforçarmo-nos significativamente mais): trata-se de permitir que os registos encontrados pela busca sejam escritos num novo ficheiro. Desta forma, podem ser efectuadas buscas sucessivas para isolar combinações de condições. Por exemplo, se precisarmos de uma lista de todos os itens que possam ser deduzidos no imposto cujo valor seja superior a 100 libras, geramos em primeiro lugar um novo ficheiro constituído pelos itens dedutíveis e, seguidamente, extraímos dele todos os itens acima de 100 libras.

Tal como eu disse, este objectivo não é difícil de realizar. Uma vez que se tenha encontrado um registo a tratar, basta chamar *outrec* para o apresentar no visor, *loutrec* para o imprimir ou *write* para o colocar num ficheiro, de acordo com o que se pretender.

Vamos pois a isso:

```
800  GO SUB extractkey
810  IF ctype THEN DIM k$ (end - begin + 1): INPUT "Target
     key:"; k$
820  IF NOT ctype THEN INPUT "Key range—low:"; low,
     "high:"; high
```

Começa a ser necessário fornecer algumas explicações. É evidente que a primeira tarefa é descobrir qual vai ser o campo usado como chave para a busca. É esta a razão da chamada a *extractkey*. Se a chave escolhida for um campo literal, a ideia de espectro não tem significado (a não ser que queira usar um espectro de chaves alfabéticas,

como, por exemplo, os nomes entre BROWN e ROGERS, mas não vou tratar desse caso aqui; se precisar de o utilizar, é suficientemente fácil para que o planeie sozinho). Assim, pedimos apenas uma única chave de registo a tratar (*target key* — chave alvo); o comprimento desta chave tem de se ajustar ao comprimento do campo correspondente no registo, o que justifica o DIM. No entanto, se a chave for numérica, pedimos um espectro de chaves.

Seguidamente, precisamos de saber qual vai ser a opção de *output* seleccionada:

```
830 INPUT "Display (1), Print (2) or File (3):"; opt
```

Podemos agora começar a procurar os registos:

```
840 GO SUB read
850 IF r$ (TO 2) = "}}" AND opt = 3 THEN GO SUB close:
    RETURN
860 IF r$ (TO 2) = "}}" THEN INPUT "Enter c to continue";
    k$: RETURN
```

O leitor acaba de ver um bocado de codificação não muito elegante. O problema está em que, ao ser identificado o fim do ficheiro, existem duas possibilidades: se não houver ficheiro de *output*, queremos apenas voltar ao *menu*, contudo, no caso de existir, temos de o fechar primeiro. As alternativas que se apresentavam eram saltar do *loop* quando fosse identificado o indicador de fim de ficheiro e testar então a opção 3, ou chamar uma sub-rotina que tratasse das duas condições. Pessoalmente, não gosto de saltar a não ser que não tenha realmente outra solução (os saltos podem originar codificações que, muito caritativamente, podem ser qualificadas de barrocas); quanto à sub-rotina, neste caso pareceria um bocado excessiva. (O problema surge porque o BASIC do Spectrum não tem uma cláusula ELSE na sua declaração IF. Algumas variantes do BASIC permitem que se diga: IF isto THEN aquilo ELSE o outro. Às vezes, isto é bastante útil, pois o equivalente no Spectrum consiste em usar duas declarações IF. Já aconteceu diversas vezes, mas, até aqui, este é o exemplo mais desajeitado.)

COMPARAÇÕES

Bom, basta de chinesices. (O BASIC é realmente encantador, afirmo-o com toda a sinceridade...)

```
870 IF ctype THEN GO SUB ckeytest
880 IF NOT ctype THEN GO SUB nkeytest
```

Neste caso, vamos *mesmo* precisar de mais um par de sub-rotinas, dado que o método de tratamento das comparações literais e numéricas vai ser significativamente diferente. A sub-rotina *ckeytest* vai fazer a comparação das variáveis literais, enquanto a comparação das variáveis numéricas vai ser feita pela sub-rotina *nkeytest*. Para voltarem, precisam apenas de um valor, «match» (correspondência), que vai ser 1 se tiver sido encontrada uma correspondência e Ø no caso contrário. Temos então:

```
890  IF NOT match THEN GO TO 840          Ir buscar o próximo
                                          registo.
895  LET bs = begin: LET es = end
900  IF opt = 1 THEN GO SUB outrec        ┐
910  IF opt = 2 THEN GO SUB loutrec       ├ Enviar o registo para
920  IF opt = 3 THEN GO SUB write         ┘ o dispositivo de saída apropriado.
925  LET begin = bs: LET end = es
930  GO TO 840                            Ir buscar o próximo
                                          registo.
```

Repare que temos de guardar «begin» e «end», pois são alterados por *outrec*.

Temos agora de escrever duas pequenas rotinas:

ckeytest 6800
nkeytest 6600

A mais fácil é *ckeytest*, pois não precisamos de nos preocupar com nenhum espectro, dado que existe apenas uma chave única:

```
6800  LET match = Ø
6810  IF k$ = r$ (begin TO end) THEN LET match = 1
6820  RETURN
```

Tratemos agora de *nkeytest:*

```
6600  LET match = 1
6610  IF VAL r$ (begin TO end) < low OR
      VAL r$ (begin TO end) > high THEN LET match = Ø
6620  RETURN
```

Em *ckeytest* parti do princípio de que não existia correspondência, atribuindo depois a match o valor 1 no caso de existir, enquanto em *nkeytest* parti do princípio de que a correspondência *existia*, sendo o valor Ø atribuído a match apenas no caso de a chave do registo testado

ser menor do que a chave mais baixa do espectro ou maior do que a sua chave mais alta.

Neste momento, tenho de fazer uma confissão. A rotina *delete* só funciona de facto com chaves do tipo c, porque a linha 69Ø faz uma comparação de variáveis literais. É possível que, na altura, o leitor tenha ficado espantado com este facto.

A razão que me levou a fazer isso é que *search* fornece uma forma mais poderosa de retirar chaves numéricas. Afinal de contas, procurar um espectro de chaves é exactamente o oposto de as retirar (usando a opção «write» em *search*), de modo que, para retirar todos os registos com chaves menores do que 5Ø, basta-nos procurar os registos com chaves maiores ou iguais a 5Ø. É possível arranjar testes de espectro mais complexos, bastando para isso expandir *nkeytest* de forma a incluir um segundo espectro, de low2 a high2, por exemplo. Desse modo, o leitor poderia retirar todos os registos com chaves compreendidas entre dois valores. (Isso também pode ser feito com a implementação habitual, mas tem como resultado a geração de dois subficheiros.)

MAIS FUNÇÕES

Que outras opções poderiam ser necessárias, ou, pelo menos, desejáveis? Bom, que tal *list,* que copiaria o ficheiro de *input* completo? Podia ser útil, mas geralmente conseguimos passar sem ela usando *search,* escolhendo uma chave numérica e dando um espectro que sabemos exceder o espectro dos valores que de facto existem no campo da chave. Como conhecemos o comprimento do campo, temos a garantia de poder fazê-lo. É possível que o leitor pense que não é razoável esperar que o utilizador execute estes malabarismos, e eu sinto-me inclinado a concordar consigo; contudo, este conjunto de rotinas está a começar a adquirir proporções bastante avantajadas, e, se apenas dispuser de uma memória de 16K, o tamanho dos *buffers* dos seus ficheiros estará a começar a encolher. Posto isto, qualquer razão que nos pudesse levar a querer dispor de mais opções teria de ser muito boa. É evidente que, se o leitor tiver um aparelho de 48K, pode continuar a inventar rotinas novas e cada vez mais esotéricas, conforme lhe der na real gana.

No entanto, há ainda mais uma rotina que deve sem dúvida estar presente: devíamos poder somar os conteúdos de um dado campo numérico de cada registo. Lembra-se de que, aquando da discussão do exemplo da conta bancária, sugeri que os débitos fossem introduzidos como negativos e os créditos como positivos? Se pudéssemos somar todos esses campos, teríamos um balanço actual. Não há dúvida de que se trata de uma facilidade muito útil!

Vai ser esta a nossa opção 5, por isso vamos precisar de ter:

```
44  PRINT AT 7, 11; "5) sum"
```

A rotina vai portanto partir da linha 1000. Vamos começar por inicializar a variável sum (soma):

```
1000  LET sum = 0
```

Seguidamente, vamos ver qual é a chave com que vamos trabalhar:

```
1010  GO SUB extractkey
```

Verificamos agora se a chave é numérica. Caso não seja, não a podermos usar aritmeticamente:

```
1020  IF ctype THEN PRINT "Key not numeric": PAUSE 120:
      RETURN
```

Agora, falta-nos apenas ler cada um dos registos do ficheiro, adicionando sucessivamente os conteúdos do campo chave ao valor que se encontra em sum:

```
1030  GO SUB read
1040  IF r$ (TO 2) = "}" THEN PRINT "Sum of □"; k$; "= □";
      sum: INPUT  "Enter c to continue"; k$: RETURN
1050  LET sum = sum + VAL r$ (begin TO end)
1060  GO TO 1030
```

Aqui, há apenas um ponto que necessita de comentário. Se fosse executado um RETURN imediatamente a seguir ao PRINT da linha 1040, o leitor precisaria de ser muito rápido para conseguir ver o que quer que fosse, por causa do CLS, que está à espreita no início da apresentação do *menu*. Assim, a declaração INPUT serve apenas para fornecer uma forma de reter o sistema até o leitor estar pronto para prosseguir. A instrução PAUSE 120, na linha 1020, encontra-se aí pelo mesmo tipo de razão. De facto, podíamos utilizar PAUSE em ambos os casos, dado que PAUSE 65535 retém o aparelho durante cerca de 21 minutos, ou até se carregar nalguma tecla, de modo que, para todos os efeitos práticos, o resultado é o mesmo.

PÔR TUDO EM ORDEM

A nossa excursão pelo mundo dos ficheiros em computador está prestes a terminar. Como já fiz notar, não vai ser difícil acrescentar novas rotinas ao sistema de gestão de dados, nem tão-pouco rever as rotinas já existentes de modo a adaptá-las a pequenas modificações das nossas necessidades. Por exemplo, quando o leitor arranjar *microdrives,* deve ser-lhe muito fácil rever o sistema de forma adequada; o programa resultante vai, evidentemente, ser muito mais fácil de utilizar, dado que será o próprio Spectrum a tratar do controlo de discos.

No entanto, *software* perfeito é coisa que não existe, ainda que os termos de comparação sejam limitados, e não pretendo de forma alguma pugnar pelos méritos deste programa. Vou limitar-me a fazer algumas sugestões sobre revisões cujo efeito seria o de dar às coisas um aspecto um pouco mais ordenado, o que talvez agrade ao leitor.

Em primeiro lugar, há um ponto referente ao próprio *cfs* que pode ser melhorado. É necessário que o utilizador esteja a par do delimitador "}}". Passamos a vida a escrever:

```
IF r$ (TO 2) = "}}" THEN . . .
```

Esse teste podia ser incluído em *read* usando uma variável chamada *eof* (de «end of file» — fim de ficheiro) que é posta a zero em *initcfs,* sendo-lhe depois, em *read,* atribuído o valor 1 no caso de os primeiros dois *bytes* de r$ serem "}}". Assim, num programa do utilizador, poderia ser usado:

```
IF eof THEN . . .
```

Esta modificação facilita um pouco e dá muito melhor aspecto.

Em segundo lugar, as mensagens fornecidas para indicar ao utilizador quais as operações de controlo dos gravadores que é necessário efectuar são uma confusão. Tinha sido muito melhor escrever quatro sub-rotinas, *recon, recoff, playon* e *playoff,* de maneira que fossem sempre geradas mensagens iguais em circunstâncias idênticas. Além desta vantagem, tornar-se-ia mais fácil rever as mensagens, caso fosse necessário, ou mesmo substituí-las por sinais a serem enviados para um porto, de forma que os gravadores de *cassettes* fossem controlados directamente, como já foi sugerido (para pormenores sobre este assunto, ver o apêndice B). Algumas revisões simples das mensagens poderiam incluir: torná-las faiscantes para chamarem mais a atenção; acrescentar um BEEP, pela mesma razão; acrescentar uma PAUSE, de modo que o RETURN para o *menu* não apagasse a mensagem tão depressa.

Em terceiro lugar, não é feita em *setup* nenhuma verificação que

garanta que os nomes dos campos começam com «c» ou com «n». É óbvio que esta circunstância tem uma importância vital, pois o resto do sistema parte do princípio de que se dá um destes casos. Existem outras situações do mesmo tipo, para as quais deviam ter sido incluídos testes. (Que acontecerá, por exemplo, se o utilizador fornecer a *extractkey* um nome de campo inexistente?)

Por último, na situação presente não é na realidade possível sair do *menu*. É evidente que isso é fácil de conseguir arranjando uma opção 6 (*exit* — saída) e acrescentando a linha 1200 STOP. Ainda não o tinha feito porque, como é lógico, de cada vez que se acrescenta uma nova opção, o valor da opção de saída sobe uma unidade, e a declaração STOP avança duzentas linhas.

ORDENAÇÃO DE UM FICHEIRO

Para terminar esta secção, vou deixar um problema para ser resolvido *pelo leitor*.

No nosso gestor de dados existe uma flagrante omissão. Qualquer sistema que se preze deve permitir que um ficheiro seja reordenado de acordo com uma nova chave, de tal forma que, por exemplo, seja possível pegar num ficheiro ordenado alfabeticamente segundo o campo do «nome» e reordená-lo automaticamente segundo, digamos, o campo do «n.º de telefone».

Evitei este problema dado que não é muito fácil de resolver usando apenas uma fita de *input*. Nos tempos em que as grandes instalações de computadores utilizavam habitualmente fita magnética, era frequentemente usada para resolver este problema uma técnica pomposamente chamada «polyfhase merge sort» («ordenação por fusão polifásica»).

O que se passava era que se tinham duas fitas de *input* e duas fitas de *output*. Era lido um registo de uma fita e seguidamente comparado com registos da outra, que eram escritos numa das fitas de *output* até aparecer um cuja chave fosse maior do que a do registo de referência. Nessa altura, trocavam-se as fitas de *input* e as de *output* e repetia-se o processo. Quando ambos os ficheiros de *input* tivessem sido completamente lidos, os ficheiros de *output* eram usados como um novo conjunto de ficheiros de *input*, e todo o processo era repetido. Por fim, após uma data de repetições, apenas ficava escrito um dos arquivos de *output*, nunca se chegando a verificar a troca de fitas. Nessa altura, sabíamos que o ficheiro estava ordenado.

Parece-lhe complicado? A mim também. Agora o leitor já sabe porque é que até agora não me ocupei deste problema. (Apesar de, de um certo ponto de vista, a rotina *add* usar uma espécie de ordenação por fusão, utilizando o ficheiro de *input* e o *buffer* de adição como

equivalentes das duas fitas de *input*; a diferença está em que os nossos dois *inputs* têm a garantia de estar previamente ordenados.)

Posto isto, *será possível* fazer a ordenação de um ficheiro em fita com uma única fita de *input?* Bom, de facto é, existindo para esse efeito diversos algoritmos bastante conhecidos. No entanto, há um que descobri recentemente, e que vai servir lindamente os nossos objectivos, pois nos possibilita o uso da estrutura de blocos que faz parte do *cfs*. (Digo «descobri» em vez de «inventei» porque, ainda que nunca tenha encontrado impressa uma descrição desta técnica, ela é tão simples que tenho a certeza de que já deve ter sido usada antes.)

ORDENAÇÃO «RODA DE ENGRENAGEM»

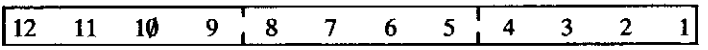
O funcionamento desta técnica é o seguinte:

Introduzimos o ficheiro de *input*, um bloco de cada vez, ordenando cada bloco antes que seja escrito no ficheiro de *output*. Se nos limitássemos a fazer isto não aconteceria nada de especial, dado que, ainda que o ficheiro estivesse localmente ordenado, podia existir no fim do ficheiro uma chave baixa que não poderia recuar para antes do princípio do último bloco por muitas vezes que o processo fosse repetido. Na realidade, não acontece *nada* de novo se usarmos esta técnica para reordenar o ficheiro de *output*. A razão desta falha está, evidentemente, no facto de as separações entre os blocos permanecerem no mesmo lugar, de tal forma que os registos se podem movimentar apenas dentro dos blocos e não entre eles. Se pudéssemos alterar as posições dos limites dos blocos...

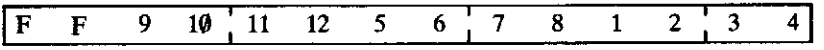
Na verdade, é fácil consegui-lo. Antes de o primeiro bloco ser transferido para o ficheiro de *output*, introduzimos nele alguns registos fingidos. Para que dê melhor resultado, o seu número deve ser metade do número de registos por bloco. Quando o ficheiro, com as suas novas posições, voltar a ser lido, os limites dos blocos estão a meio caminho entre as suas anteriores localizações, e esta «sobreposição» permite que a ordenação avance um pouco mais. Temos de nos assegurar de que a ordenação vai ignorar os registos fingidos e de que, no *output*, eles são suprimidos, de modo que as posições dos limites sejam de novo mudadas, para que possa ocorrer uma nova operação de ordenação. Na fase de ordenação seguinte, voltam a ser introduzidos os registos fingidos, e assim sucessivamente. Sabemos que a ordenação está completa quando deixarem de se verificar trocas de posições de registos.

Vai ser apresentado agora um exemplo que ilustra o princípio acima exposto. Para maior simplicidade, vou escrever apenas as chaves,

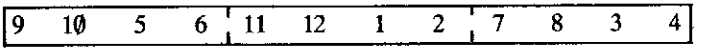
que vão ser os números 1-12, pela ordem inversa. O tamanho dos blocos que vamos utilizar é 4:



1.° passo: ordenação dos blocos e inserção de dois registos fingidos:

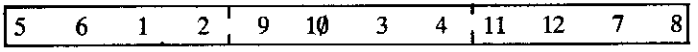


2.° passo: ordenação dos blocos e supressão dos registos fingidos:

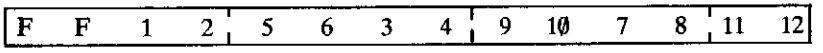


3.° passo: ordenação dos blocos e inserção dos registos fingidos:

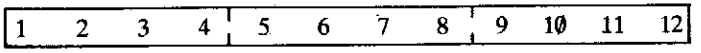
4.° passo: ordenação dos blocos e supressão dos registos fingidos:



5.° passo: ordenação dos blocos e inserção dos registos fingidos:



6.° passo: ordenação dos blocos e supressão dos registos fingidos:



E já está!

Desde que haja suficiente memória disponível, não há razão para que o bloco usado para a ordenação seja o *buffer* de *input*, pelo que não há razão para que esteja restringido ao tamanho de blocos do ficheiro. É evidente que o número de passos necessários para a efectuação da ordenação diminui à medida que o tamanho dos blocos aumenta.

Deixo ao leitor a codificação deste problema.

Se o leitor pensa que Canopus é uma lata de comida para gato[1]*, o melhor que tem a fazer é ler este capítulo.*

18

MAPAS DAS ESTRELAS

Este item implica o gasto de um bom bocado de tempo na introdução dos dados, especialmente se o leitor decidir ampliá-lo. Assim, não comece a não ser que tenha à sua frente um bom par de horas, porque é uma maçada ter de parar a meio.

A nossa ideia é escrever um programa que marque no visor imagens de diversas constelações: Orion, Cygnus, Gemini, etc. Como opções, vou incluir:

1. A apresentação sequencial automática das imagens, indicando o seu nome;
2. A busca de uma dada constelação a partir do nome e a sua marcação no visor.
3. Um teste: o computador desenha uma constelação e o utilizador tem de lhe dar o nome correspondente.

Nesta ilustração, vou introduzir no aparelho apenas seis constelações; contudo, o programa vai ser estabelecido com capacidade para vinte. Se quiser ainda mais constelações, SAVE os dados em fita, redimensione as variáveis indexadas usadas para os conter e volte a LOAD os dados. (Vai ter de pensar nisto com mais pormenor, mas esta é a ideia geral.)

ESTRUTURA DOS DADOS

Desligue o computador durante um minuto, pois antes de mais nada vamos ter de pôr a cabeça a funcionar. «Programar primeiro e fazer as perguntas depois» é uma receita infalível para o fracasso.

---

[1] Trocadilho intraduzível, baseado na semelhança sonora entre «Canopus» e «Can o' Puss», que significa «a lata do gatinho». *(N. da T.)*

Vamos precisar de dados para:

1. Nome latino da constelação (Ursa Major, etc.).
2. Nome português da constelação (Ursa Maior, etc.).
3. Posições das estrelas (montes de coordenadas para PLOT).
4. Magnitude (grau de brilho) de cada estrela.

Podíamos continuar (por exemplo, indicar a cor da estrela, adequadamente exagerada para obtermos uma apresentação bonita) mas, para começar, chegam estes dados.

Pondo de lado, de momento, o problema de conseguir ter acesso às informações requeridas e de as introduzir no aparelho, temos de decidir qual vai ser o formato a dar aos dados. É evidente que queremos colocar os nomes em variáveis indexadas; assim, para 2Ø constelações diferentes, precisamos de estabelecer duas variáveis indexadas literais cujo tamanho seja 2Ø, por exemplo n$ para o nome latinho e e$ para o nome português.

Também vamos querer imprimir coisas do tipo

"Cignus" (o Cisne)

O problema com as variáveis indexadas literais está no facto de todas as variáveis literais que as constituem terem um comprimento fixo. Suponhamos que estabelecemos o comprimento 12, para haver espaço para nomes como, por exemplo "Camelopardus" (a Girafa) e que "Cignus" é o primeiro item. Não é muito satisfatório usar a instrução

PRINT n$ (1); "□ (the □"; e$ + ")"

O resultado seria qualquer coisa do tipo:

Cygnus □□□□□□ (Cisne □□□□□□)

Haveria espaços por todo o lado.

Existe um truque útil que permite evitar isto. Acrescentamos a cada variável literal um carácter final, o décimo terceiro, cujo *código* indica qual vai ser o comprimento real que queremos utilizar. Assim, «Cygnus» será introduzido como:

Cygnus □□□□□□☆

A estrela é na realidade o carácter cujo código é 6, o comprimento de «Cignus». (Trata-se do carácter de *controlo* PRINT vírgula, mas, desde que não o tentemos imprimir, tudo corre bem.) Para imprimirmos o nome «Cignus», utilizamos:

```
PRINT n$ (1) (TO CODE n$ (1, 13) )
```

Conseguimos desta forma omitir os espaços indesejados.

É evidente que utilizamos o mesmo truque com os nomes portugueses. Temos de ser cuidadosos ao escrever as rotinas de *input* e *output*, para que esta característica seja tomada em conta.

A maneira óbvia de estabelecer as coordenadas das estrelas de uma constelação e as respectivas magnitudes é uma variável indexada. Precisa de ter uma dimensão 3 para conter a coordenada horizontal, a coordenada vertical e a magnitude; uma dimensão 20 para poder conter até 20 estrelas por constelação e uma dimensão 20 (por exemplo) para que seja possível introduzir até esse número de constelações. Isso leva a uma instrução.

```
DIM p (3, 20, 20)
```

No entanto, essa variável precisaria de $3 \times 20 \times 20$ blocos de memória, cada um com um comprimento de 6 *bytes* (para números de vírgula flutuante), ou seja $1200 \times 6 = 7200$ *bytes*. Como um Spectrum de 16K tem cerca de 9000 *bytes* livres depois de ter tratado dos ficheiros de apresentação e de atributos, quando tivessem sido introduzidos o programa e os outros dados, e tendo de contar ainda com o espaço necessário para o aparelho fazer os seus cálculos... Bom, provavelmente acabava-se-nos a memória. Temos de voltar a pensar no problema.

Na realidade, é estúpido guardar as coordenadas com vírgula flutuante: só é possível PLOT x, y quando x for um número inteiro entre 0 e 255 e y for um número inteiro entre 0 e 175. Por uma estranha coincidência, $0 - 255$ é precisamente o âmbito dos códigos de caracteres... Assim, usando um único carácter e servindo-nos do seu código, podemos obter a coordenada x; para a coordenada y e a magnitude procedemos da mesma forma.

Assim sendo, cada estrela vai ocupar três caracteres. Vinte estrelas ocupam $3 \times 20 = 60$, e podem ser, por conveniência, dispostas sequencialmente como uma cadeia. Para 20 constelações podemos agora utilizar:

```
DIM p$ (20, 60)
```

Desta forma, ocuparemos apenas $20 \times 60 = 1200$ *bytes*, o que constitui uma diminuição de tamanho apreciável.

Juntando tudo, chegamos à seguinte rotina de *input* (o programa vem mais adiante):

```
9000 REM input routine—can be deleted after use
9010 DIM n$ (20, 13): DIM e$ (20, 13): DIM p$ (20, 60)
```

```
9020 FOR i = 1 TO 20
9030 INPUT "Latin name?"; a$
9040 LET n$ (i) = a$: LET n$ (i, 13) = CHR$ LEN a$
9050 PRINT TAB 0; n$ (i) (TO CODE n$ (i, 13) );
9060 INPUT "Portuguese name?"; a$
9070 LET e$ (i) = a$: LET e$ (i, 13) = CHR$ LEN a$
9080 PRINT TAB 14; e$ (i) (TO CODE e$ (i, 13) ) '
9090 LET c = 1
9100 INPUT "horiz, vert, mag"; x, y, m
9110 IF x = 0 THEN GO TO 9150
9120 LET p$ (i) (3 * c - 2 TO 3 * c) = CHR$ x + CHR$ y + CHR$ m
9130 LET c = c + 1: IF c > 20 THEN GO TO 9150
9140 GO TO 9100
9150 NEXT i
```

OS DADOS

O passo seguinte consiste em introduzir os dados no aparelho. Mais à frente vou explicar como os arranjar para outras constelações. O leitor pode decidir saltar, de momento, esta secção, para ver o resto do programa.

RUN a rotina de *input* e introduza, quando forem pedidos, os dados que se seguem. Cada linha representa a resposta a uma indicação de pedido de *input*: deve notar-se que se deve ENTER cada item da linha separadamente. O formato do visor não vai ser exactamente o mesmo. Para parar, ENTER 0, 0, 0.

```
Cancer
Caranguejo
      95        54        4
     111       141        4
     113        87        4
     115        96        4
     149        42        4
     156        89        4

Cygnus
Cisne
      64       120        4
      72        36        3
      74        85        4
      76        76        4
      91       107        4
```

104	54	2
105	66	4
114	111	1
122	36	4
124	86	2
139	123	4
142	118	4
161	50	4
168	100	3
179	146	4
188	156	4
192	26	3

Gemini
Gémeos

66	113	1
68	94	3
76	106	4
78	132	1
88	111	4
97	82	3
100	54	3
105	123	4
118	76	4
124	144	3
140	100	3
143	36	3
152	54	2
162	76	4
168	87	3
178	89	3
193	94	4

Leo
Leão

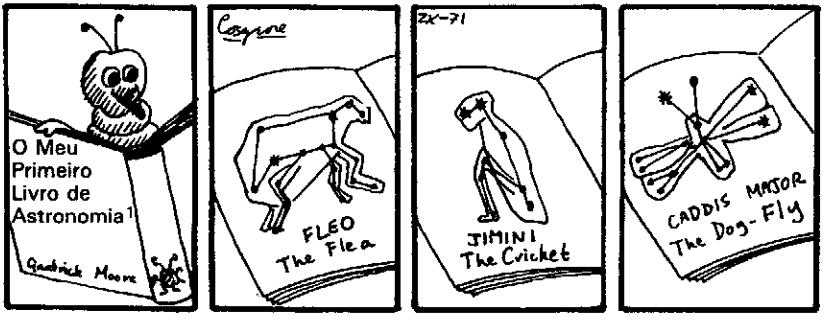
28	90	2
56	49	4
58	74	4
72	97	3
76	122	2
78	53	4
92	118	4
100	139	4
120	52	4
144	110	2
150	128	3
153	57	1
158	91	3
181	140	4
188	126	3
188	52	4
204	121	4
216	152	4

Orion
Orion

78	133	4
86	121	1
104	32	2
108	72	2
110	106	4
113	136	3
115	76	2
118	51	3
120	81	2
126	117	2
128	94	4
130	71	3
134	44	4
142	50	3
144	100	4
148	43	1
154	100	3
154	60	3
171	116	4
172	124	3

Ursa Major
Ursa Maior

27	131	2
56	140	2
73	134	2
94	128	3
104	116	2
106	87	4
124	21	3
126	15	4
133	70	3



[1] Sobre os três últimos quadros: trocadilho com as constelações do Leão (Leo), dos Gémeos (Gemini) e do Grande Cão (Cannis Major): *flea, cricket e dog-fly* significam, respectivamente, «pulga», «grilo» e «libelinha». *(N. da T.)*

136	142	2
137	118	2
173	60	3
176	66	3
176	140	4
180	156	4
183	117	4
200	112	3
211	162	3
225	103	3
226	109	3

Agora, se tiver conseguido chegar até aqui sem desfalecer, o melhor que tem a fazer é guardar (SAVE) estes dados antes que, acidentalmente, acabe por os perder todos. A maneira mais fácil de o fazer consiste em SAVE tudo, incluindo a rotina; no caso de querer introduzir posteriormente mais alguma coisa, isso traz-lhe vantagens. Se não quiser proceder desta forma, pode utilizar *armazenamento de variáveis indexadas:* isso será feiro por três vezes, usando:

```
SAVE "Latin" DATA n$ ( )
SAVE "Portuguese" DATA e$ ( )
SAVE "Stars" DATA p$ ( )
```

Para voltar a introduzir os dados, caso os tenha guardado desta última maneira, use instruções semelhantes, mas nas quais SAVE é substituído por LOAD.

VERIFICAÇÕES E CORRECÇÕES

Se, ao passar os dados, tiver feito um erro, nem tudo está perdido. Suponhamos que se enganou na quinta estrela do Leão. O que tem a fazer é dactilografar no aparelho a instrução directa:

```
LET i = 4: LET c = 5: GO TO 9100
```

Introduza então a versão corrigida de x, y e m; depois, STOP. Para verificar o seu *input*, utilize a seguinte rotina:

```
9500 FOR i = 1 TO 20
9510 PRINT n$ (i) (TO CODE n$ (i, 13) )
9520 PRINT e$ (i) (TO CODE e$ (i, 13) )
9530 FOR t = 1 TO 60 STEP 3
9540 IF p$ (i, t) = "0" OR p$ (i, t) = "□" THEN GO TO 9570
```

```
9550  PRINT CODE p$ (i, t); "□"; :
      PRINT CODE p$ (i, t + 1); "□"; :
      PRINT CODE p$ (i, t + 2)
9560  NEXT t
9570  NEXT i
```

Apenas valerá a pena utilizar esta rotina no caso de estar preocupado com a exactidão do seu trabalho; se tiver verificado com muito cuidado a apresentação no visor durante o curso do *input*, tal não deve ser necessário.

Se tiver uma impressora, substitua todos os PRINT por LPRINT. Obterá assim uma cópia permanente para referência.

A OBTENÇÃO DOS DADOS

Se o leitor quiser acrescentar outras constelações, vai ter de descobrir quais os números que deve *input*.

A forma mais fácil de o conseguir consiste em desenhar a constelação desejada num pedaço de papel milimétrico no qual foi definida uma quadrícula com 256 × 176. (Na realidade, eu usei uma quadrícula com 64 × 44 e multipliquei por quatro). Utilize uma obra de referência

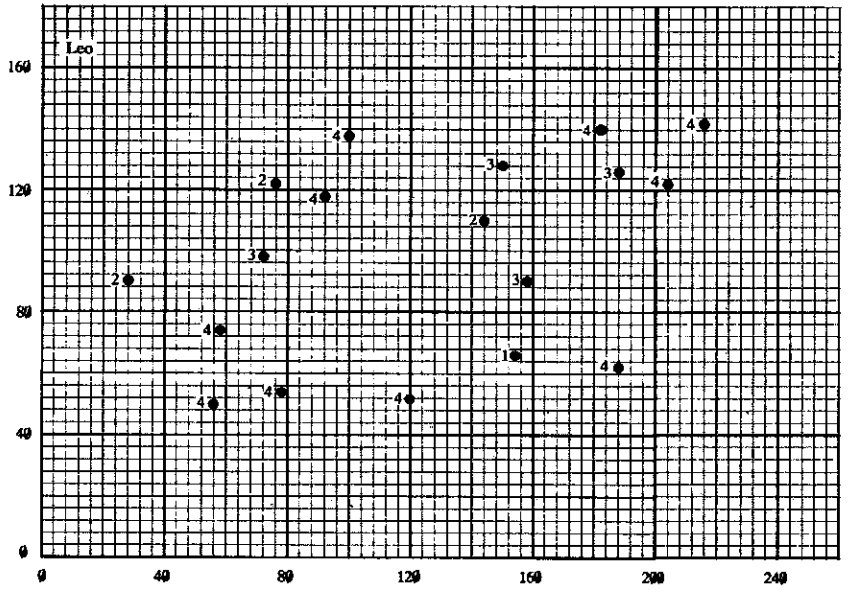
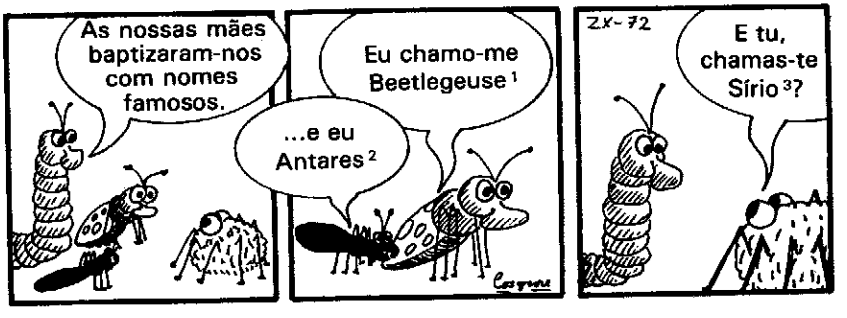


Fig. 18.1 — *Dados de* input *para o Leão*

sobre astronomia: o livro *Norton's Star Atlas,* publicado por Gall and Inglis, é bastante bom. Copie as estrelas para papel vegetal e transfira o resultado para a sua quadrícula. Laboriosamente, tire o valor das coordenadas e consulte o atlas celeste para saber as magnitudes. A figura 18.1 mostra este processo efectuado para a constelação Leão.

Fig. 18.2 — *Apresentação no visor da Ursa Maior. Está a ver a Caçarola, acima, à esquerda?*



1 Trocadilho entre «beetle» (escaravelho) e a estrela Betelguese. *(N. da T.)*

2 Trocadilho entre «ant» (formiga) e a estrela Antares. *(N. da T.)*

3 Trocadilho baseado na pronúncia do nome da estrela Sírio. De facto, «Are you Sirius?» soa da mesma maneira que «Are you serious?» que no presente contexto se pode traduzir por «Estás a gozar comigo?» *(N. da T.)*

Como alternativa a este processo, pode desenhar a constelação numa folha transparente e usar uma versão do método Sketchpad, utilizado no capítulo 1 para desenhar um mapa. O leitor precisará de escrever uma pequena rotina para transferir as coordenadas dos *pixels* para p$ e para introduzir a magnitude de forma adequada. É um projecto para o leitor, se for entusiasta. A variável do sistema COORDS (cap. 6) é capaz de se mostrar útil.

O PROGRAMA PRINCIPAL

Se o leitor quiser, pode apagar as rotinas de introdução e verificação dos dados. Mas *não dactilografe* RUN, senão perde todos os dados e tem de voltar a introduzi-los a partir da fita. Em vez disso, use GO TO 1.

É evidente que precisamos de uma rotina para marcar as estrelas. A rotina que se segue marca uma estrela, cujo tamanho é determinado pela magnitude m, no *pixel* cuja posição é x, y. (As estrelas de magnitude 1 são as mais brilhantes; vêm depois destas as magnitudes 2, 3, e assim sucessivamente.) O número da constelação de que a rotina trata é i.

```
8000  REM star plot
8010  PAPER 0: INK 7: BORDER 0: CLS
8020  FOR t = 1 TO 60 STEP 3
8030  LET k = CODE p$ (i, t)
8040  IF k = 32 OR k = 48 THEN RETURN
8050  LET x = k: LET y = CODE p$ (i, t + 1):
      LET m = CODE p$ (i, t + 2)
8060  GO SUB 8500
8070  NEXT t
8080  RETURN

8500  REM draw one star
8510  LET s = 10 - 2 * m
8520  PLOT x, y - s: DRAW 0, 2 * s
8530  PLOT x - s, y: DRAW 2 * s, 0
8540  PLOT x - s / 2, y - s / 2: DRAW s, s
8550  PLOT x - s / 2, y + s / 2: DRAW s, -s
8560  RETURN
```

Como ainda se deve lembrar, queríamos três opções: lista automática, busca a partir do nome e teste estocástico. Vamos estabelecer um pequeno *menu:*

```
100 PRINT "Star Charts" ' '
110 PRINT "Options;
          1. Automatic List
          2. Name Search
          3. Astroquiz"
120 INPUT "Option number?"; opt
```

O leitor já está prático neste jogo, por isso deixo ao seu gosto e fantasia pessoais a criação de uma apresentação mais sofisticada.

```
130 GO SUB 1000 * opt
140 PAUSE 0: CLS: GO TO 100
```

Vamos agora escrever as opções:

```
1000 FOR i = 1 TO 6
1010 GO SUB 8000
1020 PRINT AT 0, 0; n$ (i) (TO CODE n$ (i, 13) );
     "□ ("; e$ (i) (TO CODE e$ (i, 13) ); ")"
1030 INPUT "Hit ENTER to continue", d$
1040 NEXT i
1050 RETURN

2000 INPUT "Latin name of constellation?"; q$
2010 FOR i = 1 TO 6
2020 IF n$ (i) (TO CODE n$ (i, 13) ) < > q$ THEN GO TO 2045
2030 GO SUB 8000
2040 PRINT AT 0, 0; q$; "□ ("; e$ (i) (TO CODE e$ (i, 13) ); ")"
2045 NEXT i
2050 RETURN

3000 LET i = INT (1 + 6 * RND)
3010 GO SUB 8000
3020 INPUT "Which constellation is this?"; q$
3030 IF q$ = n$ (i) (TO CODE n$ (i, 13) ) THEN
     PRINT AT 0, 0; FLASH 1; "CORRECT!"
```

```
3040  IF q$ < > n$ (i) (TO CODE n$ (i, 13) ) THEN
      PRINT AT 0, 0; FLASH 1; "Sorry, wrong answer":
      PRINT AT 1, 0, FLASH 0; n$ (i) (TO CODE n$ (i, 13) )
3050  PAUSE 100: RETURN
```

Para experimentar, dactilografe GO TO 1 (não se esqueça de que *não* pode carregar em RUN) e siga as instruções apresentadas no visor.

No caso de ter introduzido mais do que 6 constelações, vai ter de, nas linhas 1000, 2010 e 3000, substituir o 6 pelo novo número de constelações.

Guarde este programa usando uma instrução do tipo:

```
SAVE "Starch" LINE 100
```

O programa vai, assim, passar automaticamente ao ser introduzido, evitando o perigo de apagar as suas variáveis.

SOLUÇÕES PARA OS PROBLEMAS CRIPTANÁLISE

1. Código abcdefghijklmnopqrstuvwxyz
   owgbueplqsznhftkirdjmyvcax

   There are nine and sixty ways of constructing tribal lays, and every single one of them is right.
   (Kipling)

2. Código abcdefghijklmnopqrstuvwxyz
   btgpxeuqfwocvzyasnrlkhidjm

   Age cannot wither her, nor custom stale her infinite variety. Other women cloy the appetites they feed.
   (Shakespeare)

3. Código abcdefghijklmnopqrstuvwxyz
   zvetrsnbjpkcuxdfgayohlqimw

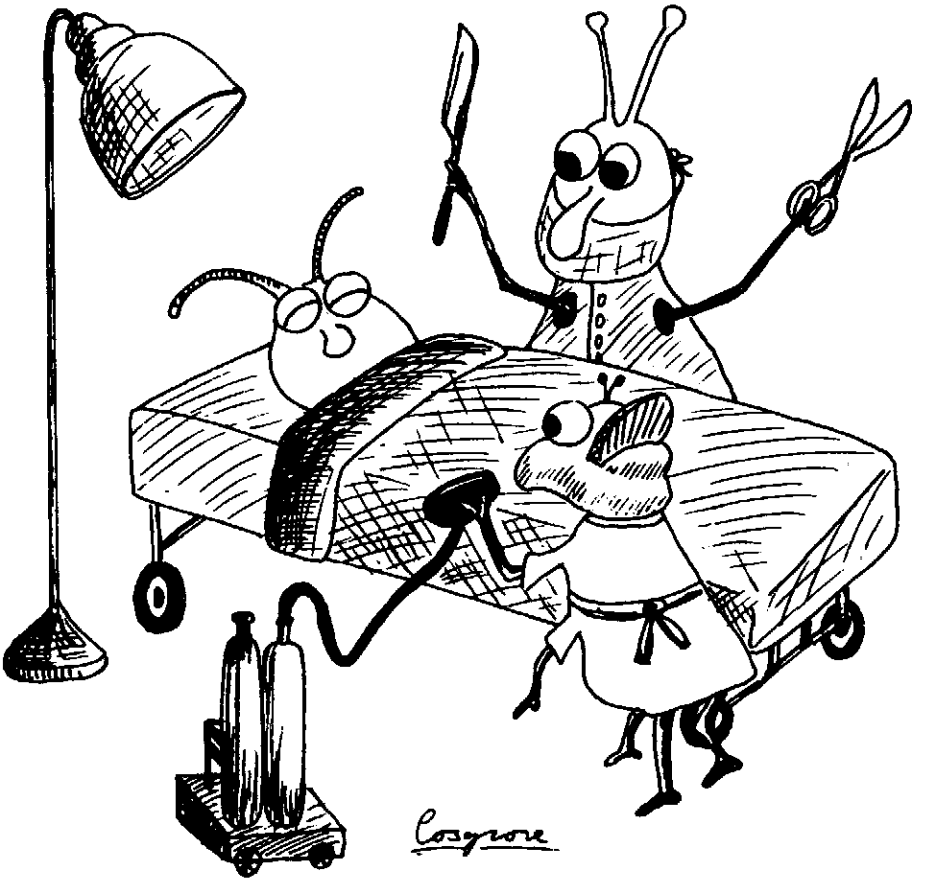
   When a man has married a wife he finds out whether her knees and elbows are only glued together.
   (Blake)

4. Código abcdefghijklmnopqrstuvwxyz
   jkyqzfclotsndxuiegmvwrabhp

   Nature's great masterpiece, the elephant: the only harmless great thing.
   (Donne)

É evidente que o leitor não vai ser capaz de decifrar a totalidade do código, mas apenas as letras que são, *de facto,* usadas.

APÊNDICES



APÊNDICE A

O SISTEMA DE FICHEIRO EM *CASSETTES*. UMA DESCRIÇÃO REFERENCIADA DO *CFS*

Apesar de, neste livro, o *cfs* ter sido usado apenas no *sdm* (O Gestor de Dados do Spectrum — Spectrum Data Manager), não existe, como é óbvio, razão nenhuma para que não seja incluído noutros programas. Para o fazer, basta introduzir todas as rotinas do *cfs*, juntamente com os seus identificadores e os números das linhas onde começam, e SAVE esse conjunto com um programa chamado «cfs». Após ter introduzido o programa que utiliza *cfs*, dactilografe no aparelho:

```
MERGE cfs
```

Faça passar a fita do «cfs», como se estivesse a efectuar um LOAD normal. O efeito resultante é a combinação do programa em memória com o da fita. Eis uma maneira eficaz de colocar *cfs* (ou outro conjunto de rotinas utilitárias) em qualquer programa.

É evidente que o programa principal não pode utilizar nenhum dos números de linhas usadas por *cfs*, e é necessário ter cuidado para que não haja sobreposição dos nomes das variáveis (caso contrário, é extremamente fácil dar a q o valor de 1, chamar uma sub-rotina, e descobrir então que tomou, misteriosamente, o valor 14). Este apêndice fornece as informações necessárias para que este tipo de conflitos não se verifique.

É apresentada uma listagem completa de *cfs*, juntamente com descrições das acções de cada rotina, e são indicados os nomes das variáveis por elas utilizadas. Deve notar-se que a listagem *não é idêntica* à fornecida no texto. Foram introduzidos os melhoramentos discutidos noutro local, além de ter sido feita alguma remuneração.

NÚMEROS DE LINHAS

O *cfs* utiliza a linha 1 e todos os números de linhas superiores a *9000*.

NOMES DE VARIÁVEIS

Os nomes de variáveis usados por *cfs* podem ser divididos em quatro tipos:

1. *Global*

Uma variável global é aquela que pode ser usada por algumas das rotinas do *cfs*, e, consequentemente, nunca deve ser redefinida pelo programa do utilizador.

Por exemplo, *initcfs* estabelece uma variável indexada chamada n$ que contém os nomes dos campos nos ficheiros. Se o utilizador a redefinir seja onde for no seu programa, todos os nomes de campos se transformarão instantaneamente em variáveis literais vazias!

2. *Local*

Estes são nomes que são usados por uma rotina do *cfs*, mas não têm qualquer significado fora dela. O programa do utilizador pode usar estes nomes desde que não necessite que os seus valores sejam preservados após uma chamada da rotina de *cfs* que os utiliza localmente. Por exemplo, *inrec* utiliza p e c como contadores de *loop*. Suponhamos que escrevemos:

```
FOR p = 1 TO 4
GO SUB inrec
NEXT p
```

No caso de o *loop*-p em *inrec* ter sido executado quatro vezes ou mais, o *loop* indicado acima só seria executado uma vez!

Escrevamos agora:

```
For p = 1 TO 4
fazer qualquer outra coisa
NEXT p
GO SUB inrec
```

Neste caso, tudo se passa de forma perfeitamente legítima.

3. *Parâmetros passados à rotina do* cfs

Estes são os nomes das variáveis com as quais o *cfs* tem de trabalhar. Por exemplo, r$ tem de ser passada para *write* para o seu conteúdo ser *output*.

4. *Parâmetros devolvidos por rotinas do* cfs

Estes são os nomes de variáveis geradas por uma rotina do *cfs* para serem subsequentemente usadas pelo programa do utilizador. Por exemplo, r$ é devolvida por *read.*

VARIÁVEIS GLOBAIS

Nome	Utilização
brp	Número de *bypes* por registo.
eof	Bandeira de fim de ficheiro: Ø = não fim de ficheiro, 1 = fim de ficheiro.
f$	Nome do ficheiro de *input.*
g$	Nome do ficheiro de *output.*
ip	Indicador para o próximo registo no *input buffer.*
i$ ( )	Variável indexada com duas dimensões que actua como *input buffer.* Tamanho determinado por w (1) e bpr.
inbc	Número do bloco de *input* actual.
n$ (11, 1Ø)	Variável indexada contendo nomes de até 1Ø campos por registo. Cada nome de campo pode ter até 1Ø *bytes* de comprimento.
op	Indicador para o próximo registo no *output buffer.*
o$ ( )	Variável indexada com duas dimensões que actua como *output buffer.* Tamanho como para i$.
outbc	Número do bloco de *output* actual.
switch	Posta em Ø quando nenhum gravador está activado; posta em 1 quando um gravador é activado.
w (11)	Variável indexada contendo o comprimento, em *bytes,* dos campos nomeados em n$. Contém também o número de registos por bloco.

DESCRIÇÕES DE ROTINAS

close

Acção	Escreve os limitadores de ficheiro no *output buffer.*
Parâmetros passados para *close*	Nenhum.
devolvidos	Nenhum.
Variáveis globais afectadas	Nenhuma (note no entanto que r$ é duplamente escrita).
Variáveis locais	Nenhuma.
Rotinas do *cfs* chamadas	*write.*

Listagem

```
9740 LET r$ = "}}"
9750 GO SUB write
9760 LET r$ = "cfsend"
9770 GO SUB write
9780 RETURN
```

getbloc

Acção	*Input* um bloco de ficheiro.
Parâmetros passados para *getbloc*	Nenhum.
devolvidos	Nenhum.
Variáveis globais afectadas	inbc, i$, ip
Variáveis locais	m$
Rotinas do *cfs* chamadas	*mesp*

Listagem

```
9800 LET m$ = STR$ inbc
9810 GO SUB mesp
9820 LOAD f$ + m$ DATA i$ ( )
9825 POKE 23692, 255
9830 GO SUB mesp
9840 LET ip = 1
9850 LET inbc = inbc + 1
9860 RETURN
```

initcfs

Acção	Inicializa o sistema e estabelece nomes de ficheiros e descrições. Os ficheiros fingidos são chamados «null».
Parâmetros passados para *initcfs*	Nenhum.
devolvidos	Nenhum.
Variáveis globais afectadas	n$, w, ip, op, inbc, outbc, bpr, eof, switch, f$, g$, i$, o$
Variáveis locais	p.
Rotinas do *cfs* chamadas	*setup, mesp, mesr, reset*

Listagem

```
9500  DIM n$ (11, 10): DIM w (11): LET bpr = 0: GO SUB reset
9505  INPUT "Enter input file name"; f$
9510  IF f$ = "null" THEN GO SUB setup: GO TO 9525
9514  GO SUB mesp
9515  LOAD f$ + "h1" DATA n$ ( )
9520  LOAD f$ + "h2" DATA w ( )
9521  GO SUB mesp
9525  INPUT "Enter output filename"; g$
9530  FOR p = 2 TO 11
9535  LET bpr = bpr + w (p)
9540  NEXT p
9545  DIM i$ (w (1), bpr): DIM o$ (w (1), bpr)
9547  IF g$ = "null" THEN RETURN
9548  GO SUB mesr
9550  SAVE g$ + "h1" DATA n$ ( )
9555  SAVE g$ + "h2" DATA w ( )
9557  GO SUB mesr
9560  RETURN .
```

inrec

Acção	Apresenta indicações para a introdução de um registo a partir do teclado, um campo de cada vez, e coloca o resultado em r$.
Parâmetros passados para *inrec*	Nenhum.
devolvidos	r$
Variáveis globais afectadas	Nenhuma.
Variáveis locais	a$, begin, p. c, end
Rotinas do *cfs* chamadas	Nenhuma.

Listagem

```
9000  DIM a$ (bpr)
9010  CLS: LET begin = 1
9020  FOR p = 2 TO 11
9025  IF n$ (p, 1) = "□" THEN GO TO 9120
9030  PRINT AT p, 0; n$ (p, 1); ":"; AT p, 4; n$ (p) (2 TO)
```

```
9040  FOR c = 1 TO w (p)
9050  PRINT AT p, 14 + c; "_"
9060  NEXT c
9070  LET end = begin + w (p) - 1
9080  INPUT (n$ (p) (2 TO) ); a$ (begin TO end)
9090  PRINT AT p, 15; a$ (begin TO end)
9100  LET begin = end + 1
9110  NEXT p
9120  LET r$ = a$
9130  RETURN
```

mesp

Acção	Apresenta uma indicação para ligar ou desligar o gravador PLAY.
Parâmetros passados para *mesp*	Nenhum.
devolvidos	Nenhum.
Variáveis globais afectadas	switch.
Variáveis locais	Nenhuma.
Rotinas do *cfs* chamadas	Nenhuma.

Listagem

```
9140  PRINT INVERSE 1; "Start" AND NOT switch; "Stop" AND
      switch; "play recorder"
9150  BEEP .2, 15: PAUSE 6: BEEP .3, 20: PAUSE 80
9160  LET switch = NOT switch
9170  RETURN
```

mesr

Acção	Apresenta uma indicação para ligar ou desligar o gravador RECORD.
Parâmetros passados para *mesr*	Nenhum.
devolvidos	Nenhum.
Variáveis globais afectadas	switch.
Variáveis locais	Nenhuma.
Rotinas do *cfs* chamadas	Nenhuma.

Listagem

```
9300  PRINT INVERSE 1; "Start" AND NOT switch; "Stop" AND
      switch; "recording"
9310  BEEP .2, 20: PAUSE 6: BEEP .3, 15: PAUSE 80
9320  LET switch = NOT switch
9330  RETURN
```

outrec

Acção	Apresenta no visor, por campos, o registo que se encontra em r$.
Parâmetros passados para *outrec*	r$
devolvidos	Nenhum.
Variáveis globais afectadas	Nenhuma.
Variáveis locais	p, begin, end
Rotinas do *cfs* chamadas	Nenhuma.

Listagem

```
9200  LET begin = 1
9210  FOR p = 2 TO 11
9215  IF n$ (p, 1) = "□" THEN PRINT: RETURN
9220  PRINT n$ (p, 1); ":"; TAB 4; n$ (p) (2 TO);
9240  LET end = begin + w (p) − 1
9250  PRINT TAB 15; r$ (begin TO end)
9260  LET begin = end + 1
9270  NEXT p
9280  PRINT: RETURN
```

putblock

Acção	*Output* um bloco para fita.
Parâmetros passados para *putblock*	Nenhum.
devolvidos	Nenhum.
Variáveis globais afectadas	op, outbc
Variáveis locais	m$
Rotinas do *cfs* chamadas	Nenhuma.

Listagem

```
9900  LET m$ = STR$ outbc
9910  GO SUB mesr
9920  SAVE g$ + m$ DATA o$ ( )
9930  GO SUB mesr
9940  LET op = 0
9950  LET outbc = outbc + 1
9960  RETURN
```

read

Acção	Lê um registo para r$.
Parâmetros passados para *read*	Nenhum.
devolvidos	r$.
Variáveis globais afectadas	ip, eof.
Variáveis locais	Nenhuma.
Rotinas do *cfs* chamadas	*getblock*.

Listagem

```
9600  IF ip = 0 OR ip > w (1) THEN GO SUB getblock
9610  IF i$ (ip) (TO 6) = "cfsend" THEN PRINT "Attempt to read past
      end of file": STOP
9620  LET r$ = i$ (ip)
9625  IF r$ (TO 2) = "}}" THEN LET eof = 1
9630  LET ip = ip + 1
9640  RETURN
```

Comentário: Repare que *eof* é posta a zero por *reset*. Assim, se um ficheiro de *input* tiver de ser lido de novo sem que *initcfs* seja chamado segunda vez e o teste de fim de ficheiro for da forma IF eof THEN..., torna-se necessário inserir GO SUB reset no início da nova leitura. Deste modo, também voltam a ser atribuídos aos indicadores dos *buffers* e aos contadores os valores correctos.

reset

Acção	Estabelece indicadores de *buffers*, bandeiras de ficheiros, etc.
Parâmetros passados para *reset*	Nenhum.
devolvidos	Nenhum.

Variáveis globais afectadas	ip, op, incb, outbc, eof, switch.
Variáveis locais	Nenhuma.
Rotinas do *cfs* chamadas	Nenhuma.

Listagem

```
9970  LET ip = 0: LET op = 0: LET inbc = 0:
      LET outbc = 0: LET eof = 0: LET switch = 0
9980  RETURN
```

setup

Acção	Permite ao utilizador definir os formatos de registo e de bloco.
Parâmetros passados para *setup*	Nenhum.
devolvidos	Nenhum.
Variáveis globais afectadas	n$, w.
Variáveis locais	nf, p.
Rotinas de *cfs* chamadas	Nenhuma.

Listagem

```
9400  INPUT "Enter no. of fields"; nf
9405  CLS
9410  FOR p = 2 TO nf + 1
9415  PRINT AT 10, 2; "Field □"; p-1
9420  INPUT "Name of field (1st char: c/n):"; n$ (p)
9422  IF n$ (p, 1) < > "c" AND n$ (p, 1) < > "n" THEN GO TO
      9420
9425  INPUT "No. of bytes:"; w (p)
9430  NEXT p
9435  CLS
9440  INPUT "No. of records per block:"; w (1)
9445  RETURN
```

write

Acção	Escreve um registo de r$ no ficheiro.
Parâmetros passados para *write*	r$.
devolvidos	Nenhum.
Variáveis globais afectadas	op, o$.
Variáveis locais	Nenhuma.
Rotinas do *cfs* chamadas	*putblock*.

```
9700  LET op = op + 1
9710  LET o$ (op) = r$
9720  IF op = w (1) OR r$ = "cfsend" THEN GO SUB putblock
9730  RETURN
```

INICIALIZAÇÕES DA LINHA 1

```
1  LET close = 9740: LET getblock = 9800: LET initcfs = 9500:
   LET inrec = 9000: LET mesp = 9140: LET mesr = 9300:
   LET outrec = 9200: LET putblock = 9900: LET read = 9600:
   LET setup = 9400: LET write = 9700: LET reset = 9970
```

APÊNDICE B

O CONTROLO AUTOMÁTICO DE *CASSETTES*

Já foi referido que, em princípio, o controlo automático de motores de *cassettes* não apresenta qualquer dificuldade. Vamos aqui apresentar uma técnica específica que permite a realização deste objectivo.

O porto ZX Spectrum PPI, comercializado por Kempston (Micro) Electronics, é um porto I/Ø adequado para este caso. Consta de três portos de 8 *bits* que podem ser programados para agir como portos de *input* ou de *output* ou ainda como uma combinação dos dois. Pelo que nos diz respeito, apenas vamos precisar de dois dos vinte e quatro *bits* disponíveis! Vamos portanto seleccionar os *bits* inferiores (bit Ø) dos portos B e C para controlarem, respectivamente, as *cassettes* PLAY e RECORD. (Escolhemos estes dois portos porque, com eles, as ligações às fichas são idênticas, enquanto com o porto A são ligeiramente diferentes.)

O porto não pode ser utilizado para dirigir um *relais* directamente, dado que apenas podem ser retiradas correntes muito fracas (níveis TTL). Por isso, o *output* é usado para ligar um transístor, como é ilustrado na figura B1.

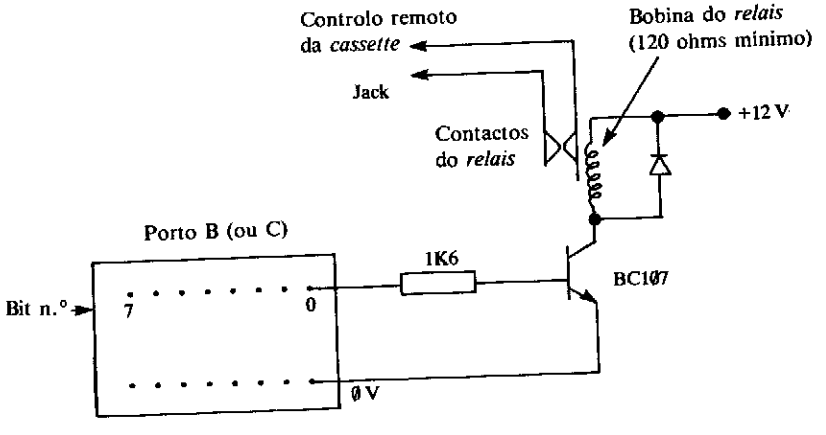


Fig. *B1*

Os portos são inicializados como *output mode* por meio da declaração:

```
OUT 127, 128
```

Esta declaração pode ser introduzida no princípio de *initcfs*.

Seguidamente, é necessário ligar o *bit* Ø do porto B para ser activada a *cassette* PLAY e ligar o *bit* Ø do porto C para ser activada a *cassette* RECORD. Para o fazermos, basta-nos alterar as rotinas *mesp* e *mesr,* substituindo-as por:

```
mesp:  9140  LET switch = NOT switch
       9150  OUT 63, switch
       9160  RETURN

mesr:  9300  LET switch = NOT switch
       9310  OUT 95, switch
       9320  RETURN
```

Os números 63 e 95 são os endereços atribuídos, respectivamente, aos portos B e C.

O diagrama de circuito (fig. B1) é baseado noutro realizado pela Kempston (Micro) Electronics. Poderá entrar em contacto com esta companhia através da morada seguinte: 60, Adamson Court, Hillgrounds Road, Kemston, Bedford MK42 8QZ.

APÊNDICE C

UM GUIA PARA O UTILIZADOR DO *SDM*. O GESTOR DE DADOS DO SPECTRUM

O SDM é um sistema simples de gestão de dados que permite o uso de um ficheiro de *input* e um ficheiro de *output*. Qualquer destes ficheiros pode ser especificado como inexistente através do uso do nome «null», reservado para estes casos. O formato dos registos é de comprimento fixo, e podem ser reconstituídos por um número de campos não superior a dez, sendo os comprimentos e os nomes destes campos definidos pelo utilizador. Cada nome de campo pode ser constituído por um número de caracteres não superior a dez, sendo o primeiro deles obrigatoriamente «c» ou «n», (apenas caracteres minúsculos), para indicar se se trata, respectivamente de um campo literal ou numérico. Esta distinção é importante, porque vai determinar a forma como vão ser feitas a procura e a adição de registos nos ficheiros. Por exemplo, uma tentativa de procura de registos cujo terceiro campo contenha 357 não vai, caso o campo tenha sido declarado do tipo «c», encontrar um registo cujo terceiro campo seja 357.Ø ou +357. Além disso, não é permitido somar campos de tipo «c».

Os nomes dos ficheiros são definidos pelo utilizador. Devem obedecer às regras normais para os nomes dos ficheiros em BASIC, exceptuando que não podem exceder um comprimento de 8 caracteres (2 a menos do que a restrição do BASIC). Se o ficheiro for muito longo, é capaz de ser mais seguro impor um limite de, digamos, 6 caracteres por nome. A razão para esta restrição reside no facto de o nome de ficheiro final passado para BASIC ser formado pelo nome de ficheiro dado pelo utilizador mais um número de bloco. Na realidade, os blocos de um ficheiro chamado «filename» vão ser:

filenameh1
filenameh2
filenameØ
filename1
filename2
etc.

Os dois primeiros blocos formam um cabeçalho que descreve ao sistema a estrutura dos registos e campos do ficheiro. A estes seguem-

-se os blocos de dados, que são etiquetados sequencialmente a partir de zero. Consequentemente, um ficheiro que tenha um número de blocos superior a 1Ø0 e um nome com 8 caracteres transgride as regras do BASIC, pelo que o programa pára com erro F. Contudo, na prática, é improvável que apareça um ficheiro deste tamanho.

As operações que o SDM permite são:

1. A criação de ficheiros.
2. A modificação de ficheiros existentes através da adição ou da remoção de registos, ou pela criação de subficheiros.
3. A procura nos ficheiros de registos com atributos especificados.
4. A soma aritmética de todos os campos numéricos especificados existentes num ficheiro.

O acesso a estas operações é veiculado através de um *menu* que é apresentado sempre que uma operação é completada. Deve, no entanto, notar-se que esta apresentação repetida do *menu* serve para facilitar a execução de várias operações sobre *o mesmo* ficheiro. Se for preciso efectuar diversas operações sobre ficheiros diferentes, deve voltar-se a passar o programa.

A ACÇÃO DO SDM

Depois de o SDM ter sido introduzido e de se ter carregado em RUN, aparece no visor a mensagem «Spectrum Data Manager», («Gestor de Dados do Spectrum»), que é seguida por uma indicação para ser introduzido o nome do ficheiro de *input*. Se o ficheiro de *input* existir, deve ser colocado no gravador PLAY, sendo o seu nome dactilografado no teclado do aparelho. O sistema vai pedir que o gravador seja activado e introduz os dois blocos cabeçalhos; seguidamente, pede que o gravador seja desligado.

No caso de não existir ficheiro de *input*, deve ser dactilografada a palavra «null», após o que o sistema irá pedir uma definição de ficheiro.

As indicações apresentadas são (por ordem):

	Indicação	Significado
	1. Enter n.° of fields	Número de campos em 1 registo.
Estes passos são repetidos para cada campo. O número do campo é apresentado como uma indicação adicional.	2. Name of field (1st char:c/n)	Introduzir o nome de um campo, constituído por c ou n seguido por 9 caracteres no máximo; por exemplo, um campo contendo um nome poderia chamar-se «cnome», outro contendo um balanço bancário seria «nbal».
	3. No. of bytes.	Este é o número máximo de *bytes* que se prevê que seja ocupado por um valor guardado no campo cujo nome acaba de ser introduzido.
	4. No. of records per block	Para simplificar as operações, o número de registos por bloco deveria ser alto, da ordem dos 2Ø ou mais. No entanto, as restrições de memória e o tamanho de cada registo podem impedir, na prática, os grandes tamanhos de blocos, especialmente num aparelho de 16K. O tamanho máximo de blocos possivel calcula-se com bastante facilidade para cada caso específico, mas geralmente é mais fácil escolher, por ensaio e erro, um valor que assegure simplesmente que a memória disponivel não vai ser excedida.

Depois disto, o sistema apresenta uma indicação para a introdução de um nome para o ficheiro de *output*. No caso de não ir ser gerado nenhum ficheiro, deve ser introduzido o nome de «null»; caso contrário, deve ser introduzido um nome de ficheiro obedecendo às regras expostas acima. O sistema vai então pedir que seja ligado o gravador RECORD e guarda os blocos cabeçalhos, após o que indica que o gravador RECORD deve ser desligado.

Seguidamente, vai ser apresentado o *menu*, de acordo com a lista abaixo:

Options are:		As opções são:	
1) create	4) search	1) criar	4) procurar
2) add	5) sum	2) acrescentar	5) somar
3) delete	6) exit	3) remover	6) sair

É dada ao utilizador a indicação de que deve introduzir uma destas opções (numéricas).

Vamos agora passar à descrição individual de cada uma das opções.

1) **create**

É indicado ao utilizador que deve introduzir cada campo de um registo por sua vez. O número de caracteres de sublinhação que aparece junto ao nome do campo informa-o sobre o tamanho permitido para este. (O tipo do campo está separado do resto do nome por dois pontos, como em *n:bal.*) À medida que cada campo é introduzido, o registo vai aparecendo no visor de modo que o utilizador possa verificar se o que acabou de introduzir está correcto. Quando um registo está completo, aparece no visor «Any more (y/n)? («Mais algum (y/n)»?). Se houver mais registos a serem introduzidos, o utilizador dactilografa «y», caso contrário carrega em «n». No primeiro caso, o processo é repetido; no segundo, é dada ao utilizador a indicação de que deve ligar o gravador RECORD e o ficheiro é fechado. (Deve notar-se que a criação de um ficheiro é pontuada por indicações para o gravador ser controlado à medida que os blocos vão ficando completos.)

2) **add**

É pedido ao utilizador que indique qual o número de registos que pretende acrescentar ao ficheiro. Seguidamente, estes são introduzidos da mesma forma que na rotina *create* (exceptuando que o processo é repetido o número de vezes requerido, não aparecendo a pergunta sobre a existência de mais registos no fim de casa introdução). A ordem pela qual os registos são introduzidos é indiferente.

É seguidamente pedido ao leitor que indique qual o nome do campo chave. Deve então ser especificado qual o nome do campo que define a ordem pela qual os registos são colocados no ficheiro. Este nome *não deve* incluir o tipo de campo (por exemplo, deve escrever-se «bal» *e não* «nbal»). Os novos registos são então inseridos automaticamente, sendo periodicamente fornecidas, pelo sistema ao utilizador, indicações para controlo das *cassettes.*

3) **delete**

É pedido ao utilizador que indique qual o número de registos que quer remover, e em seguida é-lhe pedido que especifique o nome do campo chave (excluindo o código de tipo).

Por fim, é-lhe pedido que introduza as chaves correspondentes aos registos a serem removidos.

Suponhamos, por exemplo, que se queria remover os registos de A. BROWN, G. N. DODDS e P. ADAMS. Introduzimos:

```
How many records? 3
(Quantos registos?)

Enter key field name: name
(Introduzir o nome do campo chave)

Input delection (key only): A. BROWN
Input delection (key only): G. N. DODDS
Input delection (key only): P. ADAMS
(Input registo a ser removido — apenas a chave)
```

O sistema vai então responder de forma idêntica à utilizada para a rotina *add*.

4) search

Em primeiro lugar, é pedido ao utilizador que especifique o nome do campo chave (ou seja, aquele sobre o qual vai ser efectuada a busca). Se se tratar de uma chave de tipo c, é-lhe seguidamente pedido que seja indicada uma chave alvo. Por exemplo, se existir um campo chamado *cdedimp* cujo conteúdo seja «s» se se tratar de um *item* dedutível nos impostos, e quisermos uma lista de todas as transacções dedutíveis nos impostos, as respostas aos pedidos vão ser:

```
Enter key field name: dedimp
Target key: s
(Chave alvo:)
```

No caso de o campo chave ser do tipo n, vai ser pedido um espectro. Se existir um único alvo, as extremidades do espectro a serem introduzidas devem ter o mesmo valor. Por exemplo, se existir um campo chamado *nquantia* e quisermos examinar todos os registos cujo campo da quantia contenha exactamente o valor 1ØØ, os pedidos vão ter o seguinte aspecto:

```
Enter key field name: quantia
Key range — low: 1ØØ
           high: 1ØØ
(Espectro da chave — inferior: 1ØØ
                     superior: 1ØØ)
```

No entanto, se a busca disser respeito a todos os valores iguais ou superiores a 100, podíamos introduzir:

```
Key range — low: 100
           high: 1000
```

Isto, evidentemente, se soubéssemos que não existiam valores superiores a 999. Como conhecemos o comprimento do campo, podemos sempre escolher um valor adequado. Por exemplo, a especificação feita acima tem a garantia de estar correcta desde que *nquantia* tenha um comprimento máximo de 3 *bytes*. Paralelamente, se forem requeridas todas as entradas menores do que um determinado valor, o valor «low» tem de ser escolhido de forma a ser inferior a qualquer possível valor do registo (– 100, por exemplo, para um campo de 3 *bytes*).

Finalmente, é pedido ao utilizador que indique qual o dispositivo para onde o *output* deve ser enviado. As modalidades possíveis são a representação no visor (dispositivo 1), a impressora (dispositivo 2) ou o ficheiro de *output* (dispositivo 3).

A última destas opções permite a criação de subficheiro (ou seja, de um novo ficheiro contendo apenas, por exemplo, itens dedutíveis nos impostos).

Deve notar-se que, ainda que não exista uma opção «*list*» que permita enviar para a impressora uma lista completa do ficheiro, essa função pode ser suprida por *search* se for utilizada uma chave numérica tal que tanto o valor «low» como o valor «high» excedem o espectro permitido.

5) sum

É pedido ao utilizador que indique o nome de um campo cujos conteúdos devam ser somados ao longo de todo o ficheiro. Esse campo tem de ser numérico, caso contrário aparecerá uma mensagem de erro, voltando subsequentemente o sistema ao *menu*. Se a escolha do campo tiver sido correcta, a soma final dos valores nele contidos é apresentada no visor, e o sistema espera até que o utilizador carregue numa tecla qualquer para regressar ao *menu*.

COMENTÁRIOS GERAIS

Todas as operações-padrão são realizadas sobre o ficheiro completo, de modo que não é possível o sistema fornecer uma resposta directa a uma pergunta do tipo: «Qual é a soma dos itens dedutíveis nos impostos?» Contudo, este problema pode ser resolvido através da cria-

ção de um ficheiro de itens dedutíveis utilizando *search*, sendo depois somados os valores contidos nos campos da quantia do novo ficheiro.

Paralelamente, as buscas complexas (por exemplo: o número de itens dedutíveis nos impostos cujos valores se encontram entre 5Ø e 1ØØ libras) podem ser efectuadas através do uso sucessivo da opção *search*, sendo gerado de cada vez um novo subficheiro.

APÊNDICE D

O GESTOR DE DADOS DO SPECTRUM. LISTAGEM DO PROGRAMA

É apresentada abaixo a listagem final revista do SDM. Por uma questão de simplicidade, foram aqui incluídas as rotinas do *cfs*. Se o leitor já tiver guardado (SAVE) estas rotinas, *não vai precisar* de voltar a copiar as linhas 1 e 9000-9980; o que tem a fazer é introduzi-las (LOAD) antecipadamente, ou fundi-las (MERGE) no fim.

```
0  © Ian Stewart and Robin Jones 1982
1  LET close = 9740: LET getblock = 9800: LET initcfs = 9500:
   LET inrec = 9000: LET mesp = 9140: LET mesr = 9300:
   LET outrec = 9200: LET putblock = 9900: LET read = 9600:
   LET reset = 9970: LET setup = 9400: LET write = 9700
2  LET loutrec = 8800: LET extractkey = 7800: LET loosends = 7600:
   LET compc = 7400: LET compn = 7200: LET sort = 7000:
   LET ckeytest = 6800: LET nkeytest = 6600
10 CLS
20 PRINT AT 0, 5; "Spectrum Data Manager"
25 GO SUB initcfs
26 CLS
30 PRINT AT 2, 0; "Options are:"
40 PRINT AT 3, 11; "1) create"
41 PRINT AT 4, 11; "2) add"
42 PRINT AT 5, 11; "3) delete"
43 PRINT AT 6, 11; "4) search"
44 PRINT AT 7, 11; "5) sum"
45 PRINT AT 8, 11; "6)exit"

100 INPUT "Enter option:"; opt
110 GO SUB 200 * opt
```

```
120  GO SUB reset
130  GO TO 26

200  GO SUB inrec
210  GO SUB write
220  INPUT "any more? (y/n)"; q$
230  IF q$ = "y" THEN GO TO 200
240  GO SUB close
250  RETURN

400  INPUT "How many records?"; nr
410  DIM b$ (nr, bpr)
420  FOR q = 1 TO nr
430  GO SUB inrec
440  LET b$ (q) = r$
450  NEXT q
460  GO SUB extractkey
465  GO SUB sort
470  LET ap = 1
480  GO SUB read
490  IF ap > nr OR eof THEN GO SUB loosends: RETURN
500  LET s$ = r$ (begin TO end): LET t$ = b$ (p) (begin TO end)
510  IF ctype THEN GO SUB compc
520  IF NOT ctype THEN GO SUB compn
530  IF comp < 0 THEN GO SUB write: GO TO 480
540  IF comp > = 0 THEN LET t$ = r$: LET r$ = b$ (p): GO SUB
     write: LET r$ = t$: LET ap = ap + 1: GO TO 490

600  INPUT "How many records?"; nr
610  GO SUB extractkey
620  DIM d$ (nr, end - begin + 1)
630  FOR p = 1 TO nr
640  INPUT "Enter deletion (key only):"; d$ (p)
650  NEXT p
660  GO SUB read
670  IF eof THEN GO SUB close: RETURN
```

```
680  FOR p = 1 TO nr
690  IF r$ (begin TO end) = d$ (p) THEN GO TO 660
700  NEXT p
710  GO SUB write
720  GO TO 660

800  GO SUB extractkey
810  IF ctype THEN DIM k$ (end - begin + 1): INPUT "Target key:
     "; k$
820  IF NOT ctype THEN INPUT "key range—low:"; low, "high:";
     high
830  INPUT "Display (1), print (2) or file (3):"; opt
840  GO SUB read
850  IF eof AND opt = 3 THEN GO SUB close: RETURN
860  IF eof THEN INPUT "Enter c to continue"; k$: RETURN
870  IF ctype THEN GO SUB ckeytest
880  IF NOT ctype THEN GO SUB nkeytest
890  IF NOT match THEN GO TO 840
895  LET bs = begin: LET es = end
900  IF opt = 1 THEN GO SUB outrec
910  IF opt = 2 THEN GO SUB loutrec
920  IF opt = 3 THEN GO SUB write
925  LET begin = bs: LET end = es
930  GO TO 840

1000 LET sum = 0
1010 GO SUB extractkey
1020 IF ctype THEN PRINT "Key not numeric": PAUSE 120:
     RETURN
1030 GO SUB read
1040 IF eof THEN PRINT "Sum of □"; k$; "= □"; sum:
     INPUT "Enter c to continue"; k$: RETURN
1050 LET sum = sum + VAL r$ (begin TO end)
1060 GO TO 1030

1200 STOP
```

```
6600  LET match = 1
6610  IF VAL r$ (begin TO end) < low OR VAL r$ (begin TO end) >
      high  THEN LET match = 0
6620  RETURN

6800  LET match = 0
6810  IF k$ = r$ (begin TO end) THEN LET match = 1
6820  RETURN

7000  LET inc = 1
7005  LET flag = 0
7010  FOR p = 1 TO nr - inc
7020  LET s$ = b$ (p) (begin TO end): LET t$ = b$ (p + 1) (begin TO
      end)
7030  IF ctype THEN GO SUB compc
7040  IF NOT ctype THEN GO SUB compn
7050  IF comp > 0 THEN LET t$ = b$ (p): LET b$ (p) = b$ (p + 1):
      LET b$ (p + 1) = t$: LET flag = 1
7060  NEXT p
7070  IF flag > 0 THEN LET inc = inc + 1: GO TO 7005
7080  RETURN

7200  IF VAL s$ < VAL t$ THEN LET comp = - 1
7210  IF VAL s$ = VAL t$ THEN LET comp = 0
7220  IF VAL s$ > VAL t$ THEN LET comp = 1
7230  RETURN

7400  IF s$ < t$ THEN LET comp = -1
7410  IF s$ = t$ THEN LET comp = 0
7420  IF s$ > t$ THEN LET comp = 1
7430  RETURN

7600  IF ap > nr THEN GO TO 7670
7610  FOR p = ap TO nr
7620  LET r$ = b$ (p)
7630  GO SUB write
7640  NEXT p
```

```
7650  GO SUB close
7660  RETURN
7670  GO SUB write
7680  GO SUB read
7690  IF eof THEN GO SUB close: RETURN
7700  GO TO 7670

7800  DIM k$ (9): INPUT "Enter key field name"; k$
7810  LET begin = 1
7820  FOR p = 2 TO 11
7830  IF n$ (p) (2 TO  ) = k$ THEN GO TO 7860
7840  LET begin = begin + w (p)
7850  NEXT p
7860  LET end = begin + w (p) - 1
7870  IF n$ (p, 1) = "c" THEN LET ctype = 1
7880  IF n$ (p, 1) = "n" THEN LET ctype = 0
7890  RETURN

8800  LET begin = 1
8810  FOR p = 2 TO 11
8815  IF n$ (p, 1) = "□" THEN LPRINT: RETURN
8820  LPRINT n$ (p, 1); ":"; TAB 4; n$ (p) (2 TO  );
8840  LET end = begin + w (p) - 1
8850  LPRINT TAB 15; r$ (begin TO end)
8860  LET begin = end + 1
8870  NEXT p
8880  LPRINT: RETURN

9000  DIM a$ (bpr)
9010  CLS: LET begin = 1
9020  FOR p = 2 TO 11
9025  IF n$ (p, 1) = "□" THEN GO TO 9120
9030  PRINT AT p, 0; n$ (p, 1); ":"; AT p, 4; n$ (p) (2 TO  )
9040  FOR c = 1 TO w (p)
9050  PRINT AT p, 14 + c; "_"
9060  NEXT c
```

```
9070  LET end = begin + w (p) - 1
9080  INPUT (n$ (p) (2 TO  )); a$ (begin TO end)
9090  PRINT AT p, 15; a$ (begin TO end)
9100  LET begin = end + 1
9110  NEXT p
9120  LET r$ = a$
9130  RETURN
9140  PRINT INVERSE 1; "Start" AND NOT switch; "Stop" AND
      switch; "PLAY recorder"
9150  BEEP .2, 15: PAUSE 6: BEEP .3, 20: PAUSE 80
9160  LET switch = NOT switch
9170  RETURN

9200  LET begin = 1
9210  FOR p = 2 TO 11
9215  IF n$ (p, 1) = "□" THEN PRINT: RETURN
9220  PRINT n$ (p, 1); ":"; TAB 4; n$ (p) (2 TO  );
9240  LET end = begin + w (p) - 1
9250  PRINT TAB 15, r$ (begin TO end)
9260  LET begin = end + 1
9270  NEXT p
9280  PRINT: RETURN

9300  PRINT INVERSE 1; "Start" AND NOT switch;
      "Stop" AND switch; "recording"
9310  BEEP .2, 20: PAUSE 6: BEEP .3, 15: PAUSE 80
9320  LET switch = NOT switch
9330  RETURN

9400  INPUT "Enter no. of fields:"; nf
9405  CLS
9410  FOR p = 2 TO nf + 1
9415  PRINT AT 10, 2; "Field □"; p - 1
9420  INPUT "Name of field (1st char: c/n):"; n$ (p)
9422  IF n$ (p, 1) < > "c" AND n$ (p, 1) < > "n" THEN GO TO 9420
```

```
9425  INPUT "No of bytes:"; w (p)
9430  NEXT p
9435  CLS
9440  INPUT "No of records per block:"; w (1)
9445  RETURN

9500  DIM n$ (11, 10): DIM w (11): LET bpr = 0: GO SUB reset
9505  INPUT "Enter input filename:"; f$
9510  IF f$ = "null" THEN GO SUB setup: GO TO 9525
9514  GO SUB mesp
9515  LOAD f$ + "h1" DATA n$ ( )
9520  LOAD f$ + "h2" DATA w ( )
9521  GO SUB mesp
9525  INPUT "Enter output filename:"; g$
9530  FOR p = 2 TO 11
9535  LET bpr = bpr + w (p)
9540  NEXT p
9545  DIM i$ (w (1), bpr): DIM o$ (w (1), bpr)
9547  IF g$ = "null" THEN RETURN
9548  GO SUB mesr
9550  SAVE g$ + "h1" DATA n$ ( )
9555  SAVE g$ + "h2" DATA w ( )
9557  GO SUB mesr
9560  RETURN

9600  IF ip = 0 OR ip > w (1) THEN GO SUB getblock
9610  IF i$ (ip) (   TO 6) = "cfsend" THEN PRINT "Attempt to read
      past  end of file": STOP
9620  LET r$ = i$ (ip)
9625  IF r$ (   TO 2) = "}}" THEN LET eof = 1
9630  LET ip = ip + 1
9640  RETURN

9700  LET op = op + 1
9710  LET o$ (op) = r$
```

```
9720  IF op = w (1) OR r$ = "cfsend" THEN GO SUB putblock
9730  RETURN
9740  LET r$ = "}}":
9750  GO SUB write
9760  LET r$ = "cfsend"
9770  GO SUB write
9780  RETURN
9800  LET m$ = STR$ inbc
9810  GO SUB mesp
9820  LOAD f$ + m$ DATA i$ ( )
9825  POKE 23692, 255
9830  GO SUB mesp
9840  LET ip = 1
9850  LET inbc = inbc + 1
9860  RETURN

9900  LET m$ = STR$ outbc
9910  GO SUB mesr
9920  SAVE g$ + m$ DATA o$ ( )
9930  GO SUB mesr
9940  LET op = 0
9950  LET outbc = outbc + 1
9960  RETURN
9970  LET ip = 0: LET op = 0: LET inbc = 0: LET outbc = 0:
      LET eof = 0: LET switch = 0
9980  RETURN
```

COMO FAZER O SEU PRÓPRIO INTERRUPTOR DE LOAD/SAVE

Quantas vezes é que já aconteceu ao leitor esquecer-se de retirar um borne «ear» ao guardar um programa? Quando este esquecimento ocorreu comigo pela décima quinta vez, cheguei à conclusão de que não era tarde nem era cedo para arranjar uma maneira de o impedir. A solução que aqui apresento é simples, mas prática.

Para a realizar, o leitor vai precisar de:

a) 4 bornes de conexão de 3,5 mm;
b) 1 mini-interruptor deslizante de 2 pólos e 2 passos;
c) 2 m de cabo de microfone isolado (unipolar);
d) Uma caixa de *cassette* velha.

Os elementos *a*), *b*) e *c*) estão disponíveis em qualquer loja de artigos para electrónica por uma quantia entre 25 e 50 pénis cada um; quando a *d*), qualquer caixa pequena, de plástico, serve para o efeito desejado.

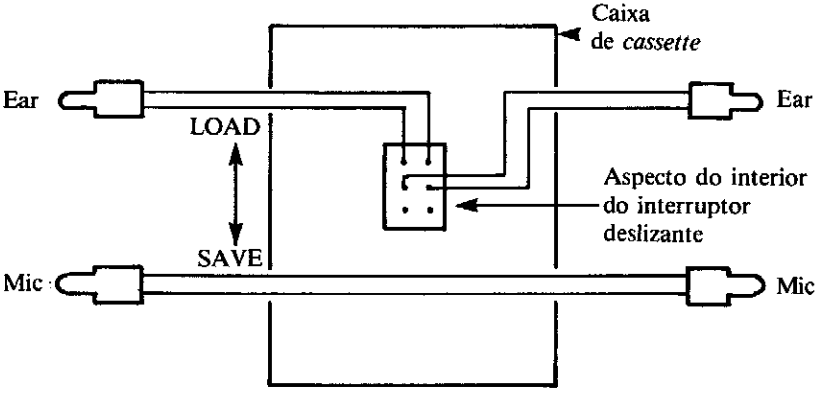


Fig. E.1 — *Diagrama de circuito para um interruptor LOAD/SAVE*

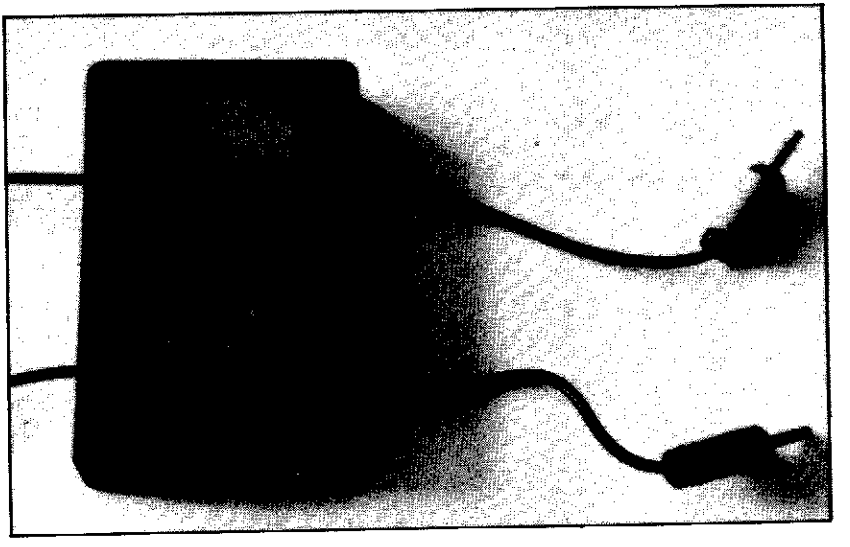


Fig. E.2 — *Aspecto exterior do interruptor*

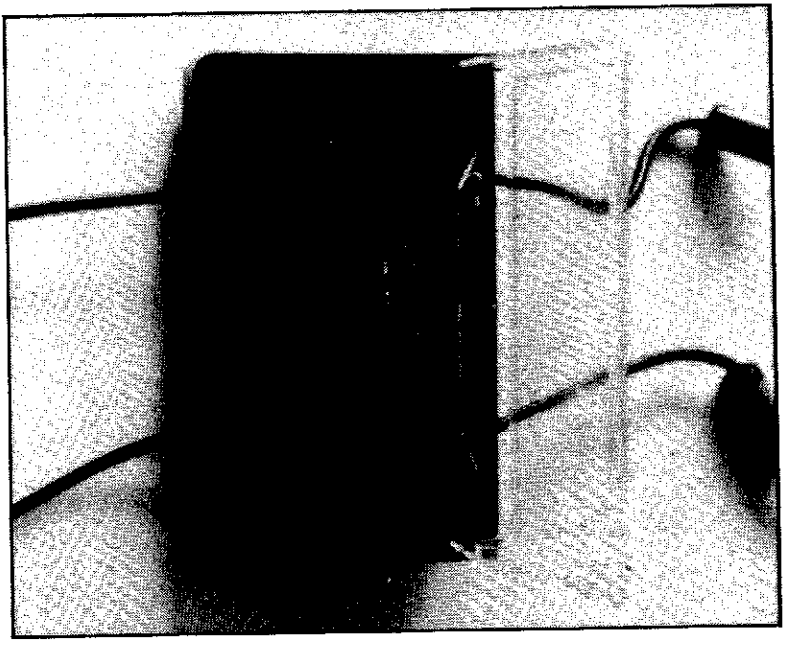


Fig. E.3 — *Aspecto interior do interruptor*

Nos dois lados estreitos da caixa, faça quatro furos que permitam a passagem dos fios. Numa das faces mais largas, abra um buraco que acomode o interruptor deslizante. (A maneira mais simples de ter a certeza de que o tamanho do buraco é o adequado é fazer um molde com um pedaço de fita adesiva em PVC.) Seguidamente, efectue as ligações ilustradas pela figura E1

Para terminar, marque os bornes com fita adesiva colorida em PVC, de modo a não os confundir uns com os outros.

Torna-se assim possível interromper o contacto no fio de «ear». Assim, para SAVE, o utilizador deve colocar o interruptor na posição SAVE (como é lógico), fazendo-o deslizar de novo para a posição LOAD quando quiser LOAD ou VERIFY. Repare que o fio de «mic» não se encontra na realidade ligado a nada dentro da caixa; contudo, para evitar perdê-lo, é conveniente fazê-lo passar pelo interior dela.

UM LIVRO DE EDIÇÕES CETOP

24 JOGOS PARA O sinclair QL

para jogar
tal qual
ou
desenvolver
ao seu próprio
gosto

TIM HARTNELL

GOTO informática

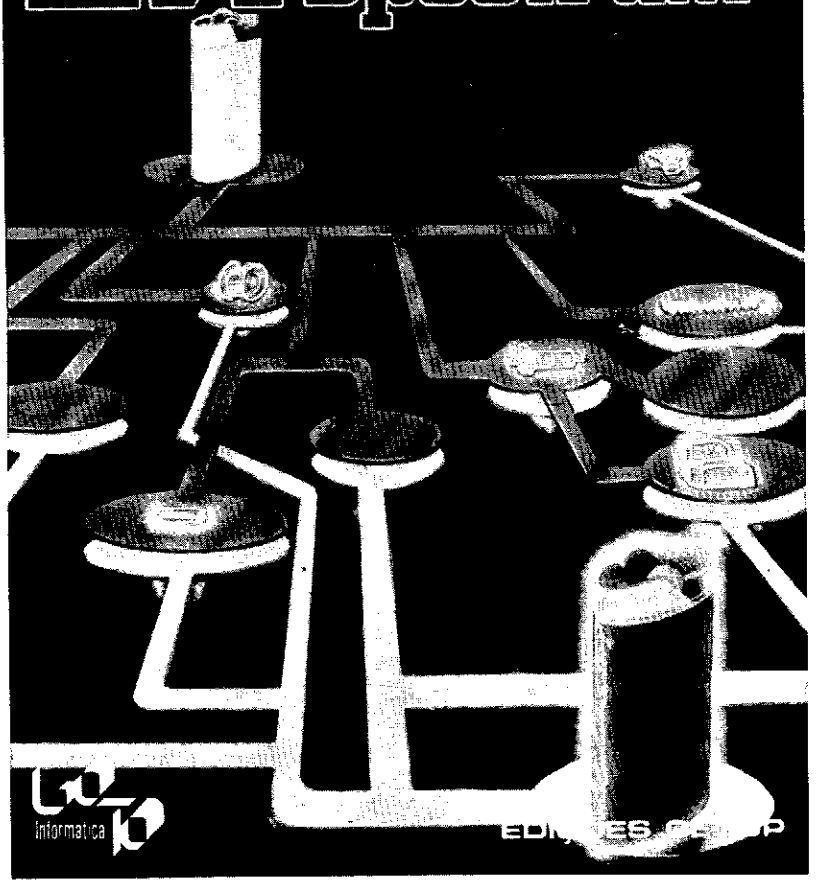
EDIÇÕES CETOP

UM LIVRO DE EDIÇÕES CETOP

ROBIN JONES
MICHAEL FAIRHURST

INTELIGÊNCIA ARTIFICIAL

ZX Spectrum



Informática

EDIÇÕES CETOP

PROGRAMAÇÃO AVANÇADA PARA O ZX SPECTRUM E TK90X DA MICRODIGITAL

Eis uma obra que irá ajudar o leitor a aproveitar ao máximo as capacidades do computador ZX Spectrum.

Aproveite:

- Ficheiros em cassette
- Gestão de dados
- Gráficos à prova de erro
- Flexível renumeração de linhas

Explore:

- Funções definidas pelo utilizador
- Variáveis do sistema
- Atributos e apresentação de ficheiro
- Conjunto de novos caracteres

Empregue:

- «Mapa de estrelas»
- Decifração de códigos
- Psicologia
- Estatísticas

Uma selecção original de programas e aplicações que necessitam só de 16K de memória, e que, desta forma, pode ser utilizada em qualquer modelo Spectrum.

POR QUE NÃO AUMENTAR AS CAPACIDADES DO SEU SPECTRUM?

Informática 6
06 1900686 2

EDIÇÕES CETOP ®