A. P. STEPHENSON

BASIC PARA MICROCOMPUTADORES

TEMPOS LIVRES

CULTURA E TEMPOS LIVRES

1. ABC do Xadrez, *Petar Trifunovitch e Sava Vukcovitch*
4. ABC do Bridge, *Pierre Jais e H. Lahana*
5. Guia Prático de Fotografia, *W. D. Emanuel*
6. ABC do Judo, *E. J. Harrison*
7. Como Fazer Cinema, *Paul Petzold*
8. Bridge Moderno, *Pierre Jais e H. Lahana*
9. Fotografia — Técnicas e Truques I, *Edwin Smith*
10. Estilos de Mobiliário, *A. Aussel*
11. Fotografia — Técnicas e Truques II, *Edwin Smith*
12. A Pesca Submarina, *António Ribera*
13. Teoria dos Finais de Partida, *Yuri Averbach*
14. Aprenda Rádio, *B. Fighiera*
15. Guia do Cão, *Louise Laliberté-Robert e Jean-Pierre Robert*
16. ABC do Aquário, *Anthony Evans*
17. Iniciação à Electricidade e Electrónica, *Fernand Huré*
18. Os Transístores, *Fernand Huré*
19. Karaté I, *Albrecht Pflüger*
20. Iniciação ao Radiocomando dos Modelos Reduzidos, *C. Péricone*
21. Construa o seu Receptor, *B. Fighiera*
22. Montagens Electrónicas, *B. Fighiera*
23. O Berbequim Eléctrico, *Villy Dreier*
24. Cactos, *J. Nilaus Jensen*
25. Iniciação à Alta Fidelidade, *Peter Turner*
26. O Aquário de Água Doce, *Paulo de Oliveira*
27. ABC do Ténis, *Fonseca Vaz*
28. Karaté II, *Albrecht Pflüger*
29. ABC da Criação de Canários, *Curt Af Anehjelm*
30. Ginástica Feminina, *Sonja Helmer Jensen*
31. Cartomancia, *Rhea Koch*
32. Calculadoras Electrónicas de Bolso, *E. Dam Ravn*
33. O Pastor Alemão, *Gilles Legrand*
34. Xadrez — Teoria do Meio Jogo, *I. Bondarevsky*
35. Manual do Super 8 — I, *Myron A. Matzkin*
36. ABC da Criação de Periquitos, *Cyril H. Rogers*
37. O Livro dos Gatos, *Bärbel Gerber e Horst Bielfeld*
38. Manual do Super 8 — II, *Myron A. Matzkin*
39. ABC do Mergulho Desportivo, *Walter Mattes*
40. Circuitos Integrados/Aplicações Práticas, *F. Bergtold*
41. A Apicultura, *H. R. C. Riches*
42. ABC do Cultivo das Plantas, *H. G. Witham Fogg*
43. ABC da Criação de Pombos, *Kai R. Dahl*
44. Construção de Caixas Acústicas de Alta Fidelidade, *R. Brault*
45. Raças de Canários, *Klaus Speicher*
46. Jogos de Cartas, *Graciano Dolma*
47. Cocker Spaniels, *H. S. Lloyd*
48. ABC da Caça, *Fabián Abril*
49. Aprenda Televisão, *Gordon J. King*
50. Iniciação à Pessoa, *Juan Nadal*

51. Basquetebol, *Marius Norregard*
52. Cães de Caça, *Santiago Pons*
53. Aprenda Electrónica, *T. L. Squires e C. M. Deason*
54. A Avicultura, *Jim Worthington*
55. A Produção de Coelhos, *P. Surdeau e R. Henaff*
56. ABC dos Computadores, *T. F. Fry*
57. Natação para Crianças, *John Idorn*
58. O Boxer, *Anni Mortensen*
59. Voleibol, *Ole Hansen e Per-Göran Persson*
60. Iniciação à Vela, *Donald Law*
61. ABC da Filatelia, *Jacqueline Caurat*
62. A Pesca à Beira-Mar, *J.-M. Boëlle e B. Doyen*
63. Enxerto de Árvores de Fruto, *Alejo Rigau*
64. A Cultura do Morangueiro, *Luis Alsina Grau*
65. Emissores-Receptores (Walkies-Talkies), *P. Duranton*
66. Iniciação à Fotoelectrónica, *Heinz Richter*
67. Doces e Conservas de Fruta, *Robin Howe*
68. A Criação de Hamsters, *C. F. Snow*
69. A Criação de Porcos, *Roy Genders*
70. Calendário do Horticultor, *Luis Alsina Grau*
71. Jogos Electrónicos, *F. G. Rayer*
72. Cultivo de Cogumelos e Trufas, *Alejo Rigau*
73. Aprenda Televisão a Cores, *Gordon J. King*
74. Gravação em Fita Magnética, *Ian R. Sinclair*
75. Poda de Árvores e Arbustos, *Roy Genders*
76. Como Treinar o Seu Cão, *E. Fitch Daglish*
77. Instrumentos de Medida e Verificação, *Heinrich Stöckle*
78. A Criação de Caracóis, *Matías Josa*
79. Rádio — Fundamentos e Técnicas, *Gordon J. King*
80. Como Fazer Gelados, *Sylvie Thiébault*
81. Iniciação à Jardinagem, *Noel Clarasó*
82. A Congelação dos Alimentos, *Suzanne Lapointe*
83. Windsurf — Prancha à Vela, *Ernstfried Prade*
84. Raças de Cães, *O. Hasselfeldt*
85. Rummy e Canasta, *Claus D. Grupp*
86. A Encadernação, *Annie Persuy*
87. Aprenda Electricidade, *Heinz Richter*
88. Taxidermia, Embalsamamento de Aves e Mamíferos, *Harry Hjortaa*
89. Jogging — Correr para Manter a Forma, *Werner Sonntag*
90. ABC da Cozinha Chinesa, *Sonya Richmond*
91. Jogos T. V., *C. Tavernier*
92. Amplificadores de Som, *Richard Zierl*
93. O Livro do Poker, *Claus D. Grupp*
94. Aprenda a Desenhar, *Rose-Marie de Prémont e Nicole Philippi*
95. O Minitrampolim na Escola, *Sonja Helmer Jensen e Klaus Dano*
96. Jogos de Luzes e Efeitos Sonoros para Guitarras, *B. Fighiera*
97. O Cultivo do Tomate, *Louis N. Flawn*
98. Pilhas Solares, *F. Juster*
99. A Criação Doméstica de Coelhos, *C. F. Snow*

100. Iniciação ao Futebol, *Wieland Männle e Heinz Arnold*
101. Horóscopos Chineses, *Georg Haddenbach*
102. Guia Prático de Mancenaria, *Charles H. Hayward*
103. Andebol, *Fritz e Peter Hattig*
104. Dispositivos Anti-Roubo, *H. Schreiber*
105. Perus, Pintadas e Codornizes, *Jerome Sauze*
106. Crepes — Doces e Salgados, *Florence Arzel*
107. Aperitivos e Entradas, *Myrette Tiano*
108. Ténis de Mesa, *Leslie Woollard*
109. Aprenda Surf, *R. Abbott e M. Baker*
110. Futebol — Técnica e Táctica, *Kurt Lavall*
111. A Vaca Leiteira, *Colin T. Whittemore*
112. O Cubo Mágico, *Josef Trajber*
113. O Perdigueiro Português, *José M. Correia*
114. Pizzas e Massas à Italiana, *Marieanne Ränk*
115. O Cubo Para Quem Já o Faz, *Josef Trajber*
116. A Pirâmide Mágica, A Torre, O Barril do Diabo, *M. Mrowka-W. J. Weber*
117. Gansos e Patos, *Marie Mourthe*
118. Iniciação ao Kung Fu, *A. P. Harrington*
119. Electrónica e Fotografia, *Hanns-Peter Siebert*
120. O Livro da Fortuna, *Douglas Hill*
121. Construção de um Alimentador de Corrente, *Waldemar Baitingu*
122. Hóquei em Patins, *Francisco Velasco*
123. Técnicas de Tiro, *Anton Kovacic*
124. Aprenda a Tricotar, *Uta Mix*
125. ABC da Patinagem, *Christa-Maria e Richard Kerler*
126. A Pesca e os seus Segredos, *Armand Deschamps*
127. O Osciloscópio, *R. Rateau*
128. Guia Prático da Banda do Cidadão, *T. M. Normand*
129. Sumos e Batidos, *Manfred Donderski*
130. Introdução à Programação de Microcomputadores, *Peter C. Sanderson*
131. Aprenda Croché, *Uta Mix*
132. ABC do Microprocessador, *P. Mélusson*
133. Guia Prático de Basic, *Roger Hunt*
134. Introdução à Electrónica Digital, *Ian R. Sinclair*
135. ABC do Video, *David Matthewson*
136. Fotografia em Movimento, *Don Morley*
137. Guia Prático de Cobol, *Ray Welland*
138. Fotografia a Pequena Distância, *Sidney F. Ray*
139. Guia Prático da Canaricultura, *Dr. Manuel Gonçalves*
140. Minielectrónica para Amadores, *Heinz Richter*
141. ABC da Programação de Computadores, *John Shelley*
142. TAROT — O Futuro pelas Cartas, *Edwin J. Nigg*
143. ABC da Equitação, *Dorothy Johnson*
144. Como Programar o seu 2 × 81, *P. Gueulle*
145. 100 Avarias TV e a Maneira Rápida de as Detectar, *P. Duranton*
146. ABC da Horticultura, *Louis Giordiano*
147. BASIC para Microcomputadores, *A. P. Stephenson*

A. P. STEPHENSON

BASIC PARA MICROCOMPUTADORES

EDITORIAL PRESENÇA

Título original:
BEGINNER'S GUIDE TO BASIC
PROGRAMMING

Tradução de Conceição Jardim e Eduardo Nogueira
Fotografia da Capa de Gil Montalverde

EDITORIAL PRESENÇA, LDA.
Rua Augusto Gil, 35-A — 1000 LISBOA

PREFÁCIO

Os computadores já não pertencem ao domínio exclusivo do cientista, do industrial ou do programador profissional. Versões pequenas, mas surpreendentemente potentes, estão a tornar-se um lugar comum na escola e em casa. Como resultado, verificou-se um aumento dramático no número de entusiastas ansiosos por dominar a arte da programação em BASIC, a linguagem mais usada para «conversar» com o computador.

Já foram escritos muitos livros sobre BASIC mas, como a programação é mais uma arte do que uma ciência haverá sempre lugar para mais um. Este livro é literalmente um guia para *principiantes* e terá pouco que interesse aos veteranos, ainda que estes possam encontrar um ou outro aspecto que talvez lhes mereça um sorriso condescendente. Tanto quanto possível, evitou-se a gíria da especialidade, mas num tema como a programação nunca é possível ter-se grande êxito... Seria difícil descobrir outro assunto com tanta gíria e palavras sonantes como a programação. Felizmente, verifica-se muitas vezes que quanto mais impressionante é a palavra menos impressionados ficamos quando descobrimos o seu significado.

Os programas apresentados como exemplos não têm quaisquer pretensões, e foram escolhidos apenas para ilustrar os princípios expostos. Resistimos à tentação de encher as páginas com programas «úteis», considerando que tal apenas levaria o leitor a copiá-los e não a compreendê-los. A única maneira de aprender a programar é fazendo-o. O leitor terá muito mais satisfação em conseguir realizar um programa que funcione, por muito sim-

ples que seja, do que em introduzir no computador uma obra-prima escrita por outra pessoa.

Quero agradecer a James Brocklebank, um colega e um amigo, por ter corrigido o manuscrito. A sua habilidade com a caneta e o tinteiro (de tinta vermelha...) tem de ser vista para se acreditar nela!

A. P. S.

1

O QUE É O BASIC?

Os computadores possuem uma «inteligência» bastante limitada. Já foi afirmado que um computador gigante possui uma capacidade mental inerente um pouco superior à de uma minhoca mas inferior à de um insecto. Seja como for, a sua falta de inteligência é contrabalançada pela sua extraordinária velocidade de funcionamento pelo que devemos considerar o computador como uma espécie de atrasado mental muito rápido, capaz de obedecer a ordens sem as questionar, sem iniciativa e sem apresentar sinais de cansaço ou aborrecimento. Infelizmente, trabalha exactamente como lhe dizem. Ao contrário de um ser humano, o computador não é capaz de interpretar as ordens de acordo com o espírito que lhes preside. Quando confrontado com uma ordem que não reconhece faz precisamente aquilo que qualquer atrasado mental faria... nada! É importante compreender desde início que todas as ordens que damos ao computador devem ser perfeitas, até à última vírgula, nunca podendo conter quaisquer palavras estranhas ao seu limitado vocabulário.

Qual é então o seu vocabulário? No que se refere ao *hardware* do computador (aos seus parafusos e porcas...), os únicos dois caracteres que compreende são o 0 e o 1. Nos primeiros tempos esta era a única maneira de conversar com ele, e o pobre programador era obrigado a dar-lhe ordens actuando sobre fiadas de interruptores, que quando estavam «ligados» representavam um 1 e quando estavam «desligados» representavam um 0. Desde então

conseguiu-se descobrir o modo de o levar a aceitar linguagens menos primitivas, mas tratou-se certamente de uma luta árdua... Primeiramente, substituiram-se os interruptores por um teclado; cada tecla em que se carregava enviava para o computador uma fiada de 1's e 0's representando um código especial. Por exemplo, ao carregar no caracter C «dizia-se» ao computador: 1000011. O código que se usava não era muito importante, desde que fosse usado sempre do mesmo modo.

Se bem que esta entrada (input) * por teclado constituísse sem dúvida uma melhoria, as ordens dadas ao computador (o *programa*) eram ainda orientadas em termos da máquina e não do homem, exigindo grande habilidade e experiência do programador. Compreendeu-se gradualmente que era possível levar o computador a entender qualquer linguagem desde que alguém fosse capaz de escrever um programa de *tradução*. Escrever esse programa era (e ainda é) uma tarefa colossal, mas depois de escrito e armazenado no interior da memória do computador tornava-se possível apresentar-lhe programas escritos uma espécie de palavreado inglês...

Existem dois tipos de programas de tradução:

1. *Assemblers*, que nos permitem escrever o *código-máquina* numa forma mais fácil e menos susceptível de erros. Se bem que o código-máquina seja extremamente eficiente em termos de velocidade de execução e de economia de memória, é ainda uma maneira difícil de avançar na arte da programação.
2. *Compiladores e intérpretes*. Se bem que existam diferenças subtis entre estas duas categorias, têm em comum o objectivo de traduzir uma linguagem de alto nível para o código-máquina equivalente. O termo «alto nível» significa aqui que se trata de linguagens que se assemelham a uma língua falada (normalmente o inglês).

---

* Preferencialmente utilizar-se-á o termo «input» em vez de «entrada».

A primeira das linguagens de alto nível foi o FORTRAN, e se bem que frequentemente actualizada e modificada esta mantém ainda a sua arquitectura original. Outra linguagem «tradicional» é o COBOL, que ainda é bastante utilizada, em particular no campo comercial. Começaram, entretanto, a surgir outras linguagens, de início lentamente mas depois com uma frequência cada vez maior. O FORTH, o APL, o PASCAL, o ADA, o PL1, o CORAL, o COMAL e o ALGOL são alguns exemplos, e para não ofender ninguém, fazemos notar que a ordem pela qual as referimos não reflecte de modo algum o estatuto que têm actualmente. De facto existem hoje tantas linguagens, cada uma delas orgulhando-se de certas vantagens sobre as outras, que o principiante se sente numa verdadeira Babel... Mas tal como nas línguas humanas, ninguém consegue dar uma resposta definitiva, livre de partidarismo de qualquer «nacionalismo», à pergunta «Qual é a melhor?». É no entanto possível obter algum acordo quanto à superioridade de certas línguas humanas em certos campos especializados. Por exemplo, o inglês foi inventado por Shakespeare para poder escrever as suas peças...

O mesmo acontece com as linguagens de alto nível dos computadores. Cada uma delas possui algumas características que a tornam superior às outras sob determinados pontos de vista. O COBOL é portanto a linguagem tradicional no comércio, tendo sido concebida aliás com esse objectivo. O FORTRAN e o ALGOL são particularmente interessantes para os engenheiros e cientistas. Cada uma destas linguagens possui aquilo que poderemos designar por uma «respeitabilidade» na profissão, um termo que não deve ser confundido com «excelência». Por outro lado não são as linguagens mais fáceis de dominar.

Em 1 de Maio de 1964, John Kemeny e Thomas Kurtz, do Dartmouth College, New Hampshire, Estados Unidos, lançaram uma nova linguagem a que chamaram BASIC. Este BASIC original passou a ser conhecido por «versão Dartmouth» a fim de ser distinguido de outras variantes mais modernas. O principal objectivo que influenciou a

concepção do BASIC foi que seria fácil de aprender e utilizar. De facto, as letras que formam o seu nome são as iniciais de «*Beginners All-purpose Symbolic Instruction Code*», o que indica o desejo original de o tornar compreensível para «principiantes». Se bem que estes consideram de facto o BASIC razoavelmente fácil de aprender, seria no entanto incorrecto considerar esta linguagem como uma espécie de trampolim para o estudo de outras. O BASIC é um fim em si mesmo, e não deve ser tratado como uma espécie de exercício para aprendizes. Aqueles que pretendam operar um computador com um mínimo de estudo prévio verificarão aliás que o BASIC possui poucos rivais sérios. Alguns autores tendem a denegri-lo e, portanto, será talvez conveniente começarmos por considerar os seus defeitos e as suas virtudes.

As virtudes do BASIC

1. É provavelmente a linguagem mais fácil de aprender.
2. Os erros de programação que ofendem a *sintaxe* (gramática) da linguagem podem ser corrigidos facilmente, permitindo o uso imediato do programa.
3. Por boas ou más razões, é a linguagem mais usada no campo dos microcomputadores. Se o leitor conhecer o BASIC, terá muitos amigos desejosos de partilharem as suas dificuldades e de o ajudarem na fase inicial do seu estudo.
4. O BASIC é uma linguagem «amistosa», livre de muitas das regras esquisitas existentes nas linguagens concebidas por puristas.

Os defeitos do BASIC

1. É lento em execução. Isto deve-se ao método de tradução do programa para código-máquina. O programa de tradução do BASIC é chamado *intérprete*. Quando se introduz na máquina um programa em BASIC, cada linha é

traduzida e executada antes de se passar à linha seguinte. A maior parte das outras linguagens utilizam um programa de tradução designado por *compilador*, que separa a tradução da execução. O programa é primeiramente «compilado» em código-máquina e em seguida pode ser executado completamente nesta versão.

2. O BASIC é pouco económico no uso da memória porque o programa originalmente escrito deve ser guardado naquela para poder ser executado sempre que necessário. No caso de uma linguagem «compilada», o programa original (chamado «programa fonte») deixa de ser necessário, sendo integralmente substituído pela sua versão em código-máquina (chamada programa «objecto»).

3. O BASIC é uma linguagem «não estruturada». O termo «estrutura» é uma daquelas palavras em moda que tendem a penetrar nos círculos profissionais e se referem a um conjunto de regras e regulamentos definidos por certos autores influentes. Sem aprofundar demasiado o assunto, um programa bem estruturado é fácil de seguir, está livre de saltos «sem nexo» e consiste numa colecção de «módulos» que podem ser verificados separadamente. O termo «estrutura» é também útil para aqueles que se especializam em críticas, porque em última instância qualquer programa, apesar de perfeitamente aceitável, pode ser sempre condenado devido à sua «má estrutura». O ALGOL é a linguagem estruturada tradicional, mas o PASCAL está mais em moda.

Quando falamos de uma linguagem bem estruturada queremos dizer que é fácil seguir nela uma estrutura formal porque esta constituía já um critério da sua concepção. Isto não significa no entanto que seja impossível escrever programas estruturados em BASIC; é apenas um pouco mais difícil fazê-lo, porque a linguagem foi concebida tendo apenas como objectivo a simplicidade.

Dialectos BASIC

Nenhuma linguagem se mantém definitivamente pura. As diferenças de dialecto e de idioma local podem muitas

vezes provocar confusões nas conversas de todos os dias. Até certo ponto isto é também verdadeiro para o BASIC devido ao facto de a versão original Dartmouth ter sido sujeita a várias modificações através dos anos, tendo como resultado a existência de pequenas mas aborrecidas variações em diferentes marcas de computador e mesmo em diferentes variantes do mesmo modelo básico. Não existe um BASIC «standard», pelo que certamente algumas das técnicas que iremos referir poderão não ser imediatamente utilizáveis na sua máquina. Sempre que possível, serão indicadas diferenças dialectais, tendo-se o cuidado de evitar utilizar um número excessivo de palavras «não Dartmouth». Muitos microcomputadores bastante populares utilizam uma forma razoavelmente normalizada a que se chama «Microsoft» BASIC (esta palavra é marca registada), se bem que existam já vários subconjuntos como o Microsoft 2.0, o Microsoft 3.0 e até o Microsoft 5.0. Algumas das formas mais modernas incluem palavras adicionais que permitem uma melhor estruturação.

Para superar o problema da velocidade já mencionado, começaram a surgir no mercado algumas versões «compiladas» do BASIC, e no que se refere à velocidade de execução é enganador insistir demasiado na lentidão desta linguagem. A grande maioria dos programas são executados tão rapidamente, do ponto de vista humano, que parecem ser praticamente instantâneos. Apenas certos programas traem a inerente lentidão desta linguagem, e apenas em algumas das suas partes. A fim de ultrapassar este problema é possível recorrer a alguns truques, alguns dos quais serão referidos mais adiante.

O hobby da programação

Há apenas alguns anos, a simples sugestão de que o homem normal poderia vir a comprar ou mesmo a estar interessado num computador teria sido considerada absurda. O desenvolvimento deste hobby para pessoas da classe média deveu-se a dois factores. O primeiro con-

sistiu na introdução do *microprocessador*, praticamente um computador numa pequena «pastilha» de silício. O segundo factor deveu-se principalmente à previsão e iniciativa de alguns pioneiros que juntaram algumas ideias e, evidentemente, um microprocessador, circuitos de memória e um teclado. Pondo tudo isto dentro de uma caixa, e ligando-a ao receptor de televisão, verificou-se que se obtinha um autêntico computador que podia ser ensinado a falar BASIC. Surgiu assim o Apple em 1973. O Apple continua a ser muito usado, tendo sofrido já bastantes melhoramentos. Seguiram-se o Tandy e o PET, e hoje o mercado está a ficar saturado de marcas.

Em muitos países existem clubes de «fans» do computador em praticamente todas as cidades importantes, e é grande a quantidade de revistas sobre o assunto que adornam as bancas de lojas especializadas em hobbys. Computing Today, Practical Computing, Personal Computer World e Your Computer são alguns exemplos disto, e constituem auxiliares muito valiosos para todos aqueles que têm o «bichinho» da programação. A maior parte dos programas publicados nestas revistas são em BASIC, convindo investigar todas as linhas para o caso de existir algum procedimento interessante e até então desconhecido que possa ser incorporado nos seus próprios programas. Você não se pode dar ao luxo de ser um lobo solitário. Estudar um único livro de BASIC (mesmo este...) nunca poderá ser suficiente para o qualificar como «programador»; você necessitará da sabedoria colectiva dos outros e da consolidação de conhecimentos produzida pela experiência. Nunca se preocupe se os seus programas não trabalharem da primeira vez, porque isso raramente acontece... mesmo quando o programador é um perito. Em vez de se deixar abater com pena de si mesmo, encolha os ombros e descubra porquê. A maior parte dos falhanços devem-se a simples erros de sintaxe, e são inevitáveis nas primeiras fases de aprendizagem, até você conseguir habituar-se à linguagem. Existem poucos prazeres intelectuais tão agradáveis como a resolução do último erro. De certo modo, a existência de muitos erros nas

primeiras fases do seu hobby pode ser uma vantagem no seu «crescimento» como programador. Um caminho demasiado fácil pode dever-se apenas à sorte e dar-lhe uma confiança em si próprio não muito desejável, levando-o ao desespero quando finalmente encontrar um problema.

Que tipo de programas?

Uma coisa é aprender a programar. Outra é decidir o tipo de programa que se pretende. A imaginação é aqui muito importante, juntamente com um certo «instinto» no que se refere aos limites da sua habilidade e à capacidade de apreciar rapidamente a quantidade de memória de que necessitará. Os objectivos da programação podem dividir-se em cinco categorias principais:

1. *Tratamento de números.* Trata-se de programas que fornecem os resultados numéricos de problemas de engenharia ou física, que podem variar desde o cálculo do número de tijolos necessário para construir um anexo à sua casa de campo até à resolução das equações diferenciais necessárias para conduzir um míssil até ao seu alvo. Surpreendentemente, este tipo de programas são bastante simples de escrever... desde que saiba onde descobrir as equações.
2. *Armazenamento e acesso de dados.* Isto envolve o tratamento de fichas computadorizadas, inventários, listas de pagamentos e manipulação de dados necessários às actividades comerciais. A organização das informações sobre colecções de selos, acontecimentos desportivos, etc., estará igualmente compreendida neste título.
3. *Educação.* O computador é um método de treino ideal, particularmente na forma de perguntas e respostas. Os programas que apresentam perguntas e pedem respostas podem ser muito bons como exercícios de consolidação de conhecimentos. O jovem pode muitas vezes ser persuadido (contra a sua natureza...) a gastar bastantes horas à frente de um teclado tentando «ganhar» ao computador,

sem se aperceber de que foi enganado e está de facto a aprender!

4. *Jogos.* Os jogos computadorizados são hoje uma verdadeira legião. São fascinantes, e permitem obter um excelente treino de programação devido à animação de que necessitam. É nesta área que a capacidade para programar pode estar subordinada à posse de uma imaginação bastante rica. Já foram escritos tantos programas para jogos que se torna cada vez mais difícil descobrir uma ideia original. Muitos deles são já uma simples repetição de temas vulgares.

5. *Sistemas de comando computadorizado.* Trata-se de uma categoria completamente diferente das anteriores, dado que exige um certo conhecimento do hardware além da habilidade de programação (*software*). O computador pode ser usado para controlar sistemas externos, como comboios eléctricos, modelos Meccano, iluminação de palco em teatro de amadores e até complexos sistemas de alarme.

Como começar

Suponhamos que você acabou de adquirir um computador. Trouxe-o para casa e depois da habitual «vista de olhos» pelo manual de instruções pegou na ficha e meteu-a na tomada. Que acontece? Nada, além de aparecer no visor uma mensagem inexpressiva do tipo READY («Pronto»). E talvez até este prazer simples lhe seja negado. Talvez tenha de se satisfazer com um ponto luminoso no canto superior esquerdo. A razão de nada acontecer é que a máquina limita-se a estar «sentada» à nossa frente, à espera que lhe digamos alguma coisa... Por outras palavras, necessita de um *programa,* e muito provavelmente este terá de ser escrito em BASIC.

Partindo do princípio de que você nada sabe sobre esta linguagem, ou aliás sobre qualquer outra, terá de estudar duas disciplinas. Primeiro terá de aprender o vocabulário e a sintaxe da linguagem. Isto significa me-

morizar uma colecção de *palavras-chave*, como lhes chamaremos neste livro, compreender o efeito *exacto* de cada uma delas e o simbolismo da pontuação. Deve aprender não só o que pode fazer, mas também, e principalmente, o que não pode. Depois disto, que é relativamente fácil de aprender por qualquer pessoa medianamente inteligente, deve aprender a escolher as palavras-chave de modo a produzir o efeito desejado. A capacidade para realizar esta parte da tarefa não é principalmente intelectual. Exige uma outra coisa: uma mistura de imaginação, instinto, meticulosidade, paciência, e talvez uma natureza um tanto pessimista, sempre à espera de problemas. Se não tiver todas estas qualidades não desespere, porque se perseverar acabará por adquiri-las — principalmente o pessimismo...

Nunca se esqueça de que o computador só fará aquilo que lhe disser, não aquilo que você quiser que ele faça. Evite o erro vulgar que consiste em culpar o computador pelos seus próprios erros. Os computadores falham por vezes, mas isto acontece muito raramente e de um modo tão catastrófico que nunca pode haver dúvidas sobre o seu estado de saúde em qualquer momento... Existe no entanto uma doença típica dos computadores a que se chama «crash» (se quiser usar um termo banal pode chamar-lhe «estoiro»...) do sistema. Os sintomas desta doença são verdadeiramente dramáticos, porque o teclado parece perder todo o comando da máquina e o visor encher-se-á de sinais estarrecedores... Isto poderá acontecer por vezes se você «ofender» inadvertidamente a máquina tomando alguma atitude menos própria relativamente ao seu sistema interno. Não se preocupe, porém, pois a doença apenas tem a ver com o software e não é caro «curá-la». Carregue simplesmente no botão RESET (se existir) ou desligue e ligue novamente o computador. Terá provavelmente perdido o seu programa mas bastar-lhe-á comunicá-lo novamente à máquina. Apenas terá perdido tempo.

As mensagens que se apressam a surgir no visor assim que você comete um erro de sintaxe ou outro só são preocupantes até você se habituar a elas. O computador

contém uma grande quantidade de mensagens de erro armazenadas nos seus «cérebros» de silício, e você deve até estar agradecido por este facto. Estas mensagens são um óptimo auxiliar da programação e de modo algum devem ser consideradas hostis. Recorde apenas que nada daquilo que você fizer com o teclado pode danificar a máquina.

Venda de programas

Depois de aprender a escrever programas com alguma confiança você poderá recuperar parte do custo do computador ou mesmo obter algum lucro tentando publicar os seus programas. As revistas já mencionadas adoram pagar programas quando estes são acompanhados de algumas observações explicativas sobre o seu funcionamento. Com sorte talvez consiga até vendê-los a empresas de software, ou anunciá-los num jornal. Seja o que for que aconteça, nunca desespere antes da trigésima tentativa...

RESUMO

Os computadores têm uma inteligência limitada. Obedecem a ordens legítimas sem fazerem perguntas e sem tomarem qualquer iniciativa.

Conversar com os computadores exige um conhecimento da linguagem que estes compreendem.

Usar a própria linguagem da máquina recorrendo a um «assembler» é muito eficaz mas bastante difícil.

As linguagens de alto nível são traduzidas para código-máquina por um programa «residente» designado por *compilador* ou *intérprete*.

A maior parte das linguagens de alto nível são compiladas. O programa original, tal como foi inicialmente escrito, é chamado programa *fonte*, e depois de ser integralmente traduzido para código-máquina passa a cha-

mar-se programa *objecto*. O ALGOL, o FORTAN, o PASCAL e o COBOL são exemplos de linguagens compiladas.

O BASIC utiliza um programa *intérprete* para traduzir em código-máquina sendo as linhas traduzidas uma a uma antes da execução. O programa fonte deve portanto encontrar-se na memória da máquina sempre que se quer utilizá-lo.

O BASIC é fácil de aprender, funciona com alguma lentidão e não se presta facilmente aos ditames da programação estruturada. Apesar disto, no entanto, o BASIC continua a ser extremamente popular, devendo de facto ser considerado como a principal linguagem de nível elevado usada em microcomputadores; não se admire no entanto por encontrar diversas variantes desta linguagem.

Antes do mais é necessário conhecer as palavras-chave e a sintaxe da linguagem. Ligar estas palavras entre si de modo a conseguir formar um programa é algo que apenas se aprende com a experiência.

Nunca considere hostis as mensagens de erro que o computador lhe apresenta. São necessárias para o auxiliar na programação.

A programação, mesmo a um nível amador, pode ser lucrativa.

2

INSTRUÇÕES, ORDENS E VARIÁVEIS

Uma *instrução* é uma ordem única ao computador. Um *programa* é um conjunto de instruções, tendo cada uma delas um *número de linha* escolhido arbitrariamente.

Quando o programa é passado no computador, este executará cada uma das instruções pela ordem estrita indicada pelo número de linha, a menos que receba outras ordens. Vejamos qual pode ser a aparência de um programa, com os números de linha à esquerda:

```
100 LET A = 20
110 LET B = 5
120 LET Z = A + B
130 PRINT Z
140 END
```

Se tiver acabado de escrever este programa em papel, pode sentar-se à frente do teclado e fazê-lo *entrar* na máquina. Quando tiver acabado de «dactilografar» cuidadosamente uma linha, deve carregar na tecla RETURN[1] a fim de informar o computador de que essa linha acabou. Até premir esta tecla a linha mantém-se apenas no visor, não tendo sido ainda armazenada pelo computador. Se cometer um erro, pode usar as várias teclas de edição

---

[1] Em vários computadores a tecla a premir é o ENTER ou o NEW LINE em vez de RETURN, pelo que convirá informar-se no manual fornecido com o computador — N.R.

para apagar, voltar atrás, etc. Só quando estiver satisfeito deve carregar na tecla RETURN e passar à linha seguinte.

Escolha do número da linha

Se bem que a escolha seja arbitrária, convém espaçar os números de linha o bastante a fim de deixar espaço para o caso de mais tarde querer acrescentar linhas entre duas linhas consecutivas, por exemplo a 120 e a 130. Escreve muito simplesmente a instrução em falta no final do programa, dando-lhe um número de, por exemplo, 125. O computador reordenará automaticamente as suas instruções segundo os números de linha na fase RUN. Se tiver escolhido originalmente os números 1, 2, 3, 4, não disporá de espaço para inserir linhas novas.

A maior parte dos BASIC's permitem-lhe colocar mais do que uma instrução numa linha desde que cada uma esteja separada por dois pontos (:). O programa anterior pode ter escrito de uma forma mais concisa:

```
100 LET A = 20: LET B = 5: LET Z = A + B
110 PRINT Z: END
```

Se bem que esta forma concisa permita poupar alguma memória, não é muito fácil de seguir; convém portanto não abusar do seu emprego nos programas mais complicados porque pode tornar-se mais difícil descobrir um erro. Deve notar igualmente que o comprimento de uma linha *de computador* pode ser superior ao da linha do visor, mas é necessário consultar o manual fornecido com a máquina para esclarecer este ponto.

O BASIC é uma linguagem de «formato livre», na qual os espaços entre os caracteres são opcionais e podem ser usados livremente a fim de melhorar a aparência do programa. No que diz respeito ao computador, não importa que você escreva:

```
100 LET A = 20
```

sob a forma

```
100 LET A = Z 0
```

Existe, no entanto, uma restrição relativamente a palavras-chave como LET e PRINT; não são permitidos espaços *entre* os caracteres de uma palavra-chave.

Se bem que ainda não tenhamos tentado explicar o programa anterior, não deve ser difícil compreender que, depois de passado, este programa levará o computador a imprimir o número 25, o que é suficiente para se poder dizer que o BASIC é «fácil». Como é óbvio, os programas com verdadeiro valor prático terão uma forma mais complexa, mas manterão apesar disso uma aparência lógica.

ORDENS

Definiu-se uma instrução como uma ordem única no interior de um programa. As *ordens* aplicam-se por outro lado a todo o programa. Ao contrário das instruções, as ordens não possuem um número de linha e são executadas imediatamente após premir a tecla RETURN.

As duas ordens apresentadas aqui são a LIST e a RUN: as restantes serão referidas à medida que forem necessárias.

A ordem LIST

Quando tiver acabado de dar entrada ao seu programa e terminado a tarefa de edição, talvez deseje observar como aparece o programa no computador. Por outras palavras, quer pedir ao computador uma *listagem*. Existem várias maneiras de fazer isto:

LIST	Apresentará todo o programa
LIST n	Apresentará uma linha com um determinado número
LIST n-m	Apresentará todas as linhas entre n e m, inclusive
LIST -n	Apresentará todas as linhas até n, inclusive
LIST n-	Apresentará todas as linhas a partir de n.

É conveniente pedir listagens durante o desenvolvimento do programa, porque dizem-nos o que se encontra de facto no computador e não o que nós pensávamos que lá estivesse... Depois de uma listagem podem-se alterar linhas, substituí-las por novas ou até apagá-las. Para apagar uma linha completamente escreve-se o número da linha e carrega-se na tecla RETURN. Depois de terem sido realizadas quaisquer modificações pede-se uma nova listagem, e continua-se o trabalho até este ficar terminado.

A ordem RUN

Depois de estarmos satisfeitos com a listagem final, a fase seguinte consiste em verificar se o programa funciona. Usa-se para tal a ordem RUN, que tal como a LIST pode possuir várias formas:

RUN — Executará o programa, começando pelo menor número de linha até atingir a afirmação STOP ou END ou até acabarem os números de linha.

RUN n — Começará a executar o programa a partir da linha n.

De definição anterior da ordem RUN parece concluir-se que as afirmações STOP e END são arbitrárias, dado que o computador pára sempre que não existam mais números de linhas. É no entanto mais lógico incluir uma destas ordens no final de programa. De qualquer modo, o leitor aprenderá mais tarde que STOP não se deve encontrar necessariamente no final físico do programa, e que pode existir mais do que um STOP num mesmo programa.

RUN n é útil quando se quer experimentar *secções* determinadas de um programa ao tentar corrigir erros. A colocação de um STOP temporário no final da secção destaca-o do restante. A linha STOP pode ser retirada mais tarde.

O que acontece depois de escrevermos RUN e carregarmos na tecla RETURN é muitas vezes decepcionante, e

se você tiver uma maneira de ser menos calma pode traduzir-se em danos físicos ao mobilário circundante. O resultado mais vulgar é nada, ou talvez uma observação excitante do tipo SYNTAX ERROR IN LINE 4679 (erro de sintaxe na linha 4679). Não esqueça as observações já feitas no capítulo anterior... O erro é seu, não do computador! Observe calmamente a linha em causa e corrija-a, em seguida escreva LIST, depois RUN e repita a sequência *ad infinitum* até começar a acontecer alguma coisa com interesse.

Uma outra ordem bastante simples é NEW. Escrever NEW e carregar em RETURN apagará da memória qualquer programa que aí se encontre. É um bom hábito escrever NEW antes de iniciar um programa novo, senão há sempre a possibilidade de aparecerem umas linhas misteriosas no meio do programa quando se começa a listá-lo, linhas essas que constituirão um resíduo do programa anterior.

OBSERVAÇÕES

Quando estamos a escrever e a dar entrada a linhas de programa pode-se compreender perfeitamente o raciocínio justificativo de cada linha, mas à medida que o programa se desenvolve é possível que a intenção inicial desapareça gradualmente da sua memória. Se o seu programa se recusa a trabalhar e você não sabe o que fazer, talvez lhe convenha pedir a ajuda de um amigo; mas se não lhe souber explicar qual o seu objectivo, pouca ajuda conseguirá. É portanto bom espalhar livremente umas tantas «observações» pelo seu programa, utilizando instruções REM.

A instrução REM

As instruções tendo como prefixo a palavra-chave REM aparecem na listagem mas são ignoradas pelo computador na fase RUN (execução). Podemos escrever notas

explicativas depois de uma afirmação REM, as quais nos ajudarão a compreender mais tarde o nosso programa. Por exemplo:

```
90 REM ***SOMAR DOIS NÚMEROS***
```

Se se tivesse incluido esta linha no exemplo do programa anterior, ela indicaria pelo menos o seu objectivo — e tornaria o programa um pouco mais «amistoso». Note os asteriscos, que não são essenciais mas são usados vulgarmente para sublinhar a observação nas listagens. Não existem quaisquer leis sintáticas numa afirmação REM pelo que nos é permitido escrever o que quisermos, pois como já se disse o computador ignora estas afirmações (exceptuando o seu número de linha) quando passa o programa. Um aviso: nunca escreva a instrução REM no início de uma linha se pretende colocar em seguida dois pontos e escrever uma afirmação «verdadeira». Se o fizer, o computador pensará (porque é estúpido) que toda a linha faz parte da afirmação REM. Suponhamos que você escreve:

```
500 REM **ACHAR MÉDIA**: A = S/N
```

O computador não executará A = S/N porque julga fazer parte da REM. Por outro lado:

```
500 A = S/N: REM **ACHAR MÉDIA**
```

é uma afirmação correcta, levando o computador a executar a instrução inicial e a ignorar depois a afirmação REM. Se bem que o computador ignore as REM's estas são armazenadas na memória e se forem em grande número ocuparão bastante espaço. Existe um meio termo entre a conveniência e a verbosidade.

VARIÁVEIS E INSTRUÇÕES DE ATRIBUIÇÃO

Você poderá querer usar um determinado número ou caracter várias vezes num mesmo programa, em vez de o

fazer entrar pelo teclado sempre que dele necessita. A versão Dartmouth permite-lhe atribuir um número a uma «variável» recorrendo à palavra chave LET. Consideremos um exemplo:

100 LET K = 256.278 [1]

Encontramos aqui uma instrução de atribuição, que colocará o número em memória sob o nome de variável K. Daí em diante, considerar-se-á que qualquer referência à variável K indicará o número 256.278, podendo este ser usado tantas vezes quantas se queira. Como estas instruções de atribuição são usadas frequentemente, considerou-se que era aborrecido ser forçado a escrever LET para todas elas, e os intérpretes BASIC mais modernos permitem-nos omitir esta palavra. Omitiremos igualmente LET nas instruções de atribuição que nos surjam daqui em diante. Pode-se aceitar o seguinte equivalente:

100 K = 256.278

O lado direito de uma afirmação deste tipo pode ser formado por qualquer *expressão* legítima como:

100 Z = K + B

Esta instrução indicará ao computador que deve somar K e B (independentemente dos números que lhes foram previamente atribuídos), colocando a soma resultante em Z.

Vejamos um aspecto muito importante:

O conteúdo das variáveis que se encontram no lado direito não é alterado.

A variável do lado esquerdo perderá o seu conteúdo anterior, sendo este substituído pelo novo resultado.

---

[1] Sendo a «conversação» com o computador realizada em inglês, usa-se o ponto entre algarismos como equivalente à nossa vírgula — N. T.

Por exemplo: se K continha 20, B continha 30 e Z continha 70, após a execução da afirmação 100 Z = K + B os conteúdos em causa passarão a ser K = 20, B = 30, Z = 50; o valor 70 que originalmente se encontrava em Z terá desaparecido definitivamente. Note bem isto, porque a confusão entre o que acontece a ambos os membros destas expressões constitui um erro bastante vulgar nas primeiras fases.

É «proibido» utilizar uma expressão à esquerda do sinal de igual. Assim, 100 A + B = K constituirá um erro de sintaxe. Este último aspecto sublinha o facto de uma atribuição de valor não ser de facto equiparável a uma equação matemática. Em muitos casos assemelha-se a uma equação, mas aí reside precisamente o seu perigo. Examine o seguinte:

100 A = A + 1

Esta afirmação é aceitável porque somará 1 ao conteúdo de A e colocará o resultado novamente em A; por outras palavras, *incrementará* o conteúdo de A. No entanto, de um ponto de vista matemático, esta expressão é obviamente absurda; não existe nenhum número que seja igual à soma de si mesmo pela unidade.

Nomeação de variáveis

O BASIC é bastante «esmerado» quanto aos nomes dados às variáveis, e discrimina claramente os números «reais», os números inteiros e aquilo a que em gíria se chama *strings* (variáveis de cadeia). Convém portanto defini-los.

Os números reais são números como 456, fracções decimais como 0,478, números mistos (parte inteira e parte fraccionária) como 458,578, e qualquer destes precedido por um sinal + ou —. Muitas vezes são referidos como números de «vírgula flutuante».

Os inteiros são números sem uma vírgula decimal.

As variáveis de cadeia são quaisquer caracteres: letras, números, símbolos de pontuação, sinais aritméticos, etc.

As regras de nomeação de variáveis devem ser rigidamente observadas, e são as seguintes:

Variáveis reais. Estas variáveis podem ser usadas de qualquer um dos modos seguintes:

Por uma única letra, por exemplo um S.

Por duas letras, por exemplo VG.

Por uma única letra seguida por um número único, por exemplo Z3.

É normalmente permissível usar letras extra a fim de tornar a lista mais informativa. É portanto possível chamar a uma variável VOLTS, mas existe o problema de o intérprete não distinguir entre duas variáveis para além dos dois primeiros caracteres. Se chamarmos a uma variável BLOOD e a outra BLOSSOM, no mesmo programa, o computador pensará que se trata da mesma coisa... o que pode conduzir ao caos.

Variáveis inteiras. As variáveis inteiras podem ser nomeadas do modo descrito para as anteriores, mas devem possuir o caracter "%" à frente do nome. Por exemplo, poderemos chamar-lhes D%, FG% ou V3%. Deve-se compreender que quando se utilizam nomes de variáveis inteiras em instruções de atribuição aquelas só podem conter números inteiros. Por exemplo:

```
100 S = 37.9
110 K% = S
```

Quando este programa é passado, S contém 37,9 mas K% conterá apenas 37 (a parte inteira do número real). Note que isto não equivale a um processo de arredondamento; em vez disso o número é truncado, dado que a parte fraccionária é pura e simplesmente eliminada. Mais concretamente, o número é «cortado», mantendo-se apenas o número inteiro imediatamente inferior; isto pode ser enganador quando se utilizam números negativos. Por exemplo:

```
100 S = — 6.1
110 G% = S
```

G% conterá — 7 porque é este o número inteiro imediatamente inferior a — 6,1. No caso de o leitor não ser muito versado em matemáticas (mas esteja descansado; também é esse o meu mal) convirá dizer que os matemáticos insistem em afirmar que — 7 é mais pequeno do que — 6...

As variáveis inteiras são úteis de dois modos. Em primeiro lugar, você poderá querer separar a parte inteira da parte fraccionária de um número de vírgula flutuante, um requisito que se torna bastante comum quando se realizam operações aritméticas. Em segundo lugar a passagem da base decimal para a binária obriga a uma série de divisões em que o resto (a fracção que resta) é armazenado separadamente.

Seria útil dar um exemplo que nos possa esclarecer sobre o modo como é possível separar um número de vírgula flutuante na sua parte inteira e na sua parte fraccionária:

```
100 I% = F
110 R = F — I%
```

Consideremos que o número de vírgula flutuante em F é digamos, 35,6. Ao passar o programa, I% conterá 35 e R conterá 35,6 — 35. Assim, F mantém o número original, mas em I% ficará agora a sua parte inteira e em R o resto.

Note que a escolha de variáveis tende a indicar a natureza do seu conteúdo. Este é um bom hábito em termos de programação, e torna mais fácil a leitura do programa durante uma listagem.

Variáveis de cadeia. Estas variáveis só podem ser designadas por nomes que terminem pelo sinal $, sendo já descritas. D$, GF$ e R3$ são exemplos de variáveis de cadeia correctas. Como já se disse, estas variáveis podem conter qualquer colecção de caracteres, incluindo mensagens em linguagem normal. Por exemplo:

```
100 A$ = «PERIGO! RADIAÇÃO»
```

Note as aspas que encerram o segundo membro da atribuição, que não são opcionais, mas sim *necessárias*. Se se esquecer delas obterá no visor uma mensagem de erro: MISMATCH ERROR. Isto significa que você usou uma variável de cadeia num dos membros e uma variável numérica no outro. Sem as aspas, o computador pensa que a sua mensagem é uma variável de vírgula flutuante porque, como se deve recordar, não existe qualquer restrição quanto ao número de caracteres de uma variável. É permissível usar algarismos em variáveis de cadeia, mas devem também encontrar-se entre aspas. Assim, 100 A$ = «45.56» está bem porque as aspas levarão o computador a considerá-las como caracteres de uma variável de cadeia. Do ponto de vista do computador não são números mas sim simples caracteres, e não se pode fazer operações aritméticas com variáveis de cadeia.

Se bem que não se possam realizar operações aritméticas normais, existe no entanto uma operação, semelhante à soma e designada por «concatenação», possível com estas variáveis. «Concatenar» significa aqui «juntar entre si». Por exemplo:

```
100 A$ = «EDITORIAL»
110 B$ = «PRESENÇA»
120 M$ = A$ + B$
```

M$ passará a conter EDITORIAL PRESENÇA. Note que as aspas propriamente ditas não são armazenadas. O sinal + usado na concatenação é o único sinal aritmético permissível entre variáveis de cadeia.

Existe normalmente um limite para o número total de caracteres que podem ser armazenados numa variável de cadeia; o habitual é 256, se bem que o leitor deva sempre consultar o manual da sua máquina. Não se esqueça de que ao contar o número de caracteres não deve esquecer os espaços em branco. As variáveis de cadeia permitem-nos fazer bastantes truques, mas é melhor deixar estes pormenores para outro capítulo.

Notação científica

A matemática e a física, particularmente a astrofísica, trabalham com números muito grandes e muito pequenos, em ambos os casos cheios de zeros. A notação científica normal representa 1 000 000 (um milhão) sob a forma 10^6, 3 000 sob a forma 3×10^3, 0,000015 sob a forma $1{,}5 \times 10^{-5}$, etc. Para evitar as dificuldades de representação do expoente, o BASIC utiliza uma notação especial com a forma mEp onde m é a mantissa e p o expoente. A letra E significa que o número que se segue é um multiplicador, uma potência de dez (o expoente). Esta notação pode ser usada em qualquer instrução de atribuição e a saída impressa surgirá automaticamente nesta forma se os números se encontrarem além dos limites de tratamento normal dos circuitos aritméticos.

Exemplos:

71 000 pode ser escrito sob a forma 7.1E4 ou 71E3
0,00056 pode ser escrito sob a forma 5.6E-4 ou 56E-5

Existe sempre um limite para o expoente máximo, normalmente entre 30 e 40 conforme a máquina.

Dígitos significativos

O rigor da aritmética realizada é medido pelo número de dígitos significativos que aparecem na «resposta» impressa. Este número varia com o computador, mas é normalmente de nove dígitos, o que na grande maioria dos cálculos é mais do que suficiente. De facto, o número de dígitos é por vezes demasiado grande, tornando-se pouco prático. Em Inglaterra, um salário semanal de, por exemplo, £125,658347 seria um exemplo absurdo de rigor. Felizmente, é fácil escrever segmentos de programa que arredondem os números de modo a terem proporções razoáveis.

Em certos casos, no entanto, nove dígitos é insuficiente. Os contabilistas certamente ficarão embaraçados

se tiverem de trabalhar com um limite de nove dígitos. Dada a «espiral inflacionária» dos nossos tempos, as quantias com muitos zeros transformaram-se em simples «trocos»... Tendo em conta que as contas devem estar correctas até ao último tostão, ficaríamos limitados a valores até nove algarismos, não obtendo rigor em quantias superiores. Se bem que seja de facto este o limite no que se refere ao rigor interno da aritmética realizada pelo computador, é possível porém transformar os nove dígitos em qualquer número deles escrevendo segmentos de programa múltiplos desde que a perda de tempo assim causada seja aceitável.

OPERAÇÕES ARITMÉTICAS E MATEMÁTICAS

As simples operações aritméticas utilizam os seguintes operadores standard:

—	sinal negativo, ou subtracção
+	sinal positivo, ou adição
*	multiplicação
/	divisão
↑	exponenciar (potenciar)
( )	parêntesis

Aviso: a multiplicação «implícita», do tipo Z — XY, é ilegal; deve possuir um asterisco (*) entre as variáveis: Z = X*Y. O asterisco é usado para multiplicação em vez do sinal X, dado que é possível confusões com a letra idêntica.

Os parêntesis podem ser usados do mesmo modo que em matemáticas «normais», a fim de indicar a ordem das operações. A ordem, ou *precedência,* na ausência de parêntesis é normalmente a seguinte:

Primeiro as operações entre parêntesis;
Depois as potenciações;
Multiplicações;
Divisões;
Finalmente, adições e subtracções.

A menos que tenha uma certeza quanto à ordem de operações (que pode variar entre as diferentes versões de BASIC), é mais seguro utilizar os parêntesis profusamente nas equações mais complexas. Não se esqueça de contar os parêntesis, porque um número ímpar produzirá certamente um erro de sintaxe. Os parêntesis podem ser utilizados dentro de outros. É necessária alguma prática para escrever as equações matemáticas normais em forma BASIC. Convirá estudar os exemplos seguintes, que indicam o modo de converter tais equações para BASIC:

Equações típicas	*Versão BASIC*
$A = bc/d$	`A = B*C/D`
$s = (k + b)^3$	`S = (K + B) ↑ 3`
$U = \frac{3 - (f + 4{,}5G)}{3 \times 10^{23}}$	`U = (3 — (F + 4.5*G))/3E23`
$d = 1 + \frac{g}{3} + g^{-3}$	`D = 1 + G/3 + G ↑ — 3`

Funções especiais (ver Apêndice B)

Os professores de matemática tratam o termo «função» com o tipo de reverência normalmente reservado aos chefes de estado. Aparentemente, as funções são coisas bastante estranhas que necessitam de muitas páginas para serem definidas rigorosamente, evitando a intromissão de impostores. No que nos diz respeito, uma *função* é uma palavra chave seguida de qualquer coisa dentro de parêntesis, que trata algumas das operações matemáticas mais comuns necessárias para completar os simples operadores aritméticos. Assim, o BASIC inclui a função seno que será usada para ilustrar o modo como se utilizam as funções especiais:

```
100 K = SIN(A)
```

Isto colocará o valor do seno do ângulo A na variável K. O conteúdo dos parêntesis é designado por *argumento* e a função é considerada como expressa em *radianos* e não graus.

Os radianos são um método natural de medir ângulos a partir da razão π (pi); os graus, se bem que muito populares, são medições arbitrárias devidas aos antigos que pensavam, erradamente, haver 360 dias no ano... ou algo de parecido. É fácil usar graus em caso de necessidade porque 1 graus = $\pi/180$ radianos. Certos teclados incluem o valor pi, mas se tal não acontecer pode-se considerá-lo como uma variável até ser necessário.

A variável presente entre parêntesis em qualquer função pode ser qualquer expressão complexa contendo outras funções, pelo que:

```
100 Z = SQR(L + SIN(K*G/H))
```

é perfeitamente legal.

Até o leitor ter bastante experiência (e até certo ponto mesmo depois disso) será bom dividir as equações extensas em módulos calculados separadamente. Há menos possibilidades de erro e é mais fácil descobri-los quando existem. Suponhamos que devemos usar a equação correspondente a funções do 2.º grau, que é:

$$Y = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

(notar que esta equação possui duas soluções).

Se bem que esta equação possa ser colocada numa instrução de atribuição única, é prudente dividi-la do seguinte modo:

```
100 D = B*B - 4*A*C
110 Y1 = (- B + SQR(D))/(2*A)
120 Y2 = (- B - SQR(D))/(2*A)
```

As duas soluções encontram-se em Y1 e Y2.

Uma equação como esta dá-nos uma oportunidade de discutirmos os erros de «divisão por zero» e de «quantidade ilegal» que tantas vezes aparecem no visor. Se A for zero, é fácil ver que o pobre computador se vê forçado a dividir por zero nas linhas 110 e 120. Mesmo os matemáticos objectam a isto, pelo que a reacção do computador é bastante compreensível. Um outro perigo desta equação é o de D ser negativo (se 4AC for maior do que B^2), porque a raiz quadrada de um número negativo é uma quantidade nebulosa e, no que diz respeito ao computador perfeitamente ilegal. A fim de impedir estas operações abortíveis, que obrigariam o computador a sair da execução do programa para o nível das ordens, teremos de incluir no programa aquilo a que se chamam «traps» («armadilhas»). Mais tarde, quando tivermos discutido mais alguns truques do BASIC, será fácil montá-las.

VARIÁVEIS SUBSCRITAS E QUADROS

O limite de dois caracteres na nomeação de variáveis pode ser restritivo, mas é possível identificar variáveis praticamente sem qualquer limite usando quadros subscritos. Estes podem existir em qualquer das três formas das variáveis BASIC:

A(n) para variáveis de quadro de vírgula flutuante
A%(n) para variáveis de quadro inteiras
A$(n) para variáveis de quadro em cadeia

O subscrito «n», que deve apresentar-se entre parêntesis do modo acima indicado, pode ser qualquer número inteiro. Podemos portanto imaginar uma variável A(1), A(2) ou até A(300), todas elas diferentes. Em conjunto representam um quadro com o nome geral «A», sendo cada elemento do quadro distinguido pelo subscrito. Se bem que tenhamos usado aqui a letra A como nome do quadro, podem ser escolhidos dois caracteres quaisquer para esse

efeito, obedecendo às mesmas regras que as variáveis simples.

Este princípio pode ser alargado de modo a incluir subscritos duplos ou mesmo triplos sendo cada um deles separado por uma vírgula. Assim, A(r,c) é um exemplo de de um quadro bidimensional. Poderemos imaginar este quadro como contendo um conjunto de números numa matriz XY de *linhas* (subscrito r) e *colunas* (subscrito c), como se mostra em seguida.

23	45	6	78	2
17	3	72	43	8
1	3	48	18	14

Trata-se de uma matriz 5 ×3, estando cada número associado a um par único de subscritos que identificam a sua posição no quadro. Assim, 23 encontra-se na primeira coluna da linha 1, sendo portanto identificado por A(1,1). A(1,4) conterá 78, e A(2,3) conterá 72.

A instrução de dimensão (DIM)

Compreende-se que os grandes quadros multi-dimensionais podem ocupar muita memória. O computador deve de facto ser avisado previamente se qualquer subscrito exceder 10.

A palavra-chave para dimensionar quadros é DIM, e o seu formato é:

DIM A(n)

no caso de quadros de uma dimensão única, onde A é o nome da variável e n é o maior subscrito usado.

Por exemplo: 100 DIM B(25) no início de um programa preparará o computador para reservar vinte e seis posições de memória para o quadro B. A posição a mais deve-se ao facto de a maior parte da BASIC's permitirem o uso do zero como subscrito, pelo que se pode utilizar as

variáveis B(0), B(1), B(3), B(25). É prudente verificar no manual da máquina se esta permite o uso do subscrito zero; este é bastante útil, como se verá adiante.

É possível que durante as fases iniciais do desenvolvimento do programa o leitor não tenha a certeza quanto ao número de subscritos que irá usar para um determinado quadro. Felizmente, o computador não se queixará de sobredimensionamento, pelo que o leitor pode fazer uma estimativa aproximada e acrescentar, digamos, mais dez. Não se esqueça no entanto de que a memória perdida está reservada, mesmo que vazia, pelo que convém não exagerar! Se por outro lado fizer subdimensionamento, esteja preparado para ver no visor uma mensagem de erro.

Tenha cuidado em nunca deixar o computador encontrar uma instrução DIM para o mesmo quadro mais do que uma vez, senão receberá uma mensagem de erro do tipo «REDIMENSION ARRAY ERROR». Se for obrigado a redimensionar, deverá primeiro limpar as variáveis através de uma instrução CLR (clear). Voltaremos a referir-nos a este ponto quando tratarmos dos saltos condicionados.

No caso de quadros multi-dimensionais, a instrução DIM deve incluir todos os subscritos:

DIM S(20,4) reservará um quadro de vinte um por cinco
DIM S(40,10,2), reservará um quadro tri-dimensional

É permitido usar quadros inteiros e quadros de cadeia além dos quadros numéricos usados nos exemplos anteriores. D%(n), F$(15,7), por exemplo, são perfeitamente legais.

INSTRUÇÕES DE IMPRESSÃO

Quando o BASIC foi lançado, o principal dispositivo de saída dos computadores era a impressora designada por «teletipo». Todas as operações, incluindo o enorme

número de correcções e repetições durante o desenvolvimento do programa, deviam ser passadas a papel — quilómetros deste. Muitas florestas canadenses devem ter sido destruídas para satisfazer o mundo dos computadores nestes primeiros tempos... As condições modificaram-se desde então. O principal dispositivo de saída é agora a unidade de «display» visual (VDU), tanto sob a forma de um aparelho especialmente concebido para este fim como sob a forma do normal receptor de televisão. No entanto, os hábitos desaparecem com dificuldade, e por isso é ainda mantida a palavra-chave original, PRINT, se bem que hoje se «imprima» mais no visor do que em papel. São evidentemente ainda necessárias impressoras tradicionais; quando se pretende utilizá-las são necessárias dar algumas ordens adicionais para desviar o output * do visor. Consideraremos que PRINT se refere sempre ao visor, a não ser quando indicado de outro modo.

Podemos usar PRINT dos seguintes modos:

1. Para imprimir o conteúdo de uma variável, por exemplo PRINT A.
2. Para imprimir directamente mensagens literais, por exemplo PRINT «SOMOS OS MELHORES».
3. Para imprimir nada excepto uma linha em branco, por exemplo PRINT.

Como a parte «print» do nosso programa é a única que indica externamente todo o nosso trabalho, vale a pena dar-lhe bastante atenção.

Existem duas marcas de pontuação, o ponto e vírgula e a vírgula, que são extremamente importantes nas instruções PRINT. O seu uso incorrecto pode transformar uma imagem compreensível num autêntico caos.

Uma simples instrução PRINT imprimirá o conteúdo da variável numérica «A» no início de uma nova linha. Se o

---

* Preferencialmente utilizar-se-á o termo «output» ao termo «saída».

número é positivo, o sinal + não é impresso mas é deixado normalmente um espaço para ele. Se o número é negativo, é impresso o sinal. PRINT A$ dá os mesmos resultados, excepto que não é deixado qualquer espaço no início, contrariamente ao que sucede com os números impressos.

PRINT A;B imprimirá o conteúdo de ambas as variáveis numéricas na mesma linha, primeiro A e depois B com um espaço entre ambas, se bem que no caso de números positivos apareçam dois espaços de intervalo devido ao facto de o sinal + ter sido suprimido. É possível imprimir diversas variáveis na mesma linha recorrendo ao separador ponto e vírgula, e o seu número máximo depende da largura do visor. A maior parte dos computadores permitem a impressão de pelo menos 40 caracteres em cada linha. Uma outra utilidade do ponto e vírgula consiste em suprimir o modo habitual de inserção de linhas. Por exemplo:

```
100 PRINT A
110 PRINT B
```

Estas instruções imprimirão A no início de uma linha e B abaixo desta, mas

```
100 PRINT A;
110 PRINT B
```

imprimirá ambas na mesma linha porque o ponto e vírgula suprimiu o habitual retorno de linha que origina nova linha.

A vírgula, tal como o ponto e vírgula, pode também ser usada para imprimir diversas variáveis na mesma linha, mas ao contrário daquele separador a simples vírgula leva a variável seguinte a surgir na próxima *zona* disponível. O visor pode ser considerado dividido num certo número de zonas por linha. Este número dependerá do computador mas em princípio haverá quatro zonas, cada uma delas com dez posições. Se uma variável se

estende para além da sua zona, a variável seguinte só surgirá na zona livre seguinte. Assim:

```
100 PRINT A,B,C
```

Produzirá a seguinte saída:

```
xxx        xxxx        xxx
```

enquanto que

```
100 PRINT A;B;C
```

imprimirá o seguinte:

```
xxx xxxx xxx
```

(o x representa em todos os casos um dígito numérico).

PRINT «POR FAVOR INDICAR NOME» é chamado um print *literal*. Podem-se usar caracteres dado que não existem regras sintáticas quando os caracteres se encontram encerrados entre aspas. Existe no entanto uma excepção: a maior parte dos BASIC's não permitem o emprego de aspas dentro de aspas. Assim PRINT «O SEU NOME É «DEREK»?» pode ser ilegal. Uma outra função importante na instrução PRINT é SPC(n), que leva o valor seguinte a ser impresso a n espaços do anterior.

Por vezes, a fim de espaçar o texto verticalmente, é necessário «imprimir» linhas em branco. A simples instrução PRINT, imprime uma linha em branco. Assim:

```
100 PRINT: PRINT:PRINT
```

imprimirá três linhas em branco.

Técnicas de impressão

É provavelmente banal definir «informação» como «aquilo que informa», mas é conveniente compreender que as palavras ou frases enviadas para o visor podem

não ser muito fortes em «poder» de informação do ponto de vista do observador humano. A *posição* de uma palavra relativamente ao resto do visor pode ser tão importante para a apresentação de informações como a palavra propriamente dita. O número de caracteres por linha e o número de linhas por «página» do visor não é excessivamente generoso na maior parte dos computadores pequenos, e portanto todas as palavras, espaços e linhas devem ser bem aproveitados. Isto não significa que se deva encher o visor com palavras e números. De facto, uma produtividade excessiva em palavras pode conduzir a uma subprodutividade em poder de informação. Não caia no erro, tão vulgar, de dar grande atenção ao funcionamento interno do programa e considerar a apresentação de «respostas» como trivial. Pratique bastante a arte de escolher as palavras mais informativas, tendo em conta a necessidade de economia. Por outro lado, não se exceda na procura de uma economia nem utilize a linguagem incompreensível típica da gíria usada nos primeiros computadores ou nos filmes americanos de gangsters...

Temos de admitir que o BASIC não é muito forte no que se refere ao posicionamento do texto. Se bem que seja possível apresentar palavras nas posições exactas que desejamos, é um pouco difícil fazê-lo até estarmos habituados ao uso dos separadores (ponto e vírgula e vírgula). A função TAB é uma outra arma disponível para o posicionamento do texto. TAB(n) antes de uma variável a imprimir assegurará que a impressão se inicie na posição n da linha... se possível. Por exemplo:

```
100 PRINT TAB(9) A
```

imprimirá A na nona coluna de uma linha. Por vezes, a primeira posição de impressão é TAB(0), pelo que você deslocará todas as palavras de uma coluna se não se lembrar disso (veja o manual). Existe normalmente um limite para o número máximo de TAB, por vezes 255. Assim, TAB(240) é equivalente, num visor de 40 caracteres por linha, a começar a impressão depois de deixar seis linhas

em branco. Vejamos mais alguns exemplos da função TAB:

```
100 PRINT A TAB(20) B TAB(32) C
```

A será impresso imediatamente (*justificada* à esquerda, como se diz em gíria; justificar é sinónimo de «alinhar»); B será impresso na coluna 20, e C na coluna 32.

```
100 PRINT TAB(12) «TABELA DE RAÍZES
    QUADRADAS»
```

Esta instrução imprimirá o texto a partir da coluna 12. Em vez de definir o número «tab» directamente, é legal o uso de uma variável ou expressão previamente definida. Assim:

```
100 PRINT TAB(D + 2*K)A
```

pode ser usado.

Abreviatura de PRINT. Como a palavra-chave PRINT é muito usada na maior parte dos programas, certos BASIC's permitem-nos abreviá-la usando o ponto de interrogação «?» no teclado a fim de evitar a escrita completa da palavra. Numa listagem, no entanto, continua a surgir a palavra PRINT mesmo que se tenha usado o ponto de interrogação.

OUTRAS INSTRUÇÕES

A instrução INPUT

Os programas necessitam de dados. Não se pode pedir a um computador para determinar, por exemplo, uma raiz quadrada a menos que se lhe forneça um número, ao qual chamamos dado. Até aqui, o único método nosso conhecido de introduzir dados num programa consiste em usar uma instrução no interior do próprio pro-

grama. O problema deste método é que se quisermos usar o mesmo programa com dados diferentes nos veremos obrigados a alterar uma instrução de atribuição. O programa não terá então muito valor prático. Felizmente, dispomos de vários métodos alternativos de fornecer os dados à máquina, consistindo um deles no recurso à instrução INPUT. Por exemplo:

```
100 INPUT A
```

Quando o computador encontra esta linha ao executar um programa, pára e espera até que se lhe introduza um dado e se carregue na tecla RETURN. Este número será colocado na variável A (ou em qualquer outro nome de variável numérica que desejemos). Dispomos assim de um modo de «conversar» com o computador durante a execução de um programa, o que nos permite alterar os dados sem modificar o programa.

Uma outra forma da instrução de input permite-nos acrescentar uma mensagem, que nos surge depois como um pedido:

```
100 INPUT «POR FAVOR INDIQUE O SEU NÚMERO
DE TELEFONE»; T
```

Esta mensagem será apresentada no visor quando o computador pára, e é preferível à anterior forma de INPUT. Note a inclusão do ponto e vírgula entre o fecho das aspas e a variável T.

Pode-se dar input a dados de vírgula flutuante, de números inteiros ou de cadeia de caracteres desde que se utilize o nome correcto da variável:

```
100 INPUT «QUAL É O SEU NOME?»;N$
```

Em particular, tenha cuidado em não provocar um erro de adaptação na instrução INPUT. Se introduzir uma letra (ou qualquer outro caracter não numérico) numa variável numérica o resultado é necessariamente uma mensagem de erro.

Um dos aspectos aborrecidos da instrução INPUT na maior parte dos BASIC's é a passagem ao nível das ordens quando a tecla RETURN é inadvertidamente premida antes de se dar entrada aos dados. Podem-se usar vários truques para impedir isto, mas não seria conveniente discutí-los neste momento.

A instrução GET

Esta instrução é semelhante a INPUT mas só permite a recepção de um caracter e na maior parte dos BASIC's o computador não esperará que você dê entrada de dados. Se já o tiver comunicado, muito bem; mas a máquina não ficará à espera dele. Torna-se aqui necessário explicar um pouco melhor o método usado pelos computadores para tratamento do input por teclado.

Quando se carrega numa tecla, a versão codificada da palavra-chave é colocada num registo «buffer». Este registo pode conter vários caracteres, que aguardam processamento pelo computador (normalmente contêm 10 caracteres). O registo está organizado de modo a aceitar os dados por ordem, isto é, o primeiro a entrar é o primeiro a sair... Voltando à instrução GET A, o computador limita-se a «agarrar» no primeiro caracter que estiver no registo buffer, colocando em A ou A$, conforme o caso. Como se disse anteriormente, a acção GET de algumas versões de BASIC espera que o operador dê entrada a um caracter (compontando-se então como uma instrução INPUT de um só caracter); outras não.

A instrução GOTO

No início do capítulo, foi afirmado que os computadores executam cada instrução pela ordem dos números de linha, a menos que recebam outras ordens. A instrução GOTO é uma destas últimas ordens. Por exemplo:

```
100 GOTO 535
```

Ao encontrar esta instrução, o computador passará simplesmente para a linha 535, considerando-a como a instrução seguinte e desprezando todas as linhas intermédias. Examinemos o seguinte programa:

```
100 PRINT «ESTE PROGRAMA É INFINITO»
110 GOTO 100
```

Trata-se certamente de um programa «infinito», porque a mensagem será impressa continuamente, enchendo o visor. A única maneira de sair deste programa consistirá em carregar na tecla STOP. Vejamos um exemplo ainda mais inútil:

```
100 GOTO 100
```

A instrução GOTO é considerada um *salto incondicional.* Mais adiante encontraremos métodos menos violentos de alterar o processamento normal, condicionando o salto.

Ainda como exemplo, e para finalizar o capítulo, estudemos o seguinte:

```
100 A = 1
110 PRINT A
120 A = A + 1
130 GOTO 110
```

IF (Se) conseguir descobrir o que este programa fará THEN (então) pode GOTO (ir para) o capítulo seguinte...

RESUMO

Qualquer programa é um conjunto de instruções, sendo atribuído a cada uma delas um número de linha arbitrário.

Os programas são comunicados à máquina linha a linha, servindo a tecla RETURN para indicar o final da linha.

Se se utiliza mais do que uma instrução por linha, todas se apresentam separadas por dois pontos.

As *ordens* não possuem número de linha e são executadas imediatamente depois de se carregar na tecla RETURN.

A ordem LIST apresentará o programa no visor, ou parte dele.

A ordem RUN executará o programa, ou parte dele.

A ordem NEW apagará todas as linhas do programa, deixando o computador «limpo».

A máquina está em modo «command» (recebe ordens) quando o cursor pisca ou é apresentado no visor alguma mensagem como READY.

As instruções REM são ignoradas pelo computador, utilizando-se apenas para introduzir observações fáceis de compreender pelo operador.

Os espaços no interior das instruções são opcionais, mas não devem encontrar-se entre palavras-chave.

As variáveis podem ser designadas por um ou dois caracteres, sendo o primeiro uma letra; os caracteres que se sigam a estes dois não são distinguidos pela máquina.

As variáveis são consideradas como números de vírgula flutuante a menos que sejam seguidas de % (números inteiros) ou $ (cadeia de caracteres).

Uma cadeia de caracteres é uma colecção de caracteres encerrados entre aspas.

As cadeias de caracteres só podem ser guardadas em variáveis de cadeia, e as instruções de atribuição não devem misturar cadeias e não-cadeias de ambos os lados do sinal de igual.

As variáveis de cadeia podem conter números, mas não é possível executar com elas as operações aritméticas normais.

As variáveis de cadeia podem ser «concatenadas» recorrendo ao sinal +.

Os números grandes ou pequenos são representados em notação científica usando mEp, onde m é o número (mantissa) e p a potência de dez a que deve ser elevado.

Existem certas funções especiais que suplementam as operações aritméticas simples, usando a forma xxx(Y), onde xxx é o nome da função e Y a quantidade (argumento) que deve ser processado pela máquina.

As *variáveis subscritas* permitem o uso de quadros nomeados, onde cada elemento é distinguido por um subscrito. Este pode na maior parte dos casos incluir o 0.

Os quadros podem ser multi-dimensionais, considerados como «linhas», «colunas», etc.

A instrução de *dimensão* (DIM) torna-se necessária no início do programa em todos os casos em que se utilizem subscritos superiores a 10.

As instruções de dimensão não devem ser novamente executadas a menos que sejam precedidas de CLR (contracção de CLEAR).

As instruções de impressão referem-se à saída por visor, a menos que se especifique outra coisa.

O ponto e vírgula é usado para separar variáveis, deixando um espaço entre elas (e um outro para o sinal positivo ou negativo do número). O ponto e vírgula é também usado para impedir a passagem para a linha seguinte.

Usam-se vírgulas simples para separar diferentes zonas de impressão.

A função TAB(n) é usada para definir a coluna exacta onde deve ser impressa a «resposta».

As saídas obtidas no visor são sempre muito importantes, nunca devendo ser apresentadas de modo pouco claro ou pouco informativo.

A instrução INPUT permite a entrada de dados durante a execução do programa.

GET é semelhante a INPUT mas apenas permite a introdução de um caracter e normalmente não fica à espera deste.

A instrução GOTO é usada para realizar saltos incondicionais.

3

DESENVOLVIMENTO DE UM PROGRAMA

FLUXOGRAMAS

Já conhecemos um número suficiente de palavras-chave BASIC para podermos construir alguns programas simples. Mesmo estes exigem algum raciocínio e cuidado, sendo bastante normal só se conseguir o resultado pretendido após a realização de alguns exercícios de «polimento», pois as primeiras tentativas terminam normalmente no cesto dos papéis.

Tradicionalmente, o desenvolvimento de um programa começava pela construção de um *fluxograma*, um tipo de diagrama que nos indicava a «estratégia» geral de execução do programa. Actualmente as opiniões apresentam-se bastante divididas quanto ao seu valor; algumas autoridades eminentes continuam a afirmar a sua utilidade, mas outras igualmente eminentes consideram-no inútil. Alguns chegam a afirmar (ou pelo menos sugerir) que quando um programa necessita de um fluxograma explicativo isso se deve ao facto de a sua linguagem ser insatisfatória. Provavelmente a verdade encontra-se entre estas duas posições. O fluxograma para a realização de simples operações aritméticas é apresentado na figura 3.1.

As acções simples são representadas em rectângulos, com excepção das paragens (STOP) que são indicadas em rectângulos com cantos curvos. Existem muitos outros símbolos normalizados, mas estes bastarão por agora. As palavras no interior das caixas devem ser escassas e humanizadas. Pretende-se indicar a «estratégia» a que um pro-

grama obedece, sendo a «táctica» representada pelo programa propriamente dito. Um excesso de pormenores ou de frases recorrendo a termos de computador torna o fluxograma redundante porque se assemelhará ao programa em vez de constituir o seu prelúdio.

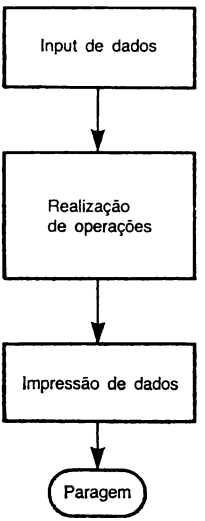

Fig. 3.1 — Um fluxograma simples.

No nosso primeiro exemplo de programa, o objectivo consiste em dar input do raio de um círculo e levar o computador a dizer-nos o seu perímetro e a sua área. Teremos de usar as fórmulas:

$$C = 2\pi r$$

e

$$A = \pi r^2$$

onde C é o perímetro, r o raio, A a área e π é igual a 3,141592654. Designaremos as variáveis por C (perímetro), R (raio), a (área) e P (valor de π). Notamos o enjoo do

leitor devido à simplicidade desta tarefa, mas ignorámo-lo porque iremos discutir apenas *princípios*, não nos preocupando com pormenores como a descoberta da área de um dodecaedro (que de resto não sabemos o que é). Entre uma coisa e outra apenas diferirão os pormenores matemáticos. Façamos a nossa primeira tentativa:

Programa 1

```
100 INPUT R
110 P = 3.141592654
120 C = 2*P*R
130 A = P*R ↑ 2
140 PRINT C
150 PRINT A
160 STOP
```

Consultando o fluxograma, notamos que a linha 100 corresponde ao «input de dados» (neste caso o raio). As linhas 110 a 130 «realizam as operações» matemáticas e as linhas 140 e 150 «imprimem os resultados» (respectivamente o perímetro e a área). No que se refere às equações originais, partimos do princípio de que já são conhecidas pelo leitor. No entanto, não é necessário que o programador compreenda as equações usadas; pode sempre fazê-lo através de um amigo ou colega, ou até consultando um livro.

Que poderemos dizer sobre a forma como o programa está escrito? Não muito, além de admitir que nos dá as respostas desejadas. É frio, inflexível, pouco amistoso e não tem certamente uma aparência profissional. Pode-se afirmar que um programa com um objectivo tão trivial merece um tratamento igualmente trivial. Esta atitude não é porém correcta, e conduz a maus hábitos que depois se tornam difíceis de eliminar quando enfrentamos um objectivo mais complicado.

Além da frieza da apresentação no visor, o programa não dispõe de quaisquer esquemas de rejeição de inputs «idiotas» feitas pelo operador. O termo «idiota» tem aqui

o seu significado menos antipático porque todos nós, apesar de brilhantes em circunstâncias normais, entramos por vezes em crise quando operamos um programa. Carregamos nas teclas incorrectas, e em momentos de extremo sadismo chegamos até a tentar afligir deliberadamente o pobre computador. Por isso, convém que os programas incluam algumas *técnicas de validação* de input. O programa do computador deve verificar todas as entradas durante a sua própria execução, rejeitando-as delicadamente se forem ridículas ou, pelo menos, enviando-nos uma mensagem interrogativa quando a grandeza de um número está um pouco além ou aquém do «normal». Para introduzir esta validação das entradas, e por outras razões, convém apresentar ao leitor uma nova instrução, o *salto condicional* do BASIC. Depois de estudar esta «poderosa» instrução, poderemos voltar ao programa 1 e dar-lhe alguma humanidade.

A INSTRUÇÃO IF... THEN (Se... Então).

Referimo-nos já à instrução GOTO, que produz um salto incondicional dentro do programa. Mas o verdadeiro poder de um computador reside nos seus processos de decisão. As decisões são feitas em BASIC usando a instrução IF... THEN. O seu formato é o seguinte:

IF condição THEN acção

Exemplo:

```
100 IF A = 23 THEN PRINT «GANHOU O PRIMEIRO
    PRÉMIO»
110 (qualquer instrução)
```

Na linha 100, o computador examina a variável A, verificando se a condição é verdadeira. Se o for, realiza a acção indicada depois de THEN. Se não for verdadeira, não realiza a acção e passa à linha seguinte, neste caso 110.

Em geral, algo é feito ou não, conforme a condição é verdadeira ou falsa. No exemplo dado, a condição basea-

va-se no sinal de igual. Mas existem outros «operadores condicionais», que apresentamos em seguida:

OPERADORES CONDICIONAIS

=	igual a
<	menor do que
>	maior do que
>=	maior ou igual a
<=	menor ou igual a
<>	diferente de

A «acção» indicada depois de THEN pode ser qualquer das instruções BASIC. Se no entanto a acção for GOTO, alguns BASIC's permitem-nos evitar o emprego desta palavra. Por exemplo:

```
100 IF A = 23 THEN 500
```

O resultado é o mesmo que se teria se estivesse escrito: THEN GOTO 500. A forma a utilizar depende das indicações contidas no manual da máquina.

Se existirem várias instruções depois de THEN (usando o mesmo número de linha), todas elas serão obedecidas se a condição for verdadeira, e nenhuma o será se for falsa. Por exemplo:

```
100 IF A = 23 THEN B = 20: PRINT «ERRADO»:
    GOTO 456
```

Se a condição for verdadeira, é atribuído o valor 20 à variável B, é impressa a palavra «ERRADO» e a próxima instrução é executada na linha 456; se for falsa, o computador ignora as três instruções e passa para o número de linha seguinte. Tanto a condição como a acção podem ser expressões complexas, como se mostra nos exemplos seguintes:

```
IF A> 2*B/T + (F/K) THEN R = 245
IF B> 13 THEN B = G + K* SQR(TAN(X)): GOTO
2500
```

```
IF B$ <> «LONDRES» THEN PRINT
«INCORRECTO»: GOTO 300
```

As condições AND (e) e OR (ou)

Na vida real, são permitidas certas coisas desde que estejam satisfeitas duas ou mais condições. Podemos por exemplo beber um copo num bar SE tivermos dinheiro E o bar estiver aberto. Por outro lado, certas coisas só são permitidas se uma OU mais condições estiverem satisfeitas, por exemplo ter dinheiro OU um livro de cheques. Pode também haver uma mistura de condições E (AND) e OU (OR). Pode ser necessário dinheiro OU um livro de cheques E um cartão de crédito.

O BASIC inclui as palavras-chaves AND e OR que podem ser usadas na parte condicional de uma instrução IF... THEN. Por exemplo:

```
100 IF A = B AND C = D THEN PRINT «SÃO DO
    MESMO TEMPO»
```

É necessário que ambas as condições sejam satisfeitas para que a instrução PRINT seja executada

```
100 IF A > 3 AND A < 10 THEN 340
```

Este último exemplo é típico do tipo de «trap» (armadilha) usada para validar inputs idiotas. Considera-se que estamos a verificar números maiores do que 3 mas menores do que 10. Deve-se ter bastante cuidado ao usar AND deste modo. É fácil cair no erro de usar OR em vez de AND. Para ilustrar isto, escrevemos novamente a mesma linha usando OR:

```
100 IF A > 3 OR A < 10 THEN 340
```

O erro é que numa relação OR será apenas necessário que uma das condições seja satisfeita, pelo que o segundo

membro permitirá o uso de todos os números inferiores a 10, incluindo os inferiores a 3, sendo a primeira condição claramente redundante!

São permitidas todas as combinações de AND e OR na condição; por outro lado as variáveis podem ser diferentes, como acontece no caso seguinte:

```
100 IF A > B AND F = K OR V1 < F*F + 6 THEN 600
```

É necessário que sejam verdadeiras as duas primeiras condições ou a terceira condição por si só é suficiente para activar a parte THEN 600.

É fácil errar a sintaxe das instruções AND e OR. Vejamos um exemplo que provocará um erro de sintaxe:

```
100 IF A = 45 OR 68 THEN 500 (erro de sintaxe)
```

Se bem que a frase nos pareça correcta, o BASIC obriga a mencionar a variável (neste exemplo, A) sempre que surge na condição OR ou AND. Assim, a linha anterior deveria ser escrita do seguinte modo:

```
100 IF A = 45 OR A = 68 THEN 500
```

Vejamos agora um exemplo de como se deve construir uma armadilha para inputs idiotas. Perguntaremos a idade de alguém com uma instrução de INPUT e rejeitaremos as entradas que forem consideradas impossíveis. O primeiro obstáculo consistirá em definir o que é aqui «impossível». O limite inferior pode ser fixado por exemplo em três, porque é razoável admitir que ninguém com menos de três anos pode operar um computador. Na outra extremidade, podemos considerar um limite de 120, idade que de acordo com um documentário televisivo não é impossível na Ucrânia. Consideramos portanto que mesmo no caso de alguém com mais de 120 anos poder sentar-se à frente de um computador, é improvável que ainda lhe reste força suficiente para carregar na tecla RETURN...

Tendo sido estabelecidos estes limites, poderemos escrever um segmento do programa com a forma seguinte:

```
100 INPUT «POR FAVOR INDIQUE A SUA IDADE»; A
110 IF A < 3 OR A > 120 THEN 100
```

Uma resposta que não se encontre dentro destes limites levará à repetição da mensagem de INPUT, que continuará no visor até ser correctamente respondida. Se bem que tenhamos conseguido incluir uma ratoeira no programa, esta não é talvez a mais apropriada; o computador parecerá pouco amistoso, talvez até um pouco sobranceiro na sua atitude. Não nos dá qualquer explicação sobre a sua recusa, limitando-se a repetir friamente a pergunta. Para humanizar esta resposta da máquina, devemos explicar a razão da recusa, mas espero que o leitor evite expressões do género «este dado não serve». O próprio computador sentir-se-á certamente embaraçado com esta resposta! Será mais razoável usar o seguinte:

```
100 INPUT «POR FAVOR INDIQUE A SUA IDADE»; A
110 IF A < 3 OR A > 120 THEN PRINT «IDADE
    INVULGAR. POR FAVOR INDIQUE-A DE NOVO»:
    GOTO 100
```

Este tipo de procedimento deve ser sempre empregue; isto é, a máquina pede o dado, verifica-o, e se não for aceitável explica porquê e pede outro dado.

Armados com a técnica IF... THEN podemos voltar ao programa 1 e tentar fazer algumas modificações. Além de alterarmos o sistema de input, podemos também melhorar a apresentação do visor, sendo ainda possível reutilizar o programa com diferentes raios sem obrigar o programa a ser interrompido de cada vez. O fluxograma deverá ser revisto para incorporar todos estes melhoramentos, como se vê na figura 3.2.

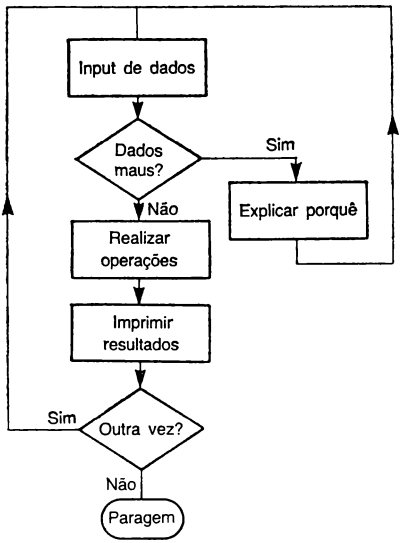

Fig. 3.2. — Fluxograma melhorado.

Note-se que surge agora um novo símbolo no fluxograma: o *losango de decisão*, que representa de modo adequado a instrução IF... THEN. O losango inferior permite uma nova execução se o operador o desejar. Deve-se sublinhar que o fluxograma nunca deve conter demasiados pormenores, limitando-se a sugerir a estratégia do programa. Os detalhes, quando necessários podem ser explicados em instruções REM no interior do próprio programa.

A partir do fluxograma podemos agora construir a segunda versão do Programa 1.

Programa 1A

```
100 CLS:P = 3.141592654:REM*LIMPAR VISOR E
    DEFINIR PI*
```

```
110  PRINT TAB(10) «O CÍRCULO»: REM*TÍTULO*
120  PRINT:PRINT:PRINT:REM*3 LINHAS EM
     BRANCO*
130  INPUT «POR FAVOR INDIQUE RAIO»; R
140  IF R < O OR R > 1E20 THEN PRINT «FORA
     DOS LIMITES»: GOTO 130
150  C = 2*P*R:A = P*R ↑ 2:REM*OPERAÇÕES*
160  PRINT:PRINT:PRINT:REM*3 LINHAS EM
     BRANCO*
170  PRINT «O PERÍMETRO É «C» UNIDADES»
180  PRINT
190  PRINT «A ÁREA É «A» UNIDADES AO
     QUADRADO»
200  PRINT:PRINT:PRINT
210  INPUT «DESEJA NOVA EXECUÇÃO?
     RESPONDA (YES) OU (NO)»; K$
220  IF K$ = «YES» THEN 100
230  IF K$ = «NO» THEN STOP
240  PRINT «POR FAVOR RESPONDA
     CORRECTAMENTE»: GOTO 210
```

A linha 100 exige algumas explicações. A palavra-chave CLS significa «limpar visor» mas não constitui um código normalizado em BASIC porque na época da versão Dartmouth não existiam visores; todas as saídas eram impressas. Consequentemente, o método de limpeza do visor pode não corresponder ao da sua máquina; convir-lhe-á consultar o manual desta a fim de descobrir o método conveniente. Entretanto continuaremos a usar CLS. Na mesma linha, pi é primeiramente atribuído à variável P. É preferível definir as constantes apenas uma vez no início do programa, porque nestas condições aumenta a velocidade de execução do programa.

A linha 110 imprime o título do programa na parte superior do visor, e usando-se TAB (10) consegue-se colocá-lo perto do centro do mesmo, se bem que a posição exacta varie evidentemente com a largura do visor. É fácil executar o programa uma vez e em seguida alterar o número TAB no caso de o título não ficar ao centro.

Depois do título existem três linhas em branco que servem para deixar algum espaço antes da impressão seguinte porque o aspecto do visor é muito desagradável quando todas as impressões se encontram juntas umas às outras.

As linhas 130 e 140 tratam do input e da sua validação. Note que os raios negativos (números inferiores a zero) são rejeitados, juntamente com os raios superiores a 10^{20}, o que deve permitir o cálculo mesmo no caso de corpos astronómicos. Esta escolha foi realizada de modo arbitrário. As operações matemáticas são realizadas na linha 150, ocorrendo depois outras 3 linhas em branco antes da apresentação final das «respostas». Examine cuidadosamente a linha 170 porque mostra como se pode apresentar o texto de uma forma humanizada. Assim, o texto literal «O PERÍMETRO É» é impresso primeiramente (note o espaço antes das aspas finais, usado para efeitos de separação), e em seguida imprime-se o conteúdo da variável C e um novo texto literal: «UNIDADES». A segunda resposta é apresentada de modo semelhante, mas três linhas mais abaixo.

A linha 210 pergunta ao operador se deseja realizar novos cálculos, indo a resposta para K$. Se «yes», todo o programa é passado novamente a partir da linha 100. Se «no», o programa passa ao modo de comando e pára. Se o operador não responder «sim» nem «não», esta atitude é considerada ilegal (talvez até mal-educada) e a pergunta é feita novamente pela linha 240.

```
                O CÍRCULO

POR FAVOR INDIQUE O RAIO 4
O PERÍMETRO É 25.1327412 UNIDADES
A ÁREA É 50.2654825 UNIDADES AO QUADRADO
DESEJA NOVA EXECUÇÃO? RESPONDA (YES)
    OU (NO)
```

Considerou-se que foi indicado o valor 4 para o raio do círculo, mas tudo o resto foi realizado pelo computador. Talvez exista um erro de arredondamento no nono dígito das respostas, mas quem se preocupa com isso?

O leitor concordará certamente que o Programa 1A é bastante melhor do que o Programa 1. Constitui no entanto um bom exemplo da quantidade relativamente grande de memória que é necessária para humanizar um programa. E mesmo nesta sua última forma o programa pode ainda ser melhorado; por exemplo, o número de dígitos nas respostas será normalmente superior ao necessário, beneficiando portanto se incluirmos algum tipo de arredondamento.

Arredondamento para N casas decimais

Estude cuidadosamente a linha seguinte, porque constitui um método vulgar de arredondar qualquer número (em formato de vírgula flutuante) para duas casas decimais:

```
100 A = INT (100*A + 0.5)/100
```

Misteriosa? Talvez, mas para facilitar considere que «A» continha originalmente, por exemplo, 32,568, e substitua este valor no lado direito da afirmação. Então:

```
A = INT (100*32.568 + 0.5)/100
A = INT (3256.8 + 0.5)/100
A = INT (3257.3)/100
A = 3257/100 = 32.57
```

que constitui o arredondamento correcto para duas casas decimais.

Podemos agora generalizar este método de modo a poder ser usado para qualquer número de casas decimais, modificando o 100 que nos aparece duas vezes. Se este valor fosse 1000, em vez de 100, arredondaria para três

casas decimais. Como 100 é 10 ↑ 2 e 1000 é 10 ↑ 3, é fácil conceber uma solução geral que permita ao operador decidir o número de casas decimais de cada vez que executa o programa. Suponhamos que se incluía a linha seguinte no programa 1A:

```
134 INPUT «QUANTAS CASAS DECIMAIS»; D
```

Antes das respostas serem impressas, a linha seguinte produziria agora um arredondamento para «D» casas decimais se fosse incluída um pouco mais adiante:

```
155  C = INT (10 ↑ D*C + 0.5)/10 ↑ D : A = INT
     (10 ↑ D*A + 0.5)/10 ↑ D
```

Esta última linha tem um aspecto aterrador, mas quando cuidadosamente analisada mostrará ser razoavelmente inofensiva. Note que as operações matemáticas para determinação do perímetro e da área continuam a ser calculadas com todo o rigor possível na linha 150, mas que a linha 155 converterá os números em C e A para D casas decimais antes de imprimir os resultados. O programa é já bastante aceitável, em termos das instruções BASIC que já conhecemos. É este o momento de estudarmos uma outra instrução importante.

A INSTRUÇÃO ON... GOTO

Muitos programas escritos equivalem de facto a vários num só, ligados entre si. Os vários sub-programas possuem normalmente um tema qualquer em comum. Por exemplo, o programa que acabámos de desenvolver trata do cálculo matemático de um círculo. A escala do programa poderia ser aumentada de modo a que o cálculo do círculo fosse apenas uma das suas capacidades, podendo também fazer cálculos de triângulos, cones, cubos, esferas, etc. Podia ser incluída numa página de «sumário» ligando os programas entre si, e atribuindo-se a cada capacidade

um número opcional. O operador seria então convidado a indicar a opção desejada, e o computador apresentaria a parte do programa correspondente. Depois de terminada a área escolhida, o operador teria a possibilidade de voltar à página de sumário.

A instrução BASIC que corresponde ao interruptor de selecção é a ON GOTO:

```
100 ON A GOTO 1000, 2000, 3500, 5000, 6500
```

O seu funcionamento é o seguinte: o computador examina A. Se A = 1, o programa passa para o primeiro número de linha (1000). Se A = 2, passa para o segundo número de linha (2000), e se A = 5, passa para o quinto número de linha (6500).

Vejamos a forma de usar esta instrução, considerando que já foi apresentada ao operador a página de sumário:

```
100 INPUT «INDICAR O NÚMERO DA OPÇÃO
    DESEJADA»; A
110 ON A GOTO 200, 300, 400
200 (primeira opção)
300 (segunda opção)
400 (terceira opção)
```

Apresentamos aqui apenas um esboço de estrutura do programa. Cada opção será evidentemente programada no interior do respectivo bloco de números de linha.

Note o formato da ON GOTO, particularmente as vírgulas depois dos números, excepto no último. É importante que a variável (neste caso «A») nunca seja um número superior ao total de números ON GOTO. Assim, no exemplo dado, se A for 4 o computador apresentará uma mensagem de erro sob a forma «instrução indefinida» ou algo semelhante.

Quando se escolhe o conjunto de números de destino da instrução ON GOTO, deve-se ter presente que ainda não se escreveram os diversos sub-programas e que

se tem apenas uma ideia do número de linhas necessário para cada. Devido a este facto convém deixar bastante espaço «numérico» entre os números de destino. Por exemplo:

```
100 ON A GOTO 200, 210, 220
```

seria demasiado escasso, deixando apenas dez números de linha para cada sub-programa. É melhor ser generoso, como no exemplo seguinte:

```
100 ON A GOTO 1000, 2000, 3000
```

Não tenha medo de desperdiçar números de linhas; não está a desperdiçar memória porque não existe qualquer ligação entre os números de linha e a memória. Existe um valor limite para a numeração de linhas, mas este encontra-se normalmente na região dos 50 000 ou mais.

Mudando um pouco de assunto, poderíamos referir aqui que depois de um programa estar terminado (se é que algum programa se pode dar como terminado), as várias correcções, apagamentos de linhas e acrescentos, que são inevitáveis durante o seu desenvolvimento, terão deixado uma numeração bastante irregular. Muitos sistemas actuais possuem uma ordem RENUMBER que nos permite decidir qual o número da primeira linha e qual o incremento a usar para as linhas seguintes. Esta ordem ordenará, de um modo regular e crescente, o programa e as instruções GOTO e IF THEN de modo apropriado. Assim, um programa que possua os números de linha 80, 82, 100, 103, 246 pode ser numerado 100, 110, 120, 130, 140, o que apesar de não ser melhor do ponto de vista técnico é-o pelo menos do ponto de vista estético.

Voltando à instrução ON GOTO, a figura 3.3 mostra-nos a estrutura geral de um programa que incorpora uma página de opção, ou «menú». Este esquema é de fácil compreensão, e constituirá o guia «estratégico» da vossa versão seguinte, o Programa 1B.

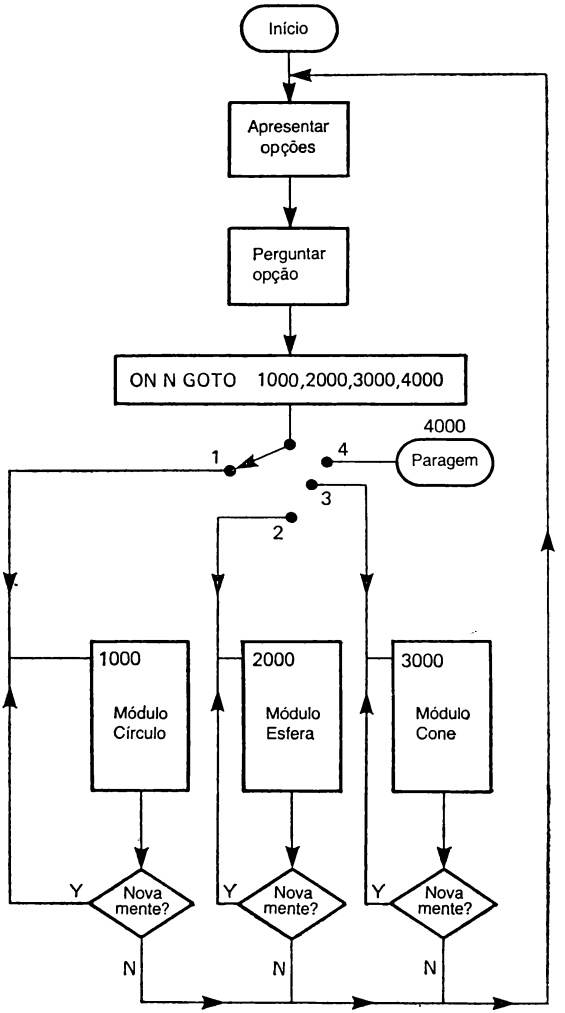

Fig. 3.3. — Estrutura de um programa de «opção».

Programa 1B

```
100 CLS:P = 3.141592654
110 PRINT TAB(10) «OPÇÕES» :PRINT:PRINT:
    PRINT
120 PRINT«CÍRCULO                    1»:PRINT
130 PRINT«ESFERA                     2»:PRINT
140 PRINT«CONE                       3»:PRINT
150 PRINT«SAÍDA DE PROGRAMA 4»:PRINT:PRINT
160 INPUT«INDICAR OPÇÃO DESEJADA»; N
170 IF N < O OR N > 4 THEN PRINT«FORA DE
    LIMITE»: GOTO 160
180 ON N GOTO 1000, 2000, 3000, 4000
999 REM: *CÍRCULO*
1000 CLS
1010 INPUT«INDICAR RAIO»;R
1020 IF R < O OR R > 1E20 THEN PRINT«FORA
    DOS LIMITES»: GOTO 1010
1030 C = 2*P*R:A = R*P*R
1040 INPUT«QUANTAS CASAS DECIMAIS?»;D:
    PRINT:PRINT
1050 IF D < O THEN PRINT «VALORES
    NEGATIVOS NÃO ACEITES»: GOTO 1040
1060 C = INT(10 ↑ D*C + 0.5)/10 ↑ D:
    A = INT(10 ↑ D*A + 0.5)/10 ↑ D
1070 PRINT «O PERÍMETRO É «C» UNIDADES»:
    PRINT:PRINT
1080 PRINT «A ÁREA É «A» UNIDADES AO
    QUADRADO»:PRINT:PRINT
1090 INPUT«NOVAMENTE? RESPONDA (YES) OU
    (NO)»;K$
1100 IF K$ = «YES»THEN 1000
1110 IF K$ = «NO»THEN 100
1120 PRINT«RESPOSTA INACEITÁVEL»:GOTO 1090
1999 REM *ESFERA*
2000 CLS
2010 INPUT«INDICAR RAIO»;R
```

```
2020 IF R < O OR R > 1E20 THEN PRINT«FORA
     DOS LIMITES»: GOTO 2010
2030 A = 4*P*R*R:V = 4*P*P ↑ 3/3: REM*AREA
     SUPERFICIAL E VOLUME*
2040 INPUT«QUANTAS CASAS DECIMAIS?»;D:
     PRINT:PRINT
2050 IF D < O THEN PRINT«VALORES
     NEGATIVOS NÃO ACEITES»: GOTO 2040
2060 A = INT(10 ↑ D*A + 0.5)/10 ↑ D:
     V = INT(10 ↑ D*V + 0.5)/10 ↑ D
2070 PRINT«A ÁREA SUPERFICIAL É «A»
     UNIDADES AO QUADRO»:PRINT:PRINT
2080 PRINT«O VOLUME É «V» UNIDADES
     CÚBICAS»: PRINT:PRINT
2090 INPUT«NOVAMENTE? RESPONDA (YES) OU
     (NO)»;K$
2100 IF K$ = «YES» THEN 2000
2110 IF K$ = «NO» THEN 100
2120 PRINT«RESPOSTA INACEITÁVEL»:GOTO 2090
2999 REM*CONE RECTO*
3000 CLS
3010 INPUT«INDICAR RAIO DA BASE»;R:PRINT
3020 IF R < O OR R > 1E20 THEN PRINT«FORA
     DOS LIMITES»: GOTO 3010
3030 INPUT«INDICAR ALTURA VERTICAL»;H:
     PRINT:PRINT
3040 IF H < O OR H > 1E20 THEN PRINT «FORA
     DOS LIMITES» GOTO 3030
3050 V = P*R*R*H/3:REM*VOLUME*
3060 INPUT «QUANTAS CASAS DECIMAIS?»; D
3070 IF D < O THEN PRINT«VALORES
     NEGATIVOS NÃO ACEITES»: GOTO 3060
3080 V = INT(10 ↑ D*V + 0.5)/10 ↑ D
3090 PRINT«O VOLUME DO CONE É «V»
     UNIDADES CÚBICAS»:PRINT:PRINT
3100 INPUT«NOVAMENTE? RESPONDA (YES) OU
     (NO);K$
3110 IF K$ = «YES» THEN 3000
```

```
3120 IF K$ = «NO» THEN 100
3130 PRINT «RESPOSTA INACEITAVEL»:GOTO 3100
3999 REM*SAIDA*
4000 STOP
```

O Programa 1B inclui muitas das técnicas até agora mencionadas e deve ser razoavelmente fácil de compreender. A extensão do programa original, comparada com a desta versão, mostra a forma como pode aumentar a sofisticação de um programa e o seu número de linhas. Contudo não se deve recear a grande extensão de um programa, porque normalmente nada tem a ver com a dificuldade do mesmo. No entanto, existem um ou dois pontos deste programa que merecem alguns comentários, se bem que siga de muito perto a estrutura apresentada na figura 3.3.

Os números de linha são usados liberalmente e permitem dispor de bastante espaço para ampliar qualquer sub-programa ou módulo. Note que as instruções REM no início de cada módulo receberam números de linha bastante estranhos, inferiores numa unidade ao valor seguinte. Isto permite-nos apagá-los (se mais tarde se verificar que a quantidade de memória disponível é escassa) sem fazer perigar o programa. Note que as linhas de destino ON GOTO ignoram estas REM's. Parece existirem duas escolas de pensamento relativamente a este aspecto. Alguns dizem que as afirmações REM são importantes e que desprezá-las em termos de programa encoraja os programadores a apagá-las depois de terem terminado o desenvolvimento do programa, o que é mau do ponto de vista de futuras modificações. Penso no entanto que se trata de uma decisão pessoal e que de resto não tem qualquer importância em termos de funcionamento do programa.

No que se refere ao subsequente apagamento das REM's, qualquer que seja a posição em que estas se encontrem no programa, é necessário ter bastante cuidado pois é possível cometer erros muito graves. Se uma GOTO ou uma IF THEN se referem a um dado número de linha

e esta foi apagada (porque se tratava de uma REM) o computador queixar-se-á, dizendo-nos que se trata de uma «instrução indefinida». O método mais seguro de se limpar as REM's consiste em deixar a palavra-chave REM, limpando apenas a parte textual. Deste modo, é sempre preservada a linha propriamente dita.

O Programa 1B é considerado apenas um exemplo, ilustrando as palavras-chave BASIC já mencionadas; o simples problema do círculo, da esfera e do cone não merecem talvez todo o «software» que o rodeia, mas não devemos esquecer que aqui apenas nos interessam os princípios de programação. Escolhemos estas equações porque são simples e dificilmente contribuirão para aumentar a dificuldade de compreensão das linhas não matemáticas que, aliás, correspondem a cerca de 90 % do programa.

O método de obter R^2 foi alterado nos diversos programas. As potências podem ser programadas usando o ↑ , ou recorrendo a uma multiplicação repetitiva (R*R). Se se trata de uma potenciação elevada, por exemplo R^4 é mais lógico usar-se R ↑ 4. Se a potência é fraccionária, por exemplo $R^{3,42}$, nem sequer podemos escolher; será forçoso empregar a seta.

Programa 1B com matemáticas diferentes. Neste caso a estrutura mantém-se, sendo apenas necessárias as seguintes modificações:

a) Alterar o texto na página de opção (linhas 120 a 140);

b) Talvez as *condições* requeiram diferentes limites nas validações de input (linhas 1020, 2020 e 3020).

c) o material textual nas linhas que apresentam as respostas (linhas 1070, 1080, 2070, 2080, 3090) e os nomes das variáveis necessitariam de ser alterados.

Poderiam existir evidentemente mais opções, mas existiria também um limite máximo imposto ao total de números de destino permitidos na instrução ON GOTO. Isto não é um problema: pode haver tantas opções sepa-

radas quantas se desejar desde que uma delas se refira a outra *página* de opções.

EXIT (saída) de programa. Uma das opções deve sempre incluir STOP ou EXIT, como acontece no programa 1B. É sempre possível sair de um programa carregando na tecla STOP, mas este método não é muito profissional. Existe ainda uma outra razão. Alguns programas requerem uma grande quantidade de dados que devem ser fornecidos pela pessoa que os utiliza. Os programas de acesso a registos, por exemplo, necessitam de alguém que de entrada a esses registos. Pense na raiva que sentirá se acabar de dar entrada a 200.º registo e durante a preparação do 201.º premir acidentalmente a tecla STOP obrigando o programa a passar ao estado de espera de ordens. Existe normalmente maneira de recuperar os dados e colocá-los novamente no programa, mas será necessário algum profissionalismo para o fazer. No entanto devemos sempre partir do princípio de que o operador de um programa é um «idiota», lembra-se? É habitual inibir a acção da tecla STOP no início destes programas incluindo neles um modo de a ultrapassar, e daí o uso da opção EXIT. É improvável que esta tecla seja carregada acidentalmente. O programa 1B não dispõe de nenhum esquema de inibição da tecla STOP, mas é bom desenvolver bons hábitos desde início. O método de inibição desta tecla, e a sua recuperação para EXIT, obriga normalmente ao uso da função POKE, mas é melhor deixar a discussão deste assunto para mais tarde.

Outras ideias para a estrutura do Programa 1B. O dinheiro exerce um certo fascínio sobre alguns de nós, e portanto vejamos um exemplo de utilização de cálculos diferentes dos anteriores:

$$A = P\left(1 + \frac{I}{200}\right)^{2Y}$$

Se se investirem P libras numa sociedade de construções que ofereça um juro de I % e o deixarmos nela durante Y anos, o nosso dinheiro aumentará para A

libras. Este livro não é dedicado a problemas de contabilidade, pelo que não tentaremos explicar porque razão o que acabámos de dizer é verdadeiro. A instrução INPUT pode pedir P, I e Y. O validação dos inputs deve verificar se I e P não são negativos, mas os limites superiores dependerão apenas das suas preferências pessoais. O módulo que tratará o problema poderá ter a seguinte forma:

```
 999 REM*AUMENTO DE CAPITAL*
1000 CLS:K = 200
1010 INPUT«INDIQUE CAPITAL INVESTIDO»;P:
     PRINT
1020 IF P < 1 THEN PRINT «VALOR MÍNIMO É 1»:
     GOTO 1010
1030 INPUT«INDIQUE TAXA DE JURO»;I:PRINT
1040 IF I < O THEN PRINT «JURO DEVE SER
     POSITIVO»: GOTO 1030
1050 INPUT«INDIQUE NÚMERO DE ANOS DE
     INVESTIMENTO»;Y:PRINT
1060 IF Y < 0.5 THEN PRINT«0.5 ANOS É O
     MÍNIMO»: GOTO 1050
1070 REM*CÁLCULO*
1080 A = P*(1 + I/K) ↑ (2*Y)
1090 A = INT(100*A + 0.5)/100:REM*ARREDONDAR
     2 CASAS DEC*
1100 PRINT«O SEU INVESTIMENTO VALE AGORA
     «A» LIBRAS»:PRINT:PRINT
1110 INPUT«NOVAMENTE?»;K$
1120 IF K$ = «YES» THEN 1000
1130 IF K$ = «NO» THEN 100
1140 PRINT«RESPOSTA NÃO ACEITE»: GOTO 1110
```

Note que os números das linhas permitem a substituição directa deste módulo no programa 1B, em vez do módulo «Círculo» nele existente. Naturalmente, seria estranho que coexistissem dois módulos, um sobre geometria e outro sobre finanças, num mesmo programa, mas aqui apenas nos interessam princípios, como já se disse. De facto, seria mais lógico juntar a este outros módulos

de finanças. Por exemplo, seria útil manter a mesma equação mas usá-la de tal modo que A, I e P fossem inputs e Y output.

Validação do output. Como poderemos saber se são correctas as saídas fornecidas pelo computador? Isto não significa que o computador possa ter errado, é altamente improvável! O culpado é sempre o programador, sendo de facto essencial verificar todos os tratamentos matemáticos recorrendo a um método qualquer. Na maior parte dos casos é fácil fazê-lo escrevendo alguns números num pedaço de papel. No exemplo anterior, por exemplo, se P = 1000, I = 10, Y = 1, a quantidade A deve ser 1102,5. Os 2,5 «extra» devem-se ao juro composto resultante do acréscimo do juro duas vezes por ano. Se obtiver uma resposta diferente quando passa o programa, verifique novamente a linha 180.

Economia de programação. A palavra «economia» tem muitos sentidos, mas aqui considera-se que diz respeito à quantidade de memória ou ao número de instruções usadas num programa para alcançar o objectivo desejado. O Programa 1B não seria muito bom sob este ponto de vista, pelo que iremos agora estudar as razões e os métodos empregues para melhorar a economia da programação.

RESUMO

Os fluxogramas, se bem que desprezados por alguns, podem ser bastante úteis na preparação do «plano de ataque» antes de realizar a codificação do programa. Os fluxogramas permitem definir a estratégia de resolução de um problema, e as linhas escritas em BASIC definem a sua táctica.

As instruções INPUT devem ser sempre seguidas de alguma forma de verificação para o caso de o operador ter introduzido dados pouco usuais.

A instrução IF condição THEN acção pode alterar a sequência de execução de um programa. Se a condição é

verdadeira, todas as instruções que se encontram nessa linha depois da palavra-chave THEN são executadas. Se a condição não é verdadeira o resto da linha é ignorado e o computador passa para a linha seguinte.

O losango usado no fluxograma corresponde à instrução IF THEN.

Se é necessário que sejam verdadeiras simultaneamente duas ou mais condições, deve-se incluir a palavra AND entre elas na parte condicional da instrução IF THEN.

Se existem várias condições possíveis mas é apenas necessário que uma qualquer delas seja verdadeira, pode-se utilizar OR.

A instrução ON GOTO equivale a um interruptor de várias vias, sendo usada para dirigir a execução do programa para um dos vários módulos possíveis.

ON S GOTO n1,n2,n3... significa que o conteúdo da variável numérica S decidirá qual a linha do programa que será executada em seguida. Se S = 1, o destino será a linha n1; se S = 3, o destino será a linha n3, etc.

É possível mas pouco conveniente saltar para uma instrução REM, porque o subsequente apagamento desta linha pode tornar-se necessário devido à falta de espaço de memória.

As linhas de destino da instrução ON GOTO devem ser bastante espaçadas quando se desenvolve o programa.

Deve-se incluir uma instrução que permite ao operador fazer a máquina sair para fora do programa sem utilizar a tecla STOP.

Todos os programas devem ser verificados por algum método externo, usando exemplos com números simples.

Os computadores dão sempre respostas correctas; são muitas vezes os programadores que lhes fazem perguntas erradas...

4

«LOOPS» E SUBROTINAS

Muitas das tarefas confiadas a um computador têm uma natureza repetitiva. De facto, poderíamos ir ainda mais longe a afirmar que só as tarefas minimamente repetitivas ganham em ser tratadas por computador. Para tratar as contas do Sr. Amaral (que talvez seja um milionário) será necessário um programa praticamente igual ao usado para as contas do Sr. Barreiros(que não o é certamente). Perguntar ao operador «Deseja repetir este programa?» envolve o mesmo número de linhas independentemente da posição em que se encontra no interior do programa. Se examinarmos um programa extenso e notarmos que um dado bloco de linhas é repetido muitas vezes, mesmo que exista uma pequena diferença de cada vez, devemos suspeitar que o programa foi escrito sem grande atenção a problemas de economia. Examinemos o programa 1B do capítulo anterior tentando descobrir repetições. As linhas 1040 e 1050 são praticamente idênticas às linhas 2040 e 2050, assim como as linhas 3060 e 3070. Notamos ainda que alguns dos textos enviados pela máquina para o visor são repetidos várias vezes.

Este capítulo introduz dois mecanismos poderosos para resolução económica dos problemas de repetição.

O «LOOP»

Um «loop» (ciclo) é um bloco de linhas repetido várias vezes. Um loop «sem ifm», como o seu nome indica, repe-

te-se constantemente e corresponde quase sempre a um erro de programação.

Vejamos um exemplo de um loop que se comporta ajuizadamente:

```
100  A = 1
110  PRINT A
120  A = A + 1
130  IF A < 21 THEN 110
140  STOP
```

Este loop imprimirá na saída os números inteiros 1, 2,3, ..., 20, quando em seguida.

Quando se discutem loops, usam-se os seguintes termos:

Inicialização é a preparação necessária antes de dar entrada a um loop. A linha 100 inicializa A como tendo o valor 1, antes de o programa entrar num loop.

Incrementação é a quantidade acrescentada à variável de cada vez que o loop é executado. A linha 20 trata neste caso a incrementação, acrescentando um 1 de cada vez.

A *variável do loop* é a variável *activa* do loop que se encontra sujeita à incrementação; neste caso é A.

Teste de saída de loop. Deve existir algum modo de abandonar o loop depois de este ter sido executado o número de vezes necessário. Este problema é resolvido na linha 130.

Vejamos agora um exemplo de loop sem fim:

```
100  A = 1
110  PRINT A
120  A = A + 1
130  GOTO 110
```

O incremento pode ser positivo ou negativo. Pode-se portanto inicializar com um valor elevado da variável e subtrair de cada vez que o loop é executado; tratar-se-á

então de um *incremento negativo*. Observemos o exemplo seguinte:

```
100  K = 20
110  PRINT K
120  K = K — 1
130  IF K > 0 THEN 110
140  STOP
```

Este loop imprimirá os inteiros 20, 19, 18, ..., 1, parando em seguida.

Um dos erros constantes na programação de loops é «faltar um». Esta calamidade, quando ocorre, deve-se a um operador condicional incorrecto existente na linha IF THEN. Fornecem-se em seguida vários exemplos de loops simples, devendo cada um deles ser cuidadosamente observado.

Objectivo: imprimir todos os inteiros entre 1 e 27, excepto 5 e 9.

```
100  A = 1
110  IF A = 5 OR A = 9 THEN 130
120  PRINT A
130  A = A + 1
140  IF A < 28 THEN 110
150  STOP
```

Objectivo: imprimir todos os números inteiros entre 17 e 3456 que sejam exactamente divisíveis por 17.

```
100  A = 17:D = 17
110  REM*DIVISÃO DE VERIFICAÇÃO*
120  K = A/D
130  IF K <> INT(K) THEN 150:
     REM*VERIFICAÇÃO DE RESTO*
140  PRINT A;:REM*IMPRIME NA HORIZONTAL*
150  A = A + 1
160  IF A < 3457 THEN 110:REM*SAIR DE TESTE*
170  STOP
```

Existem outras maneiras de desenvolver este sistema que seriam mais eficazes e mais rápidas, mas estamos apenas a fazer um exercício sobre «loops». Um método muito mais simples consistiria em incrementar A em 17 de cada vez, mas nem sempre é fácil cortar atalhos quando se luta ainda por dominar a linguagem BASIC. Com o tempo, a linguagem transforma-se numa espécie de segunda natureza, deixando a mente livre para analisar um problema e apresentar uma solução inteligente ou habilidosa que deixará os seus amigos ofuscados com o seu brilhantismo. Neste momento convirá porém uma palavra de aviso, porque o programador cheio de truques pode ser uma ameaça para ele próprio e para os seus colegas. Este tipo de sujeito, sempre à procura de métodos que impressionem os outros, pode por vezes ver-se metido em sérios apertos devido ao seu próprio «brilho», e acabará por não conseguir a ajuda de ninguém devido à natureza confusa do seu modo de codificação.

Não se deve pensar que isto significa que devamos todos continuar a usar o primeiro método que nos veio à cabeça. É sempre útil pensar cuidadosamente num problema antes de correr para o teclado. Existe um meio termo entre uma solução demasiado extensa e outra excessivamente económica, que emprega uma estrutura tão tortuosa como uma intriga...

Voltando novamente aos loops, vejamos ainda mais alguns exemplos:

Objectivo: imprimir uma tabela de raízes quadradas dos números inteiros de 1 a 15 com duas casas decimais.

```
100  I = 1:D = 100
110  S = SQR(I):S = INT(D*S + 0.5)/D
120  PRINT I,S
130  I = I + 1
140  IF I < 16 THEN 110
150  STOP
```

Objectivo: imprimir a soma dos inteiros 1 a 100.

```
100  A = 1:S = 0
110  S = S + A
120  A = A + 1
130  IF A < 101 THEN 110
140  PRINT«A SOMA É»; S
150  STOP
```

A resposta deve ser 5050.
Objectivo: ilustrar um loop de pergunta e resposta.

```
100  A$ = «ESTOCOLMO» :S = 1
110  INPUT«QUAL É A CAPITAL DA SUÉCIA?»; Q$
120  IF Q$ <> A$ THEN PRINT«ERRADO»:
     S = S + 1:GOTO 110
130  PRINT «CORRECTO CONSEGUIU ACERTAR
     EM»; S
140  STOP
```

Este é um exemplo de um loop que só pode ser concluido pelo próprio operador. Enquanto a resposta não for correcta a máquina manter-se-á no loop, aumentando progressivamente o número de tentativas. Como é óbvio, se o operador sabe listar (LIST) um programa, poderá sempre descobrir a resposta olhando para a primeira linha.
Objectivo: imprimir uma fiada de 15 asteriscos

```
100  A = 1:K$ = «*»
110  PRINT TAB(A); K$;
120  A = A + 1
130  IF A < 16 THEN 110
140  STOP
```

Note como a posição TAB pode ser uma variável.

A instrução FOR NEXT

Os exemplos anteriores de formação de loops são bastante simples, seguindo sempre um esquema lógico.

O BASIC oferece-nos no entanto uma alternativa de outro tipo usando uma instrução com o seguinte formato geral:

```
100  FOR A = S TO F STEP I
 •
 •
 •
 •          (processamento)
 •
 •
 •
 •
500  NEXT A
```

A linha 100 é a parte superior do loop. A linha 500 é a sua parte inferior. O processamento no interior do loop é repetido um certo número de vezes, dependente dos valores definidos na linha 100:

A é a variável do loop.
S é o valor inicial de A.
I é o incremento acrescentado a A de cada vez que o loop é executado.
F é o valor final de A.

Exemplo:

```
100  FOR A = 1 TO 20 STEP 1
120  PRINT A
130  NEXT A
140  STOP
```

Note a simplicidade desta forma: não é necessária qualquer linha de incrementação; nenhum teste para saída do loop; nenhuma linha separada para inicializar a variável. Os números inteiros entre 1 e 20 são impressos usando apenas três linhas de instruções. Compare este com o exemplo dado no início deste capítulo... ambos atingem o

mesmo objectivo. Se escrevermos tudo na mesma linha, a economia obtida é ainda mais evidente:

```
100 FOR A = 1 TO 20 STEP 1:PRINT A: NEXT A:
    STOP
```

Estude os seguintes exemplos de loops FOR:

Objectivo: imprimir uma tabela de potências cúbicas dos números 1, 1,1, 1,2, ..., 3,0.

```
100 FOR N = 1 TO 3 STEP 0,1
110 PRINT N,N ↑ 3
120 NEXT N
```

Objectivo: imprimir todos o inteiros nos limites 1 a 1000 que são exactamente divisíveis por 13 e por 23.

```
100 D1 = 13:D2 = 23:REM*ATRIBUIR  DIVISORES*
110 FOR I = 1 TO 1000 STEP 1
120 REM*DIVISÕES DE VERIFICAÇÃO*
130 K1 = I/D1:K2 = I/D2
140 IF INT(K1) = K1 AND INT(K2) = K2 THEN
    PRINT I
150 NEXT I
```

A impressão deverá ser 299, 598 e 897, que são os únicos números que satisfazem o objectivo. O programa demora alguns segundos a descobrir cada um destes números, pelo que será útil descobrir uma solução mais eficaz.

Objectivo: imprimir os números inteiros entre 1 e 20 pela ordem inversa.

```
100 FOR I = 20 TO STEP-1
110 PRINT I
120 NEXT I
```

Objectivo: permitir ao operador dar entrada a 20 dados na variável de quadro A$(N).

```
100 DIM A$(20):REM*INFORMAR QUE 20 ITEMS
    NECESSITAM DE ESPAÇO EM MEMÓRIA
110 FOR N = 1 TO 20 STEP 1
120 INPUT«INDICAR ITEM DE DADO»;A$(N)
130 NEXT N
140 PRINT «DERAM ENTRADA TODOS OS ITEMS
    DE DADOS»
150 STOP
```

Talvez o leitor se veja forçado a consultar novamente a secção sobre Variáveis Subscritas e quadros no capítulo 2 para conseguir perceber o sentido deste programa. O primeiro dado será colocado em A$(1), o seguinte em A$(2), e o último em A$(20). Para verificar se os dados foram de facto armazenados no quadro, elimine STOP na linha 150 e acrescente o seguinte:

```
150 FOR N = 1 TO 20 STEP 1:REM*IMPRIMIR
    DADOS*
160 PRINT A$(N)
170 NEXT N
180 STOP
```

Regras que governam os loops FOR

1. Os parâmetros na parte superior do loop estão sujeitos às seguintes limitações na maior parte dos BASIC's:

a) A variável do loop pode ser uma constante ou uma variável de vírgula flutuante, mas não pode ser uma variável inteira ou uma variável subscrita;

b) Não existem quaisquer limitações quanto à inicialização, finalização e incrementação excepto que não devem ser cadeias de caracteres. Podem até ser expressões complexas.

Por exemplo, é perfeitamente legal utilizar o seguinte género:

```
FOR A = B + C*D TO A(N) — 4*G STEP J/3.5
```

2. Nunca entre no meio de um loop FOR, pois apenas verá escrito no visor uma mensagem de erro do tipo «NEXT sem FOR».
3. É permitido sair de um loop FOR e regressar.

Redes de loops FOR

As pulgas pequenas podem viver das pulgas grandes. Do mesmo modo, os pequenos loops FOR podem «viver» dentro de loops FOR mais extensos. Diz-se então que existe uma rede de loops e, na maior parte dos BASIC's, o entrelaçamento destes loops pode ter bastantes níveis. Um loop interior pode ser executado muitas vezes por cada execução do loop onde está contido.

Vejamos um exemplo

```
100  FOR M = 1 TO 5 STEP 1
110  FOR N = 1 TO 4 STEP 1
120  PRINT M*N,
130  NEXT N
140  PRINT:PRINT
150  NEXT M
160  STOP
```

Quando este programa é executado, obtém-se na saída o seguinte:

```
1     2     3     4
2     4     6     8
3     6     9     12
4     8     12    16
5     10    15    20
```

Note que o loop interior (linhas 110 a 130) é executado quatro vezes por cada execução do exterior (linhas 100 a 150). A vírgula na linha 120 assegura que a impressão do loop interior se inicia em zonas definidas da mesma linha. A separação entre as linhas é devida às duas instruções PRINT na linha 140.

Regras das redes de loops FOR

1. Deve existir igual número de instruções NEXT e FOR.

2. Não existe normalmente qualquer limite quanto ao número de redes, excepto os devidos a considerações de espaço de memória disponível. Todos os loops FOR incompletos requerem uma área de armazenamento de parâmetros numa parte da memória a que se chama *stack*. O stack tem uma capacidade limitada, pelo que pode haver um erro de «falta de memória» mesmo que haja ainda muita memória «normal» disponível.

3. As variáveis de loops em redes de loops *devem ser diferentes.*

4. Os loops devem encontrar-se totalmente contidos nos loops exteriores. Não se devem sobrepor.

Variações em loops FOR. Certos BASIC's permitem-lhe escrever NEXT sem a variável, isto é:

```
100  FOR S = 1 TO 20
110  NEXT
```

O compilador considera que o NEXT pertence ao FOR executado há menos tempo.

```
100  FOR D = R TO K STEP 1
110  FOR H = Y TO B STEP 1
120  NEXT
130  NEXT
```

O compilador resolverá facilmente o assunto considerando que o NEXT da linha 120 pertence ao FOR da 110; e que o NEXT da linha 130 pertence ao FOR da linha 100.

Um outro «atalho» por vezes permitido tem a ver com a parte STEP da instrução FOR. Se o STEP deve ser 1, pode ser omitido; FOR S = 5 TO 57 significa então FOR S = 5 TO 57 STEP 1.

Uma outra variante, que pode ser aborrecida a menos que seja conhecida, diz respeito ao valor da variável de loop imediatamente à saída do loop. Alguns BASIC's asseguram que este valor seja o último executado (valor final). Assim, em FOR N = 1 TO 40, depois da saída em NEXT, N tem o valor 40. Outros BASIC's permitem que N passe ao valor seguinte (41 neste caso). Tudo depende do modo como o compilador está escrito, e em particular do facto do teste de final de loop ser realizado antes ou depois da incrementação. Vejamos alguns outros exemplos de redes de loops.

Objectivo: imprimir os números 1 a 20 *muito lentamente.*

```
100  FOR N = 1 TO 20 STEP 1
110  FOR D = 1 TO 1000 STEP 1:REM*ATRASAR
     EFEITO*
120  NEXT D
130  PRINT N
140  NEXT N
150  STOP
```

A maior parte dos microcomputadores possuem um «relógio» incorporado que pode ser usado para atrasos de precisão. No entanto, um modo muito simples de introduzir um atraso num programa consiste em recorrer a um loop FOR que não faça nada que não seja ser executado um dado número de vezes. O loop interior do programa acima (linhas 110, 120), conta até 1000 de cada vez antes de passar à linha PRINT. O número 1000 foi escolhido arbitrariamente e pode ser aumentado se o atraso for insu-

ficiente. Se for necessário um atraso rigoroso, é simples temporizar o loop com um cronómetro e obter uma escala de factores válida para futuros usos. Não exagere no entanto no número de execuções do loop. Se usar, por exemplo, 1E20, passarão bastantes *séculos* antes da primeira impressão!

Objectivo: imprimir uma tabela de raízes quadradas dentro dos limites escolhidos pelo operador.

```
100 INPUT«INDICAR MENOR NÚMERO DOS
    LIMITES» S:PRINT
110 INPUT«INDICAR MAIOR NÚMERO DOS
    LIMITES»;F:PRINT
120 INPUT«INDICAR INCREMENTO»;I:PRINT
130 FOR S = S TO F STEP I
140 PRINT N,SQR(N)
150 NEXT N
160 STOP
```

SUBROTINAS

Se consultar novamente o Programa 1B do Capítulo 3, notará a grande semelhança entre certos blocos de linhas. As linhas 1040 a 1060 são quase idênticas às linhas 2040 a 2060 e às linhas 3060 a 3080. Estudando o programa com maior cuidado, podem-se detectar semelhanças entre outros blocos de linhas. Sente-se instintivamente que deve existir algum método de eliminar estas repetições desnecessárias. A resposta reside numa técnica designada por *subrotina.*

Uma subrotina é definida como um bloco de códigos, escrito apenas uma vez mas que permite ao programa «chamá-lo» tantas vezes quantas as necessárias. No que se refere ao formato de uma subrotina, a única regra importante é que a última instrução seja RETURN. A subrotina pode existir em qualquer ponto no «mapa de linhas», se bem que seja vulgar (mas não necessariamente o melhor) colocar as subrotinas na parte final do programa.

Para chamar uma subrotina, usa-se a instrução GOSUB n, onde n é o número de linha onde se inicia a subrotina. O exemplo seguinte deverá ilustrar esta ideia:

```
100  GOSUB 2000
110
•
•
•
•
230  GOSUB 2000
240
•
•
•
•
300  GOSUB 2000
310
•
•
•
•
2000 REM*ARREDONDAR K PARA DUAS CASAS
     DECIMAIS*
2010 K = INT(1000*K + 0.5)/100
2020 RETURN
```

A subrotina (que é trivial mas adequada como exemplo), é primeiramente chamada na linha 100, levando o programa a passar para a linha 2000. A instrução RETURN que termina a subrotina levará o programa a passar automaticamente para a instrução que se segue à GOSUB que chamou a subrotina, que neste caso é a linha 110. A subrotina é depois chamada novamente na linha 230, e o RETURN é feito para a linha 240. É novamente chamada na linha 300, sendo o RETURN feito para a linha 310.

Chamamento de subrotinas

Antes de usar (chamar) uma subrotina pode ser necessário utilizar algumas instruções de atribuição preliminares. No exemplo anterior, a subrotina das linhas 2000 a 2020 considera obviamente que o número a arredondar existe na variável K de vírgula flutuante. Isto significa que se quisermos usar a subrotina devemos primeiramente assegurar que o número se encontre em K, ou seja temporariamente atribuído a K e recuperado para a sua original posição, ou sucede o RETURN. Como exemplo, suponhamos que se pretende arredondar a variável presente em S. O método a usar seá o seguinte:

```
220  K = S
230  GOSUB 2000
240  S = K
```

Isto pode ser desagradável porque as subrotinas servem em princípio para poupar trabalho de codificação e se houver necessidade de prepará-las deste modo o lucro acaba por não ser muito grande... serão talvez perdidas ainda mais linhas do que na programação sem subrotinas. Nestas condições, a «substância» da subrotina ocupa apenas uma linha (2010), mas acabam por ser necessárias três linhas para chamar e preparar a subrotina e mais três para a subrotina propriamente dita (se contarmos a REM). Seria melhor esquecer a subrotina e usar a linha de arredondamento de cada vez que fosse necessária.

A validade do que se disse é inegável no exemplo apresentado, mas na maior parte dos casos a subrotina é muito mais complexa, ocupando bastantes linhas, e o par de linhas usado para preparação tornar-se-á insignificante. Se, como é óbvio, o nome da variável do programa principal for o mesmo da variável da subrotina, tornam-se desnecessárias as atribuições de preparação; isto nem sempre é possível, no entanto.

Quando usar subrotinas

Tendo em conta os comentários anteriores, podemos definir as seguintes regras:

1. Se o objectivo de um módulo não é trivial e será provavelmente usado várias vezes num dado programa, as subrotinas permitirão poupar linhas de codificação, e portanto espaço de memória.
2. Se o objectivo é simples e apenas requer uma ou duas linhas, será provavelmente tão eficaz omitir a subrotina e escrever directamente a codificação no programa principal, mesmo que isto obrigue a repetições.
3. Se a velocidade de execução é o critério principal, as subrotinas devem ser evitadas tanto quanto possível porque envolvem sempre uma perca de tempo.

Antes de abandonar este assunto, seria conveniente sublinhar que apesar do que foi dito poder ser interpretado como uma espécie de campanha anti-subrotinas, não é essa a nossa intenção. As subrotinas são muito usadas em todos os tipos de programas e devem sê-lo a menos que haja contra-indicações claras.

A escolha dos nomes das variáveis

Deve-se ter bastante cuidado quando se escolhem nomes para as variáveis das subrotinas de modo a evitar confusões. Se é usada uma variável K no programa principal e esta contém dados importantes, será desastroso chamar uma subrotina de algum outro ponto do programa que possa corromper esse conteúdo. Nas versões avançadas de BASIC, as variáveis das subrotinas podem ser *declaradas* «locais», relativas apenas à subrotina, o que significa que a sua identidade global é preservada. No entanto, devemos partir do princípio de que este luxo é bastante raro.

Um modo de salvaguardar contra a corrupção dos dados pelas subrotinas consiste em escolher nomes pouco

habituais para as suas variáveis. Assim, pode-se usar A8 em vez de A, ou B8$ em vez de B$, dado que é improvável que o programa principal utilize estes nomes. Como é óbvio, se o leitor for metódico, registará todas as variáveis usadas de modo a não haver qualquer perigo de misturas... mas há sempre dias maus e a sugestão dada reduz os riscos.

Bibliotecas de subrotinas

Não torne a inventar a roda. Se tiver escrito ou copiado uma subrotina de um amigo, que é passível de uso geral, guarde-a em banda ou disco para uso futuro. Deste modo, pode constituir gradualmente uma «biblioteca» de subrotinas úteis. Quando começar um programa novo, pode carregá-las primeiro; depois, quando o programa termina, as subrotinas não usadas podem ser apagadas.

Estas técnicas não são necessariamente usadas pelos programadores profissionais, mas os aspectos mais delicados desta arte (a programação ainda não é uma ciência) podem ser aprendidos gradualmente. O que é importante nas fases iniciais é precavermo-nos contra a frustração. Todos os truques que possam conduzir à obtenção de bons programas, mesmo que não tenham grande elegância, ajudá-lo-ão a ultrapassar o período perigoso.

Redes de subrotinas

Uma subrotina pode chamar outra, que por sua vez pode chamar uma terceira. Este «entrelaçamento» de subrotinas pode ser realizado a qualquer nível, dependendo apenas da quantidade de memória da área «stack». Por exemplo:

```
200  GOSUB 2000
210
```

```
•
•
•
2000 REM *SUBROTINA (EXTERIOR)*
•
•
•
•
2050 GOSUB 3000
•
•
•
•
2060
•
•
•
2100 RETURN
•
•
•
•
3000 REM*SUBROTINA (INTERIOR)*
•
•
•
•
3050 RETURN
```

O primeiro chamamento a uma subrotina é realizado na linha 200, com GOSUB 2000. Quando esta subrotina termina a sua tarefa, o RETURN da linha 2100 passa novamente o programa para a linha 210. Mas ainda durante esta é feita chamada a outra subrotina com GOSUB 3000, na linha 2050. A segunda subrotina «retorna» na linha 3050 para a linha 2060.

A figura 4.1 talvez ajude o leitor a compreender este «entrelaçamento» das subrotinas.

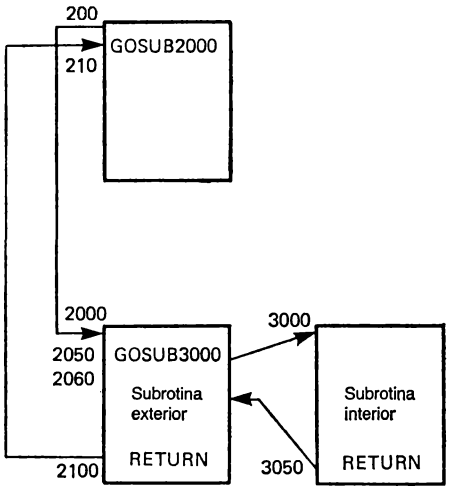

Fig. 4.1. — Fluxograma de um «entrelaçamento» de duas subrotinas.

Exemplos de subrotinas de uso geral

Os exemplos seguintes iniciam-se na linha 1000 mas é fácil alterá-las de modo a poder integrá-las em qualquer ponto do programa. Para ilustrar o método de chamada, fornece-se um curto programa de ensaio no final de cada subrotina que pode servir para ilustrar o seu uso.

Subrotina 1

```
999  REM*CARREGAR NA TECLA DE ESPAÇO PARA
     START/STOP/START/STOP ETC*
1000 GET K$:IF K$ = «»THEN 1020
1010 GET K$:IF K$ <> «»THEN 1010
1020 RETURN
```

Programa de ensaio:

```
100 FOR A = 1 TO 200 STEP 1
110 GOSUB 1000
120 PRINT A, SQR(A)
130 NEXT A
140 STOP
```

Esta subrotina é útil em programas que enviam para o exterior as informações a uma velocidade excessiva. É possível «congelar» qualquer página carregando na tecla de espaços. Carregando novamente nela recomeçará a passagem do texto. Não é necessário qualquer preparação para usar esta subrotina.

Subrotina 2

```
999 REM*ORDEM DE PAUSA POR TECLADO*
1000 PRINT«CARREGAR QUALQUER TECLA PARA
     CONTINUAR»
1010 GET K$:IF K$ = «»THEN 1010
1020 RETURN
```

Programa de ensaio:

Qualquer programa que deva ser interrompido num ponto. Por exemplo, uma página de texto que o operador deva ler mais lentamente. Carregando em qualquer tecla (normalmente outra que não a de STOP) pode-se continuar o programa.

Subrotina 3

```
999 REM*IMPRIMIR N8 COM VÍRGULA DECIMAL
    EM TAB(T8)*
1000 N8$ = STR$(INT(N8))
1010 IF INT(ABS(N8)) < 1 THEN N8$ = «1»
1020 PRINT TAB(T8 — LEN(N8$))N8
1030 RETURN
```

Programa de ensaio:

```
100 INPUT«INDICAR QUALQUER NÚMERO DE
    VÍRGULA FLUTUANTE»;N8
110 INPUT«INDICAR POSIÇÃO DA VÍRGULA
    DECIMAL»;T8
120 GOSUB 1000
130 GOTO 100
```

Para experimentar isto indicar um número, por exemplo 34,567, e notar a posição da vírgula decimal na saída. Em seguida indicar outro número, por exemplo 3456,79 com a mesma posição TAB e notar o alinhamento do ponto. A subrotina é muito útil quando se imprimem séries de números em formato «contabilista».

Subrotina 4

```
 999 REM*CONVERTER N8 TO FACTORIAL N8*
1000 M8 = 1
1010 FOR K8 = 1 TO N8 STEP 1
1020 M8 = M8*K8
1030 NEXT K8
1040 N8 = M8
1050 RETURN
```

Programa de ensaio:

```
100 INPUT«INDICAR QUALQUER INTEIRO»;N8
110 GOSUB 1000
120 PRINT N8
130 GOTO 100
```

RESUMO

Um *loop* (ciclo) é um segmento de programa que se repete várias vezes antes de sair.

A variável de loop é aumentada ou diminuída numa quantidade fixa, de cada vez que aquele é executado.

Um loop sem fim não termina nunca, constituindo por vezes um erro de programação.

A preparação dos parâmetros do loop é chamada inicialização.

A incrementação é o processo de acréscimo de uma quantidade fixa à variável de loop.

Os incrementos podem ser positivos ou negativos.

O teste (ensaio) de saída determina o momento em que o número de execuções do loop satisfaz os parâmetros deste.

Uma forma simples, mas não a única, de formar um loop consiste em usar a instrução FOR.

A parte superior do loop possui o formato FOR A = S TO F STEP I. A parte inferior do loop é a palavra-chave NEXT A.

As variáveis S e F na instrução FOR constituem os valores inicial e final, respectivamente da variável de loop A.

A variável I é o incremento.

A variável de loop não deve ser uma cadeia de caracteres nem uma variável inteira. S, F e I podem ser expressões inteiras ou de vírgula flutuante.

Os loops FOR podem ser utilizados a vários níveis no interior uns dos outros, desde que nunca se sobreponham.

Cada rede de loop deve possuir uma variável de loop de nome diferente.

Um número excessivo de redes de loops «FOR» pode esgotar o espaço disponível na parte da memória designada por «stack».

Certas versões BASIC permitem o uso isolado de NEXT sem indicação da variável. Considera-se então que cada NEXT se refere ao FOR mais recente.

As subrotinas são módulos que podem ser chamados de qualquer ponto do programa.

As subrotinas permitem poupar blocos de codificação repetitivos.

Certas subrotinas requerem uma ou mais declarações de atribuição antes de chamar a subrotina para o caso das variáveis requererem equiparação.

As subrotinas são chamadas usando GOSUB n, onde n é o número da linha onde se inicia a subrotina.

A última instrução de uma subrotina deve ser RETURN. Consegue-se assim que o programa principal recomece a partir da instrução que se segue imediatamente à GOSUB que chamou a subrotina.

As subrotinas permitem economizar memória mas fazem perder algum tempo de processamento.

As subrotinas podem ser levantadas e armazenadas em banda para uso subsequente noutros programas, desde que sejam passíveis de uso geral.

Tenha cuidado ao recorrer a subrotinas de uso geral cujos nomes das variáveis possam corromper as variáveis do programa principal. Este risco é minimizado escolhendo nomes pouco habituais nas subrotinas.

As subrotinas podem chamar outras subrotinas para executar parte da tarefa. Chama-se a isto, «nesting» (rede) de subrotinas.

Um número excessivo de níveis de subrotinas pode esgotar a capacidade do «stack», apesar de haver bastante memória disponível noutros locais.

Todas as subrotinas usadas numa «rede» devem poder ser usadas isoladamente.

5

DADOS E QUADROS

Certas palavras têm uma vasta gama de significados, e o melhor exemplo disto é o termo «dados». No que se refere aos computadores, definiríamos dado como «aquilo que tem um significado para o computador», mas isto incluiria praticamente tudo: palavras-chave, variáveis, texto em linguagem normal e até os números de linha. Para diminuir o número de implicações da palavra e por uma questão de conveniência, poderemos definir dado do seguinte modo:

O dado é qualquer informação adicional requerida por um programa. Os programas devem sempre receber algumas informações. Por exemplo, será inútil pedir a um computador que calcule a raiz quadrada de X se não lhe dermos a conhecer o valor de X. Poderemos resolver evidentemente o problema através de uma atribuição simples como X = 34.567. O «dado» é então 34.567, mas neste caso constitui simplesmente uma constante fornecida pelo próprio programa e que portanto se encontra fora do espírito sugerido pela definição referida.

Existem três maneiras de fornecer dados a um programa:

1. Através da instrução INPUT. Neste caso é o operador que fornece os dados. A instrução GET é semelhante mas fornece apenas um caracter.
2. «Carregando» o dado a partir de uma memória periférica sob controlo do programa. Os dados podem

estar armazenados em cassettes, bandas magnéticas, diskettes ou discos. Este método de recepção de dados baseia-se portanto no uso de *periféricos*. Como os periféricos são usados para armazenar tanto dados como programas, é habitual designá-los por *ficheiro* (o que é outro termo excessivamente usado), distinguindo no entanto entre *ficheiros de programa e ficheiros de dados*. As técnicas de uso destes ficheiros serão indicadas noutro capítulo.

3. Recorrendo a instruções READ/DATA.

INSTRUÇÕES READ/DATA

Estas duas instruções são sempre usadas em conjunto. Assim, se existe uma instrução READ no programa, o computador espera encontrar (em qualquer outro ponto) uma instrução DATA correspondente. Antes de entrarmos em pormenores, devemos admitir que estas instruções não cabem estritamente no significado do termo «dado» indicado anteriormente. As instruções READ/DATA ocupam uma área bastante extensa porque, apesar de fornecerem os dados requeridos pelo programa, estes estão numa linha do próprio programa. No entanto, o método de recolha dos dados é significativamente diferente do empregue em atribuições simples.

As instruções READ/DATA são usadas principalmente para aquilo que pode ser designado por *dados semi-permanentes*, porque o formato permite alterá-las facilmente. Estudemos o exemplo seguinte:

```
100  READ A,B,C
 •
 •
 •
 •
 •
 •
2000 DATA 345,500,56.789
```

As instruções READ e DATA podem estar em qualquer ponto do programa e sempre relacionadas entre si, porque uma instrução READ procura todas as linhas do programa (para a frente e para trás) até ser encontrada a instrução DATA. Quando esta é descoberta, o primeiro item nela indicado é atribuído à primeira variável da instrução READ, o segundo à segunda variável, etc., sempre por ordem sequencial.

Assim, no exemplo acima, 345 é colocado em A, 500 em B e 56.789 em C. A principal regra a respeitar neste caso consiste portanto na necessidade de existirem tantos items de dados quantas as variáveis. Nestas condições, se a instrução READ englobar quatro variáveis, o computador espera encontrar quatro items de dados, e queixar-se-á no caso de isso não acontecer enviando uma mensagem de erro: «erro por falta de dados», ou algo semelhante.

Uma outra regra consiste em que depois de terem sido lidos (READ) todos os items de dados estes *não* devem ser lidos novamente (esta última regra está sujeita a uma excepção, que indicaremos mais tarde).

Linhas READ e DATA múltiplas

Existe um limite para o número de items de dados que podem ser escritos numa linha, mas é possível continuar a escrevê-los em linhas subsequentes, como se mostra em seguida:

```
1100 DATA 3,6,8,4,56,5,1,9,34,56
1120 DATA 5,67,3,45,789,3
```

Os dez items da primeira linha e os seis da segunda comportam-se, no que se refere às correspondentes instruções READ, como dezasseis items consecutivos. Do mesmo modo, as variáveis READ podem estender-se por várias linhas:

```
100  READ A,B,F,G1,KU,S,Y2,Z
120  READ R1,P2,P,TE,Q,X,H5,LI
```

As 16 variáveis read podem armazenar os 16 valores de dados.

Os items de dados podem ser numéricos, caracteres em cadeia ou misturas, mas as correspondentes variáveis READ devem estar-lhes equiparadas:

```
100  READ S,K$,R1
2000 DATA 475.7,BLODWEN,38
```

O item do meio é uma cadeia de caracteres e é aceite pela variável de cadeia existente na instrução READ na mesma posição. Note que as habituais aspas não são requeridas em torno da cadeia Blodwen nos items de dados.

Instruções RESTORE e CLR. Já se referiu que os dados, depois de utilizados por uma instrução READ, não devem ser lidos novamente. Se no entanto for essencial num programa voltar a lê-los, deve usar primeiro a palavra-chave RESTORE. Esta instrução conduz o chamado *pointer* (indicador) de dados novamente para o primeiro dado.

O *pointer* é um termo de programação (principalmente usado em código máquina ou código *assembly*), mas para os nossos objectivos pode ser considerado como um contador que regista o número de items de dados lidos sequencialmente pela instrução READ. À medida que é lido cada um, o contador em causa avança uma unidade, «apontando» para o seguinte. Quando todos os items de dados são lidos, o «pointer» fica uma unidade à frente, isto é, apontando para um item não-existente! Se se fizer uma nova tentativa de leitura, o computador não encontra qualquer dado e informa.

A instrução RESTORE obriga o pointer a voltar ao início dos dados, permitindo novamente a sua leitura.

A instrução CLR

Esta instrução é bastante brutal. Coloca todas as variáveis em zero; cadeias de caracteres são passados a

«cadeias nulas», o que significa «ausência de caracter», e mesmo as variáveis DIMensão são passadas a zero. CLR terá igualmente o efeito de RESTORE, se bem que seja importante distinguir entre ambas:

RESTORE desloca simplesmente o «pointer» de dados para o início;

CLR limpa todas as variáveis na máquina;

CLR não afecta quaisquer instruções do programa.

A ordem NEW, já mencionada num capítulo anterior, é a mais definitiva de todas... apaga tudo, incluindo o programa.

Uso de DATA/READ para preencher um quadro

Em vez de utilizar a instrução READ na forma já anteriormente conhecida (variáveis separadas por vírgulas), pode-se usar um loop FOR juntamente com variáveis subscritas para colocar todos os dados num único quadro. Por exemplo:

```
100 FOR N = 1 TO 6 STEP 1
110 READ S(N)
120 NEXT
•
•
•
•
•
•
2000 DATA 3,6,89,3,5,1
```

3 será assim colocado em S(1), 6 em S(2)... e 1 em S(6). Se houvesse 200 items de dados, poderiam ser todos incluídos no quadro, desde que o loop FOR fosse alterado para FOR N = 1 TO 200.

Se os items de DATA fossem cadeias de caracteres, as linhas seriam iguais excepto a variável de quadro:

```
100  FOR N = 1 TO 8 STEP 1
110  READ K$(N)
120  NEXT N
•
•
•
•
2000 DATA LONDRES, INGLATERRA, PARIS,
     FRANÇA
2010 DATA COPENHAGA, DINAMARCA, OSLO,
     NORUEGA
```

Se bem que os exemplos tenham sempre colocado a instrução READ num número de linha menor do que o das instruções DATA, esta ordem pode ser invertida. É no entanto mais prático colocar DATA no final do programa porque, como já se disse, os dados podem desta forma serem considerados semi-permanentes. Assim, as cidades e países mencionados no último exemplo podem ser facilmente alterados fazendo uma listagem do programa (ordem LIST) até às últimas linhas e usando o cursor para alterar os dados ou escrever novamente as linhas.

Depois de os items de dados estarem armazenados no quadro, é relativamente fácil processá-los do modo que se deseje. A acção de leitura de um quadro, como se mostrou acima, constitui um módulo completo e será certamente o primeiro passo na maior parte dos programas que empreguem instruções DATA/READ. Para ilustrar a conveniência dos quadros de dados, consideremos que o módulo referido serviu para preencher o quadro K$(N):

Objectivo: imprimir o conteúdo do quadro sob a forma de uma lista.

```
200  FOR N = 1 TO 8 STEP 1
210  PRINT K$(N)
220  NEXT N
```

Objectivo: imprimir o conteúdo do quadro em quatro pares, cada um deles constituído por uma cidade e respectivo país.

```
200 FOR N = 1 TO 8 STEP 2
210 PRINT K$(N) TAB(12) K$(N + 1)
220 PRINT
230 NEXT N
```

Note o truque de somar dois de cada vez mas usando os subscritos N e N + 1 para assegurar que os pares sejam impressos na mesma linha. O segundo elemento de cada par é TAB(12) adiante, se bem que o número de TAB seja aqui uma escolha pessoal dependente apenas da largura do visor. Neste caso não seria prudente usar a vírgula como separador, porque os nomes geográficos podem estender-se para além do limite adequado a uma boa aparência estética. A saída será a seguinte:

```
LONDRES         INGLATERRA
PARIS           FRANÇA
COPENHAGA       DINAMARCA
OSLO            NORUEGA
```

Programa de perguntas e respostas

Para ilustrar ainda melhor estes quadros de dados, vamos desenvolver um programa que pode ser útil como teste de consolidação para crianças (ou adultos). O tema será o mesmo de anteriormente, corresponder as cidades com os seus países, mas para alterar a matéria tratada bastaria modificar a instrução DATA e uma ou duas das linhas de texto. A primeira tentativa poderá dar o seguinte resultado:

Programa 2

```
80  DIM Q$(10)
100 REM*LER PARA QUADRO*
```

```
110  FOR N = 1 TO 10 STEP 1
120  READ Q$(N)
130  NEXT N
200  S = 0:REM*PASSAR RESULTADO PARA ZERO*
210  REM*APRESENTAR PERGUNTAS E
     VERIFICAR RESPOSTAS*
220  FOR N = 1 TO 10 STEP 2
230  PRINT«QUAL É A CAPITAL DE»Q$(N):INPUT
     A$
240  IF A$ = Q$(N + 1) THEN S = S + 1
250  NEXT N
300  REM*APRESENTAR RESULTADO *
310  PRINT «O RESULTADO FOI»S
320  STOP
1000 READ *PARES DE PERGUNTAS E
     RESPOSTAS*
1010 DATA FRANÇA, PARIS, INGLATERRA, LON-
     DRES, DINAMARCA, COPENHAGA
1020 DATA SUÉCIA, OSLO, UTOPIA, ZABALONIUM
```

Note que as perguntas estão colocadas nas posições ímpares e as respostas nas posições pares. Isto pode explicar o raciocínio que justifica as linhas 230 e 240. Estas perguntas não pretendem ser Geografia de alto nível exceptuando, talvez, a última. Se alguma vez encontrarem a Utopia, a sua capital tanto pode ser Zabalonium como qualquer outra coisa! É uma pergunta que serve no entanto para impedir qualquer pessoa de obter um resultado de 100 %. Existe no entanto uma pequena falha que permite ao intelectual menos escrupuloso fazer batota. Se o leitor souber LISTar o programa, a resposta correcta aparecerá no visor, à frente da instrução DATA. No entanto, uma vigilância adequada deverá impedir que tal venha a suceder.

O programa está apenas em esboço livre de quaisquer influências «cosméticas», a fim de permitir a apreciação dos seus aspectos essenciais. Para desenvolver o programa ainda mais, será necessário pensar no impacto

que poderá ter sobre o indivíduo que responde ao teste. Tal como se encontra, o programa é um tanto «frio», pouco amistoso. Os acrescentos seguintes serviriam para humanizar a interacção entre o computador e o aluno:

1. Em primeiro lugar deveriam ser dadas instruções que ajudassem o aluno a accionar as teclas.
2. No caso de uma resposta errada, deveria existir uma opção que permitisse apresentar a resposta correcta.
3. O resultado poderia ser dado sob a forma de uma percentagem, em vez de um único número.
4. A percentagem deveria ser graduada, e acompanhada por palavras de elogio ou crítica.
5. As mensagens no visor deveriam incluir algumas linhas PRINT (em branco) a fim de ficarem a maior distância umas das outras.
6. O operador deveria poder voltar ao princípio depois de terminado o teste.

A versão que se segue (Programa 2A) incorpora as características referidas e contém algumas instruções REM que deveriam explicar o seu funcionamento interno. Está concebida, no seu início, de tal modo que o número de perguntas pode ser aumentado alterando simplesmente o valor «M». O título do tema pode ser alterado modificando o texto na linha 120. O texto da pergunta pode ser modificado na linha 370 de modo a adaptar-se à nova matéria das perguntas. As perguntas e respostas nas nas linhas DATA devem evidentemente ser alteradas. Verifique se o número total de items de dados concorda com «M».

Programa 2A

```
 80  REM***PERGUNTAS E RESPOSTAS***
100  CLR:M = 10:REM*NÚMERO 8 DE PERGUNTAS*
110  DIM Q$(M)
120  T$ = «CAPITAIS»:REM*TÍTULO DO TEXTO*
130  F = 0:REM*FLAG*
```

```
140 REM*LER PERGUNTAS E RESPOSTAS*
150 FOR N = 1 TO M STEP 1
160 READ Q$(N)
170 NEXT N
180 S = 0:REM*PASSAR PONTUAÇÃO PARA ZERO*
190 CLS:REM*LIMPAR VISOR*
200 PRINT TAB(12)T$:PRINT:PRINT
210 GOSUB 680: REM*IMPRIMIR LINHA*
220 PRINT«QUANDO APARECER UMA PERGUNTA,
    ESCREVA»:PRINT
230 PRINT«A SUA RESPOSTA. DEPOIS CARREGUE»:
    PRINT
240 PRINT TAB(12)«NA TECLA DE RETURN»
250 GOSUB 680
260 PRINT «SE A RESPOSTA ESTIVER ERRADA,
    QUER»:PRINT
270 PRINT «SABER A RESPOSTA CERTA?»:
    PRINT
280 INPUT «RESPONDA (YES) OU (NO)»;K$
290 REM*SE YES, O FLAG F PASSA A 1*
300 IF K$ = «YES» THEN F = 1:GOTO 340
310 IF K$ = «NO» THEN F = 0:GOTO 340
320 PRINT «RESPOSTA NÃO ACEITE»:GOTO 280
330 REM*APRESENTAR PERGUNTA E PEDIR
    RESPOSTA*
340 CLS:REM*LIMPAR VISOR*
350 FOR N = 1 TO M STEP 2
360 GOSUB 680
370 PRINT«QUAL É A CAPITAL DE»Q$(N):INPUT A$
380 GOSUB 680
390 IF A$ = Q$(N + 1) THEN S = S +1:GOTO 430
400 IF F = 0 THEN 430
410 IF F = 1 THEN PRINT «ERRADO»:PRINT
420 PRINT «A RESPOSTA CORRECTA É»Q$(N + 1)
430 NEXT N
440 REM*APRESENTAR PONTUAÇÃO E
    OBSERVAÇÕES*
450 CLS:REM*LIMPAR VISOR*
```

```
460 PRINT TAB(12)«RESULTADOS»
470 GOSUB 680:PRINT:PRINT
480 P = S*100/M:REM*PONTUAÇÃO
    PERCENTUAL*
490 W = (M/2) — S:REM*RESPOSTAS ERRADAS*
500 PRINT«RESPONDEU «S» CORRECTAMENTE»
510 GOSUB 680
520 PRINT «RESPONDEU «W» ERRADAMENTE»
530 GOSUB 680
540 PRINT«TEM UMA PONTUAÇÃO DE «P» POR
    CENTO»
550 GOSUB 680:PRINT:PRINT
560 IF P < = 40 THEN PRINT «NÃO FOI MUITO
    BOM, NÃO LHE PARECE?»:GOTO 600
570 IF P > = 41 ANO P < 60 THEN PRINT «NÃO
    FOI MAU»:GOTO 600
580 IF P > = 60 AND P < 80 THEN PRINT «BOM
    RESULTADO»:GOTO 600
590 PRINT «ABSOLUTAMENTE MAGNÍFICO»
600 PRINT:PRINT:PRINT
610 GOSUB 680
620 INPUT «QUER TENTAR DE NOVO?»;K$
630 IF K$ = «YES» THEN 100
640 IF K$ = «NO» THEN STOP
650 PRINT «RESPOSTA NÃO ACEITE»: GOTO 620
660 REM**********************************
670 REM*SUBROTINA.IMPRIMIR LINHA*
660 PRINT «.....»
690 REM*PARES DE PERGUNTAS E RESPOSTAS*
700 DATA FRANÇA, PARIS, INGLATERRA,
    LONDRES, DINAMARCA, COPENHAGA
710 DATA SUÉCIA, OSLO, UTOPIA, ZABALONIUM
```

Note o uso de uma subrotina que apenas imprime uma linha e portanto não requer qualquer preparação prévia para entrar GOSUB. Pode-se imprimir uma linha com qualquer das teclas «gráficas» existentes nos micro-computadores, se bem que uma escolha demasiado estranha

nada acrescente ao conteúdo informativo do visor. Uma simples tecla «-» é tão boa como qualquer outra e tem o mérito de manter o programa *portátil*, um termo que se refere a um programa que está escrito de tal modo que poderá provavelmente passar noutras máquinas. Para escrever programas portáteis deve-se pretender antes do mais evitar o uso das características específicas de uma máquina (por muito agradáveis que sejam) dado que não existem nas outras. As teclas gráficas dependem muito da máquina usada. De qualquer modo a maior parte das impressoras «standard» produzem resultados inestéticos quando alimentadas com teclas gráficas.

Instruções DATA e dimensão da memória

A memória dos computadores assemelha-se ao dinheiro. Nunca parece haver em quantidade suficiente. Como estará escrito provavelmente no manual da sua máquina, a memória de leitura e escrita é chamada RAM, e pode ser considerada como um conjunto de pequenas caixas capazes de armazenarem um *byte* cada. Um byte pode conter um caracter, e é composto por oito *bits*. O método normal de medir a dimensão da memória consiste em juntar 1024 bytes e chamar a este conjunto «1K». A maior parte das máquinas actualmente vendidas possuem pelo menos 8K, normalmente 16K ou 32K, e algumas 48K ou mais.

Para ter uma ideia dos gastos em termos de memória, note que o Programa 2 usou 1505 bytes (cerca de 1,25K) sem incluir as linhas DATA. Se considerarmos agora que cada elemento dos dados possui uma média de dez caracteres (dez bytes) e o Programa 2 for aumentado de modo a incluir 100 perguntas e 100 respostas, seremos forçados a consumir mais 20×100 bytes, isto é, cerca de 2K. O espaço total ocupado pelo programa e pelos dados será ainda inferior a 4K, desde que evidentemente não empregue nomes demasiado compridos.

Esboços de programas

Além dos items de DATA e das poucas linhas previamente mencionadas, a estrutura do Programa 2 pode ser usada como esboço e empregue repetidas vezes para uma vasta gama de temas. A parte «esboço» pode ser mantida separadamente em banda ou disco, sendo possível formar e armazenar uma biblioteca de perguntas.

O método de constituição da biblioteca deverá ser o seguinte:

1. Carregar a banda com o esboço do programa.
2. Escrever no teclado os pares de dados a partir da linha 700.
3. Modificar o título (linha 120), o texto que precede a pergunta (linha 370) e a constante «M» da linha 100 de acordo com o número de items de dados.
4. Armazenar o resultado novamente sob a forma de um programa completo.

É possível processar qualquer tipo de tema desde que se aceite um tratamento do tipo uma pergunta/uma resposta. Os temas podem ser reis/datas, países/moedas, elementos atómicos/números atómicos, palavras em português/inglês ou, para aqueles qque se interessem pelo assunto, estrelas «pop» e respectivas datas de nascimento.

Tabelas de consulta

As instruções DATA/READ são muito úteis para programar tabelas de consulta. Sempre que existe uma relação um a um entre duas coisas, é possível formar um quadro que mantenha a relação correcta entre as duas questões para qualquer uso futuro. Por exemplo, a Terça-feira é o terceiro dia da semana, pelo que podemos associá-la ao número 3. Setembro é o nono mês, pelo que o número 9 identifica-o perfeitamente.

Se os itèms de dados estão dispostos pela ordem sequencial apropriado, o subscrito do quadro servirá de identificação.

Para ilustrar esta ideia, estude o segmento de programa seguinte:

```
100  FOR N = 1 TO 12 STEP 1
110  READ M$(N)
120  NEXT
 .
 .
 .
 .
10000 DATA JANEIRO, FEVEREIRO, MARÇO, ABRIL,
      MAIO, JUNHO
10010 DATA JULHO, AGOSTO, SETEMBRO,
      OUTUBRO, NOVEMBRO
10020 DATA DEZEMBRO
```

Deve ser já evidente para o leitor que M$(5) conterá MAIO, ilustrando a relação correcta. Assim, no resto do programa será possível referenciar qualquer mês através do seu número «N».

Estas tabelas de consulta podem ser usadas para *códigos*. Um código especial, bem conhecido de todos os que estudaram o «hardware» dos computadores ou a programação em código-máquina, é o código *hexadecimal* usado para identificar as dezasseis combinações de quatro bits binários. Se os dados do exemplo anterior fossem alterados para 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F e o loop FOR para 16, seria fácil descodificar a base hexadecimal. C seria ligado a 12, F a 15, etc.

RESUMO

Os dados são definidos livremente como qualquer tipo de informação necessária a um programa.

Os dados podem ser fornecidos ao computador recorrendo às instruções INPUT ou GET, lendo-os a partir de banda ou de disco, ou utilizando instruções DATA/READ.

Usam-se memórias periféricas (memórias de apoio) para armazenar fichas de programas e fichas de dados.

Os items de dados semi-permanentes podem ser armazenados na instrução DATA, devendo ser separados entre si por uma vírgula.

Os diversos items dados podem ser colocados em variáveis utilizando a instrução READ numa ordem sequencial.

Os items dados, depois de utilizados por uma instrução READ, não podem ser lidos novamente a não ser que o «pointer» que os selecciona seja colocado a zero através de uma instrução RESTORE.

Os items de dados podem ser formados por misturas de valores numéricos e cadeias de caracteres, desde que sejam destinados a variáveis de equivalentes.

É possível usar a tecla CLR para limpar todas as variáveis existentes no programa, passando-as a zero.

Os items de dados são normalmente lidos para um quadro subscrito, numérico ou em cadeia de caracteres conforme for mais apropriado, recorrendo a um loop FOR.

Ao introduzir os items de dados é necessário dispô-los por uma ordem sequencial equivalente à ordem pela qual serão lidos mais tarde.

As funções DATA/READ são muito úteis para formar tabelas de consulta.

6

FUNÇÕES DE TRATAMENTO DE VARIÁVEIS DE CADEIA

A maior parte dos BASIC's oferecem uma vasta variedade de palavras-chave que permitem realizar algumas operações «cirúrgicas» sobre as variáveis de cadeia. É possível eliminar, dispor de modo diferente, inverter, trocar ou dividir caracteres isolados ou blocos inteiros dentro da cadeia. Como se sugere no título do capítulo, estas palavras-chave são todas funções, ou seja, possuem o formato XXX( ). Antes de passarmos aos detalhes de cada função, convém descobrir porque razão nos interessa tratar as palavras de um modo tão drástico.

Comecemos pelos nomes e iniciais. Se lhe pedirem que indique o nome você escreverá certamente J. S. Jardim, A. M. Prado ou M. Campinas. Dentro do computador será no entanto mais lógico, do ponto de vista da procura e recuperação posterior dos nomes, arquivar o apelido antes das iniciais. Outra utilização pode ser a redução do trabalho de input do operador. O Programa 2, que tratou do problema das perguntas e respostas, poderia ser modificado de modo a aceitar, por exemplo, as primeiras três letras da cidade em vez do seu nome completo. Este método permitiria não penalizar um estudante que, apesar de bom em geografia, fosse um tanto infeliz em ortografia. Os conhecimentos geográficos seriam suficientemente demonstrados se o aluno escrevesse EST em vez de Estocolmo. Quando mais tarde considerarmos as técnicas de tratamento de ficheiros veremos que forçar o operador a introduzir todo o nome de uma ficha pode impedir que esta seja encontrada na memória. O primeiro ou os dois

primeiros caracteres deveriam bastar para ter acesso a uma ficha, se bem que deva existir a opção de usar o nome completo.

Os jogos de palavras obrigam normalmente a um tratamento estranho dos caracteres, anagramas, etc. Outros campos de aplicação importantes são os códigos e a quebra de códigos. Este capítulo vai fornecer todos os pormenores sobre as funções de tratamento das cadeias de caracteres no formato vulgarmente empregue na maior parte das versões do BASIC usadas em microcomputadores.

Descobrir o comprimento de um string (cadeia de caracteres)

Comprimento designa aqui o número total de caracteres existentes na cadeia, incluindo os espaços. O formato é LEN(A$), com a variável de cadeia em causa dentro de parêntesis. Assim,

```
100  L = LEN(A$)
```

contará o número de caracteres existentes em A$ e colocará o resultado na variável numérica L.

Separação de caracteres à esquerda

O formato é agora LEFT$(A$,N), onde N é o número de caracteres da esquerda que devem ser separados. Por exemplo:

```
100  K$ = LEFT$(A$,3)
```

colocará os três primeiros caracteres do lado esquerdo de A$, em K$ (sem evidentemente corromper A$).

Antes de estudarmos as outras funções sobre variáveis de cadeia, vejamos algumas aplicações das já referidas.

Objectivo: modificar o programa 2A de modo a aceitar a resposta como correcta se os *dois primeiros caracteres* estiverem certos.

Substituir a linha 390 pelo seguinte:

```
390  IF LEFT$(A$,2) = LEFT$(Q$(N + 1),2) THEN
     S = S + 1:GOTO 430
```

Esta instrução tem uma aparência complicada, mas se a examinar cuidadosamente verificará que a variável Q$(N + 1) deve possuir o seu próprio par de parêntesis no interior dos parêntesis correspondentes à função LEFT$.

Objectivo: considerando uma variável de quadro T$(n) que contém 100 palavras, imprimir todas as que tenham quatro letras.

```
500  FOR N = 1 TO 100
510  IF LEN(T$(N)) = 4 THEN PRINT T$(N)
520  NEXT N
```

Objectivo: idêntico ao anterior, mas imprimindo todas as palavras com cinto letras que comecem pela letra S.

```
500  FOR N = 1 TO 100
510  IF LEN(T$(N)) = 5 AND LEFT$(T$(N),1)
     = «S» THEN PRINT T$(N)
520  NEXT N
```

Separação de caracteres à direita

O formato é agora RIGHT$(A$,N), onde N é uma vez mais o número de caracteres a separar. Assim:

```
100  K$ = RIGHT$(A$,3)
```

colocará os três caracteres mais à direita de A$ em K$.

Separação de caracteres do meio

O formato é MID$(A$,N,M). Permitirá separar M caracteres da cadeia A$, começando pelo enésimo caracter a a contar do início. Assim:

```
100  K$ = MID$(A$,3,2)
```

colocaria dois caracteres, começando pelo *terceiro,* em K$.

Esta função facilmente provoca erros, sendo habitual usar o formato do modo errado. Vamos portanto ver alguns exemplos que permitirão ilustrá-la melhor:

Suponhamos F$ = «LIVERPOOL CITY». A linha

```
K$ = MID$(F$,6,4)
```

colocaria POOL em K$; a linha

```
K$ = MID$(F$,4,3)
```

colocaria ERP em K$; a linha

```
K$ = MID$(F$,8,4)
```

colocaria OL C em K$; a linha

```
K$ = MID$(F$,1,1)
```

colocaria L em K$.

Junção de variáveis de cadeia (concatenação)

Se bem que esta técnica tivesse sido brevemente mencionada no capítulo 2, vamos tratá-la aqui a fim de dar continuidade à exposição das restantes funções de tratamento de variáveis de cadeia. O sinal + entre duas ou

mais cadeias ligará ambas *sem* qualquer espaço intermédio.

Considerando K$ = «NEW» e F$ = «NES»

A linha A$ = K$ + F$ colocará NEWNES em A$.

Para consolidar os nossos conhecimentos sobre as funções de tratamento de variáveis de cadeia, estudemos os seguintes módulos:

Objectivo: INPUT qualquer palavra ou frase e imprimir os caracteres separadamente no plano vertical.

```
100 INPUT«INDICAR UMA PALAVRA OU
    FRASE»;K$
110 L = LEN(K$):REM*DESCOBRIR NÚMERO DE
    CARACTERES*
120 FOR N = 1 TO L STEP 1
130 Q$ = MID$(K$,N,1):REM*SEPARAR UM
    CARACTER DE CADA VEZ*
140 PRINT Q$
150 NEXT N
```

Objectivo: INPUT qualquer palavra ou frase e imprimi-la em sentido inverso.

```
100 INPUT«INDICAR UMA PALAVRA OU
    FRASE»;K$
110 L = LEN(K$)
120 FOR N = L TO 1 STEP — 1:REM*CONTAR
    PARA TRÁS*
130 PRINT MID$(K$,N,1);:REM*PRIMEIRO O
    ÚLTIMO CARACTER*
140 NEXT N
```

Objectivo: considerando que uma variável de quadro A$(N) contém 20 códigos que representam um inventário, imprimir todos os códigos que possuem uma barra (/).

```
100 FOR N = 1 TO 20 STEP 1
110 L = LEN(A$(N))
120 FOR S = 1 TO L
```

```
130 IF MID$(A$(N),S,1) = «/» THEN PRINT A$(N)
    :S = L
140 NEXT
150 NEXT
```

O loop FOR exterior escolhe a palavra. O loop FOR interior observa os caracteres um por um até descobrir «/». Depois de o encontrar, não há interesse em investigar o resto da palavra, pelo que se usa a instrução S = L para terminar o Loop prematuramente. Isto é melhor do que abandonar o loop com GOTO.

Objectivo: INPUT um nome na forma: duas iniciais, espaço, apelido; depois imprimi-lo começando pelo apelido e colocando em seguida as iniciais.

```
100 PRINT«INDICAR O SEU NOME SOB A FORMA
    DE 2 INICIAIS»
110 PRINT«DEPOIS ESPAÇO E DEPOIS APELIDO»
120 INPUT K$
130 L = LEN(K$)
140 FOR N = 1 TO L STEP 1
150 I$ = LEFT$(K$,3):REM*INICIAIS E ESPAÇO
    EM I$*
160 J$ = RIGHT$(K$,L — 3):REM*APELIDO EM J$*
170 R$ = J$ + «» + I$:REM*CONCATENAR AS
    PARTES*
180 PRINT R$
```

Estes exemplos devem ser bastante úteis para a compreensão do modo como se usam as funções de tratamento das cadeias de caracteres; não se envergonhe porém se as considerar confusas. A maior parte das pessoas consideram-nas um obstáculo da linguagem BASIC, e necessitam de bastante prática para se habituarem a elas. A pior de todas é MID$, devido aos dois parâmetros da função; é fácil trocá-los. O melhor modo de consolidar o que aprendeu consiste em definir alguns objectivos próprios e tentar escrever o módulo. Comece por objectivos fáceis

mas aumente a dificuldade até atingir o impossível (a definição de impossível neste sentido é quando o nosso amigo mais bem informado fracassa...). Não se esqueça de que um computador pode fazer tudo aquilo que souber ensinar-lhe!

Existem mais algumas palavras-chave BASIC que convém aprender aqui apesar de não servirem explicitamente para tratamento de cadeias de caracteres, num sentido formal. Estão no entanto directamente relacionadas com este tratamento.

Conversão de variável de cadeia para variável numérica

É muitas vezes conveniente armazenar números numa variável de cadeia. Infelizmente, as operações aritméticas não podem ser realizadas com variáveis de cadeia porque estas são consideradas simples caracteres e não números. É no entanto possível transferir a cadeia de caracteres para variáveis numéricas usando a função VAL( ). VAL significa «valor», e bastará esta palavra para explicar o seu objectivo. Consideremos então:

```
100  K = VAL(A$)
```

Se A$ contém 34, esta linha produzirá K = 34.

Se o conteúdo de A$ não fosse numérico, o «valor» obtido seria zero (0). Assim, se A$ = «SETEMBRO», K seria 0. Se A$ for uma mistura de números e letras, VAL(A$) actuaria até ao primeiro valor não numérico. Mais alguns exemplos talvez ajudem o leitor:

Se A$ = «NEVOEIRO», K = 0. Se A$ = «4568», então K = 4568. Se A$ = «34 NEVOEIRO», K = 34. Se A$ = = «NEVOEIRO3468G», K = 0.

Note cuidadosamente o último exemplo; se bem que existissem vários algarismos em A$, o primeiro dado era o não numérico «F», pelo que o valor considerado foi zero.

Conversão de variável numérica a variável de cadeia

Consegue-se isto usando STR$( ), que é na prática a função oposta a VAL( ). Assim:

```
100  A$ = STR$(K)
```

converterá o conteúdo numérico de K para uma cadeia de caracteres, e colocará o resultado em A$. Por exemplo: se K = 34, A$ = «34».

Manipulações com o código ASC II

Quando se carrega numa tecla de um computador é produzido um determinado número de código, correspondente a essa tecla. Com uma única excepção, a maior parte dos computadores pequenos utilizam o «American Standard Code for Information Interchange», normalmente designado pelas suas iniciais, ASCII. Todas as teclas possuem um número de código ASCII. A maior parte dos manuais contêm uma lista dos códigos ASCII para cada caracter do teclado, se bem que possam empregar alguns caracteres não standardizados.

O código ASCII é publicado no final deste volume, mas é conveniente consultar o manual da sua própria máquina para conhecer as diferenças que possam existir. A função BASIC que nos indica o código ASCII é ASC( ). Por exemplo:

```
100  K = ASC(«S»)
```

colocará em K o código ASCII para o caracter S. Note que as aspas são aqui obrigatórias, a menos que o conteúdo dos parêntesis seja já uma variável de cadeia, como acontece no exemplo seguinte:

```
100  K = ASC(A$)
```

Qualquer que seja a cadeia de caracteres existente em A$, só o código ASCII do *primeiro* caracter é colocado em K.

O código ASCII completo contém letras maiúsculas e minúsculas, números, sinais de pontuação e caracteres de controlo. A memorização de todo este código não é necessária nem aconselhável, mas o leitor perderá menos tempo se pelo menos conhecer os códigos usados para os números e as letras. Estes códigos seguem uma ordem bastante simples, como se pode ver:

0 = 48	A = 65	a = 97
1 = 49	B = 66	b = 98
...	...	...
9 = 57	Z = 90	z = 122

Assim, desde que recordemos que os números começam em 48 e as letras em 65, será fácil descobrir o código de todos os caracteres do mesmo tipo. Assim, E = 69 e 5 = 53. As minúsculas nem sempre existem nos microcomputadores e, mesmo quando existem, é possível que não seja seguido o código ASCII normal.

Vejamos um módulo de programa que imprimirá os números de código ASCII e os correspondentes caracteres desde 48 a 90:

```
100 FOR N = 48 TO 90 STEP 1
110 PRINT N, ASC(N)
120 NEXT
```

Os números 0, ..., 9 virão primeiro, depois alguns sinais de pontuação, e finalmente as letras A a Z.

Existem diversas teclas que não possuem qualquer significado como caracteres e são designadas *códigos de controlo*. Assim, a tecla RETURN não está associada a qualquer caracter; a função que lhe cabe é «retorno e mudança de linha», sendo portanto classificada como *controlo*. As teclas STOP, SHIFT e RESET são mais alguns exemplos de teclas de controlo. O código ASCII para RETURN é 13.

Conversão do código ASCII para caracteres

Esta acção é o inverso da anterior, e é produzida pela função CHR$( ), que significa «cadeia de caracteres». Assim,

```
100  A$ = CHR$(N)
```

colocará em A$ o carácter cujo número de código ASCII é N. Nestas condições, se N = 65, A$ conterá «A».

Terminamos aqui as funções de tratamento das cadeias de caracteres. O programa que se segue ajudará a consolidar alguns dos pontos mais difíceis ou obscuros.

Objectivo: dar input a uma palavra ou frase e imprimir uma análise sob a forma: número de caracteres, número de vogais, número de espaços, número de algarismos.

```
100  CLS:V = 0:S = 0:T = 0:M = 0:REM* LIMPAR O
     VISOR E TOTALIZAR*
110  PRINT«INDICAR QUALQUER PALAVRA OU
     FRASE»
120  INPUT A$
130  L = LEN(A$):REM*DESCOBRIR
     COMPRIMENTO DA CADEIA*
140  FOR N = 1 TO L
150  Q$ = MID$(A$,N,1):REM*EXAMINAR UM
     CARACTER*
160  REM*VERIFICAR SE É VOGAL*
170  IF Q$ = «A» OR Q$ = «E» OR Q$ = «I» OR
     Q$ = «O» OR Q$ = «U» THEN V = V + 1
180  REM*VERIFICAR SE É ESPAÇO*
190  Q$ = «» THEN S = S + 1
200  REM*VERIFICAR SE É NÚMERO*
210  K = ASC(Q$):REM*CÓDIGO ASCII PARA
     CARACTER EM K*
```

```
220 IF K > = 48 AND K < = 57 THEN M = M + 1
230 REM*VERIFICAR SE É LETRA*
240 IF K > = 65 AND K < = 90 THEN T = T + 1
250 NEXT N:REM*PASSAR AO CARACTER
    SEGUINTE*
260 REM*IMPRIMIR ANÁLISE*
270 PRINT:PRINT
280 PRINT «.....»
290 PRINT «NÚMERO DE CARACTERES.....»
    L:PRINT
300 PRINT «NÚMERO DE VOGAIS.....»V:PRINT
310 PRINT «NÚMERO DE ESPAÇOS.....»S:PRINT
320 PRINT «NÚMERO DE ALGARISMOS»M:PRINT
330 PRINT «NÚMERO DE LETRAS.....»T:PRINT
340 PRINT «.....»:PRINT
350 INPUT «NOVAMENTE?»;K$
360 IF K$ = «YES» THEN 100
370 IF K$ = «NO» THEN STOP
380 PRINT «RESPOSTA NÃO ACEITE»:GOTO 350
```

Ordenação alfabética de nomes

A ordenação de números ou nomes é uma tarefa para a qual o computador está maravilhosamente preparado. Já se escreveu muito sobre o problema da ordenação, e consequentemente são muitas as técnicas empregues. O principal problema reside na velocidade de execução, porque escolher nomes numa lista de 500 e colocá-los por ordem pode demorar bastante tempo a menos que se utilizem métodos bastante habilidosos.

A forma como as cadeias de caracteres (nomes) são ordenados consiste em aproveitar a sequência dos números de código ASCII. O código de A é um número mais pequeno do que o de B, e portanto numa verificação condicional o computador considerará que A é menos do que B, R menos do que Z, etc.

A forma mais simples de realizar esta selecção segue os princípios seguintes:

1. Compara-se a primeira cadeia com a segunda, permutando-as me caso de necessidade. Coloca-se um «flag» se houver permuta.
2. Em seguida comparam-se a segunda e a terceira cadeias, continuando do mesmo modo até todos os pares terem sido processados.
3. Este tratamento é repetido até se obter uma passagem completa dos nomes sem que o flag de permuta tenha sido colocado. Neste momento a listagem pedida estará ordenada.

O programa que se segue ilustra este método.

Objectivo: INPUT vinte palavras e imprimi-las por ordem alfabética.

```
100  CLS:T = 20:REM*LIMPAR VISOR E
     INICIALIZAR T*
110  REM*INPUT VINTE PALAVRAS*
120  FOR N = 1 TO T STEP 1
130  PRINT«INDICAR NÚMERO DE PALAVRAS» N
140  INPUT A$(N)
150  NEXT
160  REM*ORDENAR*
170  F = 0:REM*LIMPAR FLAG TO 0*
180  FOR N = 1 TO T — 1 STEP 1
190  IF A$(N) < = A$(N + 1) THEN 240:REM*
     AUSÊNCIA DE PERGUNTA*
200  REM*ROTINA DE PERMUTA*
210  B$ = A$(N):REM*CÓPIA EM B$*
220  A$(N) = A$(N + 1)
230  A$(N + 1) = B$
240  NEXT N
250  IF F = 1 THEN 170:REM*REPETIR ATÉ F =0*
260  REM*IMPRIMIR A LISTA ORDENADA*
270  CLS
280  FOR N = 1 TO T STEP 1
```

```
290  PRINT A$(N)
300  NEXT N
400  STOP
```

Note que na linha 180 o loop FOR vai apenas até 19 porque a última comparação será A$(19) com A$(N + 1). Se bem que possamos considerar que o tempo necessário para ordenar vinte palavras não é mau, experimente alterar T para 100 na linha 100, o que resultará num pedido de 100 palavras. Dê entrada a estas (se tiver paciência), faça RUN do programa novamente e note o tempo que ele demora...

O problema deste tipo de programa de ordenação é o aumento exponencial no tempo de execução de cada vez que se aumenta o número de palavras a serem ordenadas. Existem muitos outros métodos de ordenação que a executam com maior rapidez, mas utilizam instruções que são difíceis de seguir e necessitam de bastantes páginas para serem explicadas convenientemente. Estes métodos podem ser encontrados em livros mais desenvolvidos; alguns aliás são dedicados apenas ao problema geral da ordenação.

O programa anterior foi escrito em termos de variáveis de cadeia e funciona bastante bem quando escolhe palavras. Se, no entanto, as «palavras» forem formadas por caracteres numéricos e tentarmos ordenar uma cadeia de números ocorre uma falha misteriosa. De facto, se considerarmos os dois números 45 e 356, o programa anterior considerará 45 como um número superior a 356; isto deve-se ao facto de os números, que se encontram no interior de variáveis de cadeia, serem tratados como caracteres individuais. Como o primeiro algarismo encontrado num dos números é 4 e o primeiro algarismo do outro é 3, o primeiro será considerado superior ao segundo.

Como poderemos resolver isto? Existem duas soluções possíveis. O programa pode ser novamente escrito, usando desta vez A(N) em vez de A$(N) passando as cadeias restantes para números. Alternativamente, pode-se

inserir a função VAL( ) na linha 190, que é a principal linha de comparação.

```
190 IF VAL(A$(N)) < = VAL(A$(N + 1)) THEN 240:
    REM*AUSÊNCIA DE PERMUTA*
```

Isto dará o resultado pretendido se a variável de quadro A$(N) contiver apenas números, mas desta vez será inútil para o tratamento de letras. A forma mais simples de sair disto consistiria em utilizar dois programas, um para os números e outro para as palavras. Na prática, seria improvável que os números e as letras fossem misturados excepto no tratamento de formas codificadas, caso em que os dígitos numéricos não têm qualquer significado em termos de grandeza.

RESUMO

Para descobrir o número de caracteres, incluindo espaços, existentes numa variável de cadeia, utiliza-se LEN(A$). Exemplo:

```
L = LEN(A$)
```

Para separar N caracteres à esquerda de uma cadeia usa-se LEFT$(A$,N). Exemplo: B$ = LEFT$(A$,3) separará os três caracteres mais à esquerda da variável A$, passando-os para B$.

Para separar caracteres à direita de uma cadeia utiliza-se RIGHT$(A$,N). Exemplo: B$ = RIGHT$(A$,2) colocará os dois caracteres à direita de A$ em B$.

Para separar M caracteres a partir do caracter N de uma cadeia usa-se MID$(A$,N,M). Exemplo: B$ = MID$(A$,3,1) colocará o caracter que se encontra na terceira posição de A$, em B$.

Concatenar significa juntar cadeias de caracteres recorrendo ao sinal +. Exemplo: C$ = A$ + B$.

Para mudar números, de uma variável de cadeia, para uma variável numérica, usa-se VAL(A$). Exemplo: K = = VAL(A$).

Para mudar números de uma variável numérica para variável de cadeia, usa-se STR$(K). Exemplo: A$=STR(K).

Para descobrir o número de código ASCII de um caracter utiliza-se a instrução ASC(A$). Exemplo: K = = ASC(«C»). Esta instrução colocará em K o código ASCII correspondente a C.

Se se utiliza ASC(A$) quando A$ possui vários caracteres, só o primeiro é convertido pelo programa.

Convém memorizar alguns códigos ASCII, como os dos números e os das letras. Assim, recorda-se que a letra A corresponde a 65, sendo as letras seguintes designadas ordenadamente pelos números que se seguem. Os números começam por «0», a que corresponde o número de código 48. As letras minúsculas partem do número 97, que representa a.

É fácil escolher cadeias de caracteres ou números e ordená-las, porque todos os caracteres representados no teclado correspondem a números de código ASCII que estão ordenados sequencialmente.

Para descobrir um caracter a partir do número de código ASCII, utiliza-se a instrução CHR$(N). Exemplo: A$ = CHR$(N) colocará em A$ o caracter cujo número de código ASCII é N.

7

PALAVRAS-CHAVE DIVERSAS

A principal função de um interpretador BASIC consiste em poupar ao utilizador as duras realidades da linguagem da máquina. O leitor recordará certamente do capítulo 1 que o código-máquina (ou a «linguagem» desta) é difícil e requer um grande conhecimento do seu hardware e bastante prática. Este conhecimento não terá necessariamente de incluir electrónica, se bem que seja vantajoso possuir algumas ideias sobre o assunto. Estaria deslocado neste livro ocupar muitas páginas com o hardware dos computadores, mas existem algumas palavras-chave BASIC que ocupam uma área intermédia, formando uma ponte entre a linguagem de alto nível e a linguagem da máquina. Para utilizar estas palavras-chave híbridas com alguma confiança e evitar que a máquina se recuse a trabalhar, é conveniente possuir pelo menos um conhecimento relativo da anatomia da máquina. Para aqueles que se interessam por estudar este tema em detalhe, convirá consultarem outras obras; o que diremos em seguida não «partirá o gelo» se bem que nos permita «mordê-lo».

Bits binários

O «binário» é um sistema de contagem que utiliza apenas dois algarismos, o «0» e o «1», os quais podem ser representados electricamente por duas faixas de tensão diferentes, uma «baixa» e outra «alta». A baixa tensão

varia normalmente entre 0 e 0,4 volts, e a alta entre 2,4 e 5,0 volts. Um número expresso em binário consiste numa cadeia de uns e zeros e, tal como acontece na notação decimal vulgar, o valor de cada dígito depende da posição em que se encontra nessa cadeia. Em binário, passa-se à ordem superior em potências de dois e não em potências de dez. Cada dígito binário é designado por *bit;* nestas condições, um bit pode ter o valor 0 ou 1. Um conjunto de oito bits é designado por *byte.*

Mas existem ainda alguns outros termos de gíria que penetram lentamente na nossa linguagem. Quatro bits é muitas vezes designado por «nibble» e dezasseis bits foi recentemente chamado «gulp». Ninguém se lembrou ainda de arranjar um nome para trinta e dois bits, mas se se mantiver esta obcessão pelo nosso aparelho digestivo é muito possível que venha a chamar-se «belch» (arroto)...

Os computadores pequenos utilizam uma pastilha de silício (chip) a que se chama *microprocessador* em termos de controlo central. A responsabilidade de memorização das nossas ordens recai sobre outros chips a que se chama RAM's. A tarefa de interpretar as ordens e realizar funções de vigilância foi investida nos chips ROM. Existe uma grande diferença entre RAM e ROM, que não é porém evidente nos seus nomes. RAM significa «memória de acesso aleatório», o que significa que se pode depositar ou retirar informação livremente, sem preocupações de ordem. O nome, no entanto não mostra imediatamente as duas grandes diferenças que a demarcam da ROM.

(*a*) Uma memória RAM pode ser usada para escrever informações novas ou ler as já existentes. As RAM's são *voláteis* no que diz respeito à fonte de alimentação. Se o computador for desligado, as informações anteriores «evaporam-se».

(*b*) ROM significa «memória somente para leitura», e a informação previamente gravada que contém nunca pode ser alterada pela máquina ou pelo operador. As ROM's não são *voláteis* e a informação que contêm é independente da fonte de alimentação.

Em geral, os programas e as manipulações do teclado são dirigidos à RAM, mas a informação ROM permite-nos fazê-lo com uma relativa facilidade, evitando que sejamos obrigados a mexer no interior da máquina.

Comprimento de palavra

A maior parte dos microprocessadores normais tratam as informações apenas em blocos de oito bits de cada vez. Diz-se assim que o *comprimento de palavra* é de oito bits, ou seja, de um byte. A fim de se adaptarem ao esquema definido pelo microprocessador, as RAM's e as ROM's devem também ser compostas de células de memória capazes de armazenarem um byte cada. É o comprimento de palavra que distingue essencialmente o computador gigante da versão «micro», porque se tudo o resto for igual, quanto maior for o comprimento da palavra maior a quantidade de informação processada de cada vez.

Examinemos agora as limitações impostas por este comprimento de um único byte, verificando quanta informação numérica e textual pode ser incluída numa única palavra. O maior número binário que se pode escrever num byte é 11111111, o que equivale a 255 em decimal. Para compreender porquê, recordemos que o bit da direita (chamado bit menos significativo ou «lsb») vale 1, o seguinte 2, o terceiro 4, depois 8, 16, 32, 64 e finalmente o último (primeiro da esquerda chamado bit mais significativo ou «msb») 128. Se somarmos estes números veremos que equivalem a 255. Existe um modo simples de descobrir o binário puro máximo numa palavra de N bits:

$$\text{Valor máximo} = 2^N - 1$$

Assim, numa palavra de oito bits, o número máximo é $2^8 - 1 = 256 - 1 = 255$, na base decimal.

A reduzida quantidade de números que podem ser tratados por um byte é obviamente uma grande desvantagem, e de facto é ainda pior do que já se sugeriu. O valor binário referido é «puro», sugerindo portanto que existem outras formas de representação binária. Com efeito existe uma, a que se chama «complemento de dois» porque permite a representação do sinal do número numa forma aritmética conhecida como tal. Nesta forma, o «msb» é designado *bit de sinal*, considerando-se por convenção que 0 indica que o número é positivo e 1 que este é negativo. Assim, 01111111 é o maior número positivo existente nesta notação, correspondente apenas a + 127 na base decimal.

Os números negativos são difíceis de trabalhar porque se tornam bastante confusos; as regras a aplicar são as seguintes:

Complementar os bits e em seguida somar 1.
Ignorar qualquer transporte no final da operação.
Complementar um número em binário significa substituir os 1's por 0's e vice-versa.
Exemplos:

00000111 = + 7	11111001 = — 7
00000001 = + 1	11111111 = — 1

Como tudo isto nos parece estúpido, sentimos a tentações de culpar os malucos dos computadores, acusando-os de serem complicados sem necessidade. Existe no entanto uma razão para tudo isto, devida ao hardware electrónico. É muito complicado conceber circuitos electrónicos que possam realizar adições binárias, e portanto para evitar a tarefa igualmente ingrata de conceber um circuito «subtractor» recorre-se ao truque de somar normalmente... um número negativo. Representando os negativos na forma «complemento de dois», obtém-se o mesmo resultado e não se tem mais problemas com a electrónica. O princípio básico do código-máquina é simples: se é mais fácil do ponto de vista da máquina, o ser humano tem de se adaptar! Em defesa do código-

máquina, no entanto, convirá referir que um rograma escrito neste código (ou em linguagem assembly, que é praticamente a mesma coisa) é bastante eficaz e funciona a uma velocidade fantástica quando comparada à do mesmo programa escrito em BASIC. Com efeito, o princípio básico do BASIC é: se é mais simples para o ser humano, a máquina que se adapte...!

Organização da memória

Devido ao reduzido comprimento da palavra, uma das tarefas realizadas pelo intérprete BASIC consiste em atribuir um certo número de bytes a cada número. Normalmente serão reservados cinco bytes para cada número de vírgula flutuante. É desagradável pensar nisto quando se escreve uma afirmação simples como 100 A = 5. Se bem que o número cinco pudesse caber facilmente num único byte, é ainda necessário reservar cinco bytes para o caso da variável «A» acabar por ter, algumas linhas mais abaixo, o bonito valor de 4.678247E30.

As variáveis inteiras, por outro lado, são normalmente restringidas a um número limite muito mais reduzido, em geral com apenas dois bytes, o que na forma «complemento de dois» lhe permite atingir um máximo de $2^{15} - 1 = 32767$, em decimal. No caso de alguns BASIC's as variáveis inteiras reservam o mesmo número de bytes que as de vírgula flutuante, apesar de se manter a restrição a 32767, pelo que é inútil nestes casos usar variáveis inteiras para obter alguma economia de memória. O método usado em cada máquina poderá, com sorte, ser descoberto no respectivo manual, procurando nas linhas escritas com letras mais pequenas...

Leis de precedência em bytes duplos

Quando mais tarde tratarmos das palavras-chave PEEK e POKE, será muitas vezes necessário alterar o conteúdo de certos bytes da memória. Se a informação a

mudar estiver contida num único byte, não haverá problemas. Estes surjem quando a informação está contida em dois bytes adjacentes da memória e é tratada como um número. Todos os bytes da memória ocupam endereços-máquina, estando cada célula da memória associada a um endereço único. Em código-máquina, este endereço será codificado num sistema conhecido por hexadecimal, mas felizmente em BASIC este foi modificado para decimal. Infelizmente não existe qualquer regra fixa, para além da que se pode encontrar no manual da máquina, quanto ao byte de um byte duplo que deve ser considerado «de menor ordem« ou «de maior ordem». Para explicar isto, imaginemos um número de dois bytes armazenado numa única palavra de 16 bits. Suponhamos que se trata de:

00000011 00000011

O byte da esquerda é conhecido por byte de maior ordem, e o da direita como o de menor ordem. Considerando-os por agora como bytes individuais, o da esquerda contém 3, e o da direita o mesmo valor. Se no entanto forem considerados como um único byte duplo, o byte de ordem superior valerá 256 vezes mais do que o byte de ordem inferior devido ao valor atribuídos à posição. O valor total do duplo byte é portanto $(256 \times 3) + 3$, isto é, 771.

Acontece que algumas máquinas armazenam primeiro o byte de maior ordem e em seguida o de menor ordem no endereço seguinte, enquanto outras procedem de modo contrário.

Exemplo:

Endereço	*Conteúdo em decimal*
781	5
782	4

Se o conteúdo do endereço 781 é considerado como byte de menor ordem, a palavra de duplo byte armazena $5 + (256 \times 4) = 1029$. Se o conteúdo de 781 for considerado

como byte de maior ordem, a palavra será $(5 \times 256) + 4 = 1284$.

Como se disse anteriormente, a convenção usada depende daquilo que é decretado no manual.

Endereços RAM especiais

A maior parte da RAM pode ser usada livremente pelo programador, mas em muitas máquinas uma pequena proporção da RAM é utilizada pelo sistema residente. Se bem que o programa que leva o microprocessador a comportar-se como um computador se encontre «enterrado» em ROM, necessita de algum espaço para «respirar» que lhe permita realizar cálculos intermédios. Isto só pode ser feito com o auxílio de memórias de leitura (read) escrita (write), pelo que é habitual utilizar parte da RAM. Nesta área são armazenadas «coisas» como indicadores (pointers) de dados, a fim de permitir a execução correcta das instruções DATA e READ. Já foi mencionada a propósito dos loops FOR e das subrotinas uma área da memória designada por «stack». A sua localização é feita na área especial de RAM. Muitos outros indicadores e números de referência são igualmente armazenados nela, e qualquer interferência «não autorizada» por parte dos programas em BASIC escritos por nós pode conduzir a sintomas esquizofrénicos no sistema. A localização desta parte reservada de endereços de memória RAM depende também da máquina. A maior parte dos manuais incluem uma lista detalhada dos pointers RAM e dos seus endereços específicos.

Uma outra área da RAM com funções especializadas é conhecida por *screen RAM*. A maior parte dos microcomputadores utilizam aquilo a que se chama sistemas de visor «memory-mapped». Os caracteres apresentados no visor são armazenados em forma codificada na «screen RAM», e automaticamente inspeccionados cerca de cinquenta ou mais vezes por segundo a fim de produzirem uma apresentação suficientemente fixa. Suponhamos que

o seu computador apresenta no visor 25 linhas de 40 caracteres cada, num total de 1000 caracteres por imagem. Mesmo que aparentemente só esteja a ser apresentado um único caracter, por exemplo um «A», continua a ser necessário «pintar» 999 espaços. Um espaço, de facto, é também um caracter com um código ASCII próprio. Nestes casos, são reservados 1000 endereços RAM para armazenar o equivalente a um visor cheio de caracteres. Alterar um destes tem como consequência imediata o aparecimento de um caracter diferente no visor numa posição proporcional ao endereço em que se encontra na screen RAM.

Para ilustrar esta ideia, suponhamos que a screen RAM ocupa os endereços 20000 a 20999 inclusive. O conteúdo do endereço 20000 é interpretado como um código de caracter, e este caracter aparecerá na *primeira* posição de impressão do visor, na linha superior à esquerda. A posição de impressão seguinte corresponderá ao código presente no endereço 20001, etc. O endereço 20999 contém o código destinado ao caracter da última linha à direita.

Terminamos assim o resumido estudo necessário para compreender o modo corrector de usar as instruções POKE E PEEK. Seria prudente recordar àqueles, que possam ter ficado confusos com alguns destes pormenores sobre a construção da máquina, que pouco perderão se decidirem ignorá-los e não utilizarem as duas instruções referidas. Uma das utilidades destas consiste em pôr coisas a mexer no visor; infelizmente, este movimento não é normalmente regular, fazendo recordar os antigos filmes mudos.

A instrução POKE

O seu formato é simples, de facto até excessivamente simples:

POKE endereço, N

Esta instrução colocará o número N num endereço-máquina. Por exemplo:

```
100  POKE 2345,7
```

colocará o número 7 no endereço-máquina decimal 2345.

O número que se encontrava anteriormente neste endereço é destruído e substituído pelo novo número. Considerando o habitual comprimento de palavra, 8 bits, sabemos que 7 é de facto a cadeia binária 00000111. Do que já se disse deveria ser óbvio que tentar «poke» um número superior a 255, em decimal, provocará uma mensagem de erro indicando a «quantidade ilegal» indicada na linha 100.

A instrução PEEK

Ao contrário da POKE, esta instrução é de facto uma função, pelo que não é de admirar a presença do habitual par de parêntesis. O seu formato é:

PEEK (endereço)

Por exemplo:

```
100  K = PEEK(346)
```

colocará em K uma cópia do que se encontra no endereço-máquina 346. O conteúdo deste endereço não é alterado. Ao contrário da POKE, que pode provocar o caos, a PEEK é bastante benigna.

Utilidade da PEEK e da POKE

Existem três áreas em que estas duas instruções se revelam particularmente úteis:

Imagem no visor. Se actuarmos através de uma instrução POKE sobre a área designada «memória de visor»,

o efeito obtido será a alteração da função PRINT normal. Quando escrevemos PRINT S, a variável é impressa na posição do visor definido pelo *cursor*. Usando POKE, o cursor é ignorado porque a impressão é agora definida no interior da própria instrução POKE. Se já se encontra um caracter na posição «poked», este é eliminado sem cerimónias em favor do intruso. O caracter de facto apresentado depende do «código poke», que deveria idealmente ser o ASCII standard. Infelizmente, algumas máquinas têm de jogar com o código ASCII a fim de obterem os códigos necessários para caracteres artificiais normalmente designados por «gráficos». Estes caracteres incluem riscos, manchas e pequenas curvas (que raramente parecem estar adaptados uns aos outros), usados para fins «cosméticos» ou em jogos.

Alteração do sistema. Já referimos anteriormente o facto de uma determinada área da RAM ser pertença exclusiva do sistema, que a utiliza para armazenar pointers. Esta área é definida no manual, e na maior parte dos casos encontrar-se-á na extremidade «inferior» da memória, isto é, no endereço 0. A quantidade de RAM utilizada pelo sistema varia, mas é normalmente de 1K (1024 células de memória). É habitual dividir esta área nas chamadas «páginas», sendo a parte inferior da memória designada por «página zero», a seguinte «página um», etc. As dimensões de uma página não estão infelizmente standardizadas, se bem que se tenta actualmente a atribuir-lhe 256 células. Assim, a área correspondente à página zero estender-se-ia do endereço 0 ao endereço 255, a página um do endereço 256 ao 511, a página dois do endereço 512 ao 767, etc.

Estudando o manual, é possível modificar parte dos limites impostos pelo sistema residente. Por exemplo, pode ser possível diminuir os limites de memória para os nossos programas BASIC a fim de libertar algum espaço de endereço para subrotinas escritas em código-máquina. Seria possível alterar a posição normal do cursor; inibir a tecla STOP ou até modificar o comportamento normal das instruções BASIC. Já deve ser evidente para o leitor

que o uso indiscriminado da POKE nesta área da memória é, no mínimo, arriscado. Uma POKE mal sucedida não pode provocar «danos» no computador; felizmente o pior que pode acontecer é o «estoiro» a que já nos referimos. Se tiver muita paciência, mesmo este «estoiro» pode ser tolerado. Basta-lhe desligar a máquina (ou voltar ao princípio se existir um botão RESET em algum sítio) e voltar a introduzir todo o seu programa. Depois de várias repetições aprenderá a tratar a instrução POKE com o respeito que merece.

Uso do «port» de saída. Num ponto nebuloso do interior da maior parte dos computadores encontra-se uma «tomada» com vários pernes designada por «port» de saída, ou algo de semelhante. Esta serve para enviar ou receber sinais de certos dispositivos especializados que talvez você mesmo possa construir. As ligações a este «port» («porto») só podem ser conseguidas quando se possuem os necessários conhecimentos de hardware. Para utilizar o «port» depois de ter realizado tais ligações, pode-se usar a instrução PEEK para examinar os sinais de entrada e a POKE para examinar os de saída. Estamos a considerar aqui que o port de saída é, tal como o visor, «memory mapped», e ainda que você conhece o seu endereço-máquina. Certos ports de saída são muito sofisticados e permitem tantas permutações de funcionamento que o uso de um deles se pode revelar uma tarefa estarrecedora, que apenas poderá ser enfrentada pelos mais corajosos. É uma pena que a sofisticação e a complexidade durmam normalmente na mesma cama.

Quando se pretende activar um dos dispositivos externos, deve-se escolher o «número» enviado ao port de saída tendo em mente um determinado perne. Esta é outra área onde a habilidade na manipulação dos bits binários é essencial. Assim, se o dispositivo necessita de um «1» para ser activado e for ligado ao *terceiro* perne da direita de um port de oito bits, o «número» enviado ao port deve ser um decimal que tenha um «1» nesta posição do equivalente binário. Por outro lado, o número deve

ser tal que não altere quaisquer outros dispositivos ligados. Pense nisto... talvez seja difícil de compreender!

Módulos de programas usando PEEK e POKE

Das observações anteriores relativas à natureza das instruções PEEK e POKE, orientadas para a máquina, conclui-se que para apresentar alguns exemplos teremos de partir de alguns pressupostos.

Consideremos então que a memória do visor vai de 32768 a 33767, o que equivale exactamente a 1000 endereços. Consideremos ainda que existem 25 linhas no visor, e que cada uma tem um comprimento de 40 caracteres. Consideramos em último lugar que os códigos de caracteres POKE são iguais aos ASCII.

Objectivo: permitir ao operador colocar qualquer caracter em qualquer posição do visor usando a instrução POKE em vez de PRINT.

```
100  S = 32678:REM*POSIÇÃO NO CANTO
     SUPERIOR ESQUERDO DO VISOR*
110  L = 40:REM*CARACTERES POR LINHA*
120  INPUT «QUE CÓDIGO DE CARACTER?»; C
130  INPUT «QUANTAS LINHAS ABAIXO?»;X
140  INPUT «QUANTAS COLUNAS À FRENTE?»;Y
150  POKE S + Y*L + X,C
160  GOTO 100
```

As constantes S e L podem ser alteradas de acordo com os «memory maps» de cada máquina. A GOTO na linha 160 é útil para experimentar diferentes posições e caracteres POKE. Note que qualquer caracter, que se encontre no visor e se «atravesse no caminho» da instrução, é sem cerimónia substituído pelo caracter «poked».

Objectivo: movimentar uma «bola» no visor.

```
100  CLS:REM*LIMPAR VISOR*
110  S = 32678:Y = 10:L = 40:F = 30:D = 100
```

```
120 C = 79:REM*CÓDIGO ASCII PARA A LINHA
    '0'*
130 FOR X = 1 TO F STEP 1
140 POKE S + Y*L + X,C:REM*POKE
    CARACTER*
150 FOR Z = 1 TO D:NEXT:REM*ATRASO*
160 POKE S + Y*L + X,32:REM*POKE UM
    ESPAÇO.ASCII(32)*
170 NEXT
```

A «bola» é colocada na posição X da linha 140 e guardada por um tempo determinado pelo loop de atraso na linha 150. É depois extinta pela linha 160, que apresenta um espaço em vez da bola. A execução seguinte do loop FOR aumenta X na linha 130, e portanto a bola parece mover-se. Como este módulo forma o esboço de uma gama de programas que servem para movimentar coisas no visor, convém investigar o efeito das mudanças de parâmetros:

1. O valor de Y determina a coordenada Y na qual se dá o movimento.
2. O valor de V determina o comprimento do movimento X.
3. O valor de D determina a velocidade da bola; um D pequeno produz um movimento rápido.
4. O valor de C determina o caracter que se move (em forma ASCII).

Objectivo: imprimir uma margem rectangular.

```
100 CLS:REM*LIMPAR VISOR*
110 S = 32678:L = 40:C = ASC(«*»)
120 W = 39:H = 22:REM*LARGURA E ALTURA*
130 FOR X = 0 TO W
140 POKE S + X,C:REM*MARGEM SUPERIOR*
150 POKE S + H*L + X,C:REM*MARGEM
    INFERIOR*
160 NEXT
170 FOR Y = 0 TO H
```

```
180  POKE  S + L*Y,C:REM*MARGEM  ESQUERDA*
190  POKE  S + L*Y,C:REM*MARGEM DIREITA*
200  NEXT
```

Através de ajustamentos apropriados das constantes de largura e altura, a margem pode ocupar qualquer área que se pretenda no visor. Pode dar um «toque» artístico a uma página desde que o texto seja posicionado de modo a respeitar a margem. Acrescentando RETURN, o módulo transforma-se numa subrotina útil para jogos de bola. Bastarão mais algumas linhas para produzir um jogo de ténis ou futebol.

É fácil programar um «toque» com algum objecto introduzindo uma PEEK antes de cada movimento da bola para verificar se o caracter na posição seguinte é ou não um espaço.

Os operadores lógicos AND e OR

Existem duas situações em que se utilizam estas duas palavras-chave BASIC. A primeira já a encontrámos ao discutir a instrução IF/THEN. A segunda envolve algum conhecimento das funções dos operadores Booleanos AND e OR. São usados principalmente para modificar *bits escolhidos* no interior de uma palavra, e não pela toda.

AND é usada para limpar bits escolhidos, passando-os ao valor 0. Suponhamos que se encontra em X um número (considerado como uma cadeia de bits) e que alguns bits devem ser limpos a zero sem perturbar os restantes. O método consiste em aplicar através de AND uma «máscara» consistindo em zeros nos pontos onde interessa limpar os bites de X, e em 'uns' nos outros. A regra é simples: só quando ambos os bits correspondentes são 1's continuará a existir um «1» quando a operação estiver terminada. Por exemplo:

Máscara	10000001
X antes	10111001
X depois	10000001

O exame do exemplo dado pode conduzir à impressão falsa de que o resultado presente em X é igual à própria máscara. Isto nem sempre é verdade, como se mostra no exemplo seguinte:

Máscara	01011111
X antes	10111001
X depois	00011001

O módulo seguinte permitir-nos-á experimentar diferentes valores de X e da máscara.

```
100  INPUT«INDICAR VALOR DE X»;X
110  INPUT«INDICAR NÚMERO-MÁSCARA»;M
120  X = X AND M
130  PRINT «O VALOR DE X É AGORA»X
140  GOTO 100
```

Quando se experimentam diversos valores pode-se ficar admirado com os resultados porque não parece existir qualquer sentido aritmético nas respostas. Não se preocupe, porque não há qualquer razão para existir esse sentido! O operador AND não é aritmético. O facto de o X e os valores de máscara serem indicados sob a forma de números decimais é uma simples coincidência; a única maneira de compreender esta operação consiste em pegar num lápis e numa folha de papel, escrever o padrão de bits usado como máscara e em seguida converter para decimal a fim de obter o «número».

Para que serve a AND? A sua principal utilidade reside na conversação (se é que esta pode ser considerada a palavra exacta) com os dispositivos externos ligados ao port de saída. Pode-se comutar qualquer deles para 0 sem perturbar os restantes. Se o dispositivo já se encontra em 0 nada acontece; interessa-nos apenas *assegurar* que seja 0 seu valor.

A operação OR tem uma função oposta à AND. Sempre que aparece um 1 na máscara, o correspondente bit em X é igualado a 1. Sempre que se encontra um 0 na

máscara, os bits correspondentes são deixados como estão. Por exemplo:

Máscara	10000100
X antes	01110000
X depois	11110100

O operador OR pode ser usado para comutar dispositivos externos para 1 sem perturbar os restantes.

Gráficos

O termo «gráficos» possui vários significados além da versão óbvia do dicionário. Tudo o que apareça no visor é evidentemente uma imagem *gráfica* no seu sentido mais lato, mas em gíria de computador a palavra tem dois sentidos específicos:

Teclas gráficas. Estas teclas já foram mencionadas anteriormente e são típicas dos microcomputadores concebidos para uso pessoal, em particular para a realização de jogos e de imagens diversas. O pormenor obtido é fraco porque cada risco ou mancha ocupa toda a área de um caracter. A realização de uma aritmética simples sobre o número de caracteres por linha e o número de linhas por visor revelará a pobreza da resolução obtida. É como tentar pintar com uma escova de dentes!

Gráficos de alta resolução. Para trabalho artístico, gráficos, desenhos de precisão e jogos de alta qualidade, os microcomputadores oferecem por vezes caracteres gráficos de «alta resolução». O visor pode ser considerado dividido em coordenadas x e y, dependendo o número de cada da quantidade de memória que podemos gastar. Assim, o programador pode escolher, por exemplo 200 por 150, 350 por 250, e em certos casos até 600 por 320 pontos diferentes... uma resolução quase tão elevada como a usada em televisão.

Para usar imagens de alta resolução é necessário acrescentar certas palavras-chave ao vocabulário BASIC,

como por exemplo PLOT x,y; DRAW x1,x2; DRAW y1,y2, etc. Se a máquina nos permite utilizar a cor, existirá ainda uma palavra chave extra, COLOUR, N. N é um número que define uma das cores disponíveis, normalmente entre quatro e dezasseis.

O uso de gráficos de alta resolução não se resume a tocar nas teclas correspondentes, mas com a prática e recorrendo aos auxiliares de programação disponíveis pode ter a certeza de que o esforço dispendido será devidamente recompensado. De facto, abre-se assim todo um mundo novo em termos de programação para aqueles que souberem perseverar.

Não tentaremos discutir aqui os gráficos de alta resolução porque o BASIC não está suficientemente estandardizado neste campo.

Números aleatórios

O BASIC inclui uma palavra-chave (não muito estandardizada) que produz um número pseudo-aleatório muito útil em jogos de sorte e em estatísticas. Nas revistas da especialidade surgem regularmente artigos pontificando sobre o carácter não-aleatório deste número. Podem-se usar vários truques para impedir a repetição de uma mesma sequência de números, mas na maior parte dos casos não é de facto importante evitá-la em absoluto.

Um dos formatos é RND(X), onde X é qualquer número positivo, pelo que a linha:

```
100  R = RND(1)
```

colocará um número aleatónio em R.

Relógios

Normalmente existe um «relógio» nos microcomputadores, que pode ser usado para obter atrasos rigorosos

(desde que não sejam demasiado pequenos). Pode-se aceder ao relógio através de uma palavra-chave, por exemplo TI, começando a contagem a partir do momento de ligação. Consulte o seu manual se quiser mais pormenores.

A função definida pelo utilizador

As funções standard de origem, como SIN, COS, TAN, EXP, são de facto muito úteis, mas nenhum conjunto de funções pode evidentemente satisfazer todos os gostos. Para resolver este problema, o BASIC fornece ao utilizador um meio de definir uma função que possa depois ser usada bastantes vezes num mesmo programa. Existem duas partes diferentes na sua estrutura; a primeira *define* a função, e a segunda *utiliza-a.*

O forato de definição é:

```
DEF FN A (X) = função de X
```

onde DEF FN é a palavra-chave que significa definir função, A é o nome da função, que pode ser qualquer variável numérica, e X é *dummy variable* (variável fictícia). A função pode ser qualquer expressão legítima. Por exemplo:

```
100  DEF FN K (V) = V + SIN(C)
110  DEF FN E (Y) = EXP(Y) — EXP(— Y)
```

O formato para uso da função definida é:

```
FN A (J)
```

onde FN é a palavra-chave, A é o nome da função que apareceu na parte DEF, e J é a variável ou constante que substitui a variável fictícia. Por exemplo:

```
500 S = FN K (T)
510 D = FN S (P)
```

Os formatos não são fáceis de memorizar, sendo necessário admitir que esta estrutura não é usada tão frequentemente como seria de esperar dada a sua utilidade.

Para ilustrar o uso da estrutura DEF FN, estude os módulos seguintes:

Objectivo: imprimir uma tabela de X e SIN(X) — 0,5 para valores de X 1, 2, 3, ..., 20.

```
100 CLS:REM*LIMPAR VISOR*
110 DEF FN S(X) = SIN(X) — 0.5
120 FOR X = 1 TO 20 STEP 1
130 PRINT X, FN S (X)
140 NEXT X
```

Objectivo imprimir uma tabela como a anterior, mas usando a função SQR(R ↑ 2 + X ↑ 2).

```
100 CLS:R = 10
110 DEF FN Z(X) = SQR(R ↑ 2 + X ↑ 2)
120 FOR X = 1 TO 20
130 PRINT X, FN Z (X)
140 NEXT X
```

Se bem que a variável dos exemplos anteriores (X) fosse igual à variável fictícia, não existe qualquer regra que afirme que isto deva acontecer. Desde que o nome da função seja o mesmo quando é chamada e quando é definida inicialmente, a variável pode ser alterada. Assim:

```
100 DEF FN F1(V) = 3*V + V/56
110 K = FN F1(P)
```

colocará 3*P + P/56 em K, apesar de a definição inicial ter sido feita em termos de V. O nome da função é F1.

O uso da estrutura DEF FN é limitado na maior parte dos BASIC's a uma linha do computador. Normalmente é imposta outra restrição: impossibilidade de usar variáveis de cadeia! Qual é então a sua utilidade? Esta

questão é pertinente porque, olhando para os exemplos anteriores que imprimem tabelas da função, dir-se-ia que a função poderia ter sido programada directamente no interior de um loop FOR sem ser definada previamente. No entanto, depois de definida, a função pode ser usada muitas vezes no mesmo programa, chamando-a apenas «pelo nome» a qualquer momento, usando FN. Podemos considerá-la uma espécie de «subrotina» de atalho, onde em vez de se saltar para a subrotina ou sair dela recorrendo ao uso de números de linha, pode ser chamada de qualquer ponto do programa usando FN em vez de GOSUB. Uma vantagem desta função relativamente às subrotinas consiste na possibilidade de *substituir a variável* quando a função é chamada. Com subrotinas, é muitas vezes necessário mudar as variáveis antes de as chamar, a fim de as «adaptar» às variáveis escolhidas quando se concebeu a subrotina. Usando FN, tudo isto se torna desnecessário.

Vamos escrever uma função definida pelo utilizador que sirva para arredondar qualquer número para, digamos, duas casas decimais:

```
100  DEF FN R (X) = INT(X*100 + 0.5)/100
```

Na linha 300 poderemos querer arredondar K para duas casas decimais, e portanto usamos:

```
300  K = FN R (K)
```

Mais abaixo, na linha 450, queremos arredondar G:

```
450  G= FN R (G)
```

Se a função tivesse sido escrita como uma subrotina, teriam sido necessárias linhas preparatórias antes (e linhas de conversão depois) de a chamar, a fim de adaptar as variáveis. Note que isto não é necessário no exemplo citado. Não pretendemos aqui negar valor às subrotinas; estas podem ter qualquer comprimento e utilizam qual-

quer variável numérica. A escolha dependerá do próprio programa.

Memória livre

Existe normalmente um método para saber qual a quantidade de memória que ficou livre depois de se dar entrada a um certo número de linhas de programa. Não existe qualquer palavra-chave BASIC standard para este fim, pelo que estamos perante mais um caso de «consulta ao manual».

Subrotinas em código-máquina

A lentidão inerente do BASIC quando comparado com as linguagens compiladas não é normalmente óbvia porque o termo «lento» é relativo. Em certos casos, no entanto, certas partes de um programa aumentam excessivamente a sua duração. Estas situações podem ser resolvidas escrevendo essa parte do programa sob a forma de uma subrotina em código-máquina. Além da tarefa de escrever o código-máquina, resta ainda o problema de ligar o programa BASIC à subrotina em causa. Também aqui não existe qualquer palavra-chave BASIC standard, se bem que USR, significando «subrotina do utilizador», seja hoje bastante popular. A sintaxe de uso varia bastante, por isso o que se segue será considerado apenas como guia:

USR N pode significar ir para a subrotina em código-máquina situada no endereço-máquina N.

O retorno da subrotina para o programa BASIC é realizado usando a instrução RTS no final do programa em código-máquina. RTS significa «retorno da subrotina».

Em algumas máquinas a palavra USR, seguida de uma variável, indica o parâmetro que deve ser passado para o acumulador da máquina. Este tema é no entanto bastante complicado, sendo estudado com mais pormenor noutros livros.

Esperando pela entrada por teclado

É possível que o seu BASIC contenha a palavra-chave INKEY. Esta é usada para dar algum tempo ao operador para carregar numa tecla. O tempo permitido é definido entre parêntesis a seguir à função.

Assim:

```
X = INKEY (N)
```

significaria «esperar durante N unidades de tempo». Se se carregou numa tecla dentro deste limite de tempo, o código ASCII da tecla premida é colocado em X. As unidades de tempo empregues podem equivaler a centésimos de segundo, caso em que N = 100 significaria esperar um segundo.

Para teclas não numéricas pode-se usar uma variante de INKEY. INKEY$ seria utilizada do seguinte modo:

```
A$ = INKEY$(100)
```

colocaria em A$ o caracter premido em vez do código ASCII.

Re-numeração

Depois de uma extensa sessão de programação, ao longo da qual foram realizadas bastantes correcções e introduzidas muitas linhas novas, o resultado fica pouco «estético». Os números de linha, que inicialmente pareciam disciplinados, evoluindo agradavelmente de dez em dez, depressa começam a parecer-se com uma sequência

numérica completamente casual. Emendar isto pode ser aborrecido, porque é muito fácil cometer erros na medida em que se devem mudar não apenas os números de linha mas também todas as referências a eles existentes em GOTO, IF THEN, GOSUB, ON GOTO, ON GOSUB. É de facto um trabalho enorme. Centos BASIC's, mais completos do que outros, oferecem-nos no entanto uma belíssima palavra-chave denominada RENUMBER S,I. É usada apenas no modo de comando, o que significa que não pode ser escrita à frente de um número de linha. Assim:

```
RENUMBER 100,10
```

dará uma nova numeração ao programa, de tal modo que a primeira linha ficará com o número 100 e todas as seguintes serão espaçadas de 10. Todas as referências a números de linha são correctamente modificadas. Por exemplo:

```
RENUMBER 50,4
```

renumerará a partir de 50, com um espaço de 4 linhas entre cada instrução.

RESUMO

Um *bit* binário apenas assume os valores de 1 ou 0.

A quatro bits chama-se «nibble», a oito «byte».

Começando a partir da esquerda, o «peso» dos bits é 1, 2, 4, 8, 16, etc.

A RAM é volátil, sendo perdida a informação que contém quando se corta a energia.

A ROM não é volátil, apenas podendo ser usada a informação já nela existente. Não é possível usar o BASIC para escrever na ROM novas informações.

Os programas por nós escritos vão para a RAM. As informações da ROM são usadas para decifrar e executar o programa.

O comprimento de palavra é o número de bits processados de cada vez, sendo normalmente de oito bits. Os microcomputadores da segunda geração utilizam já 16 bits ou mais.

O maior número binário puro numa palavra de oito bits é 255, em decimal; o maior número em notação «complemento de dois» é 127.

O *complemento de dois* permite tornar os números positivos ou negativos. O bit mais significativo é o bit de sinal; 1 representa negativo, o 0 positivo.

O *complemento de dois* é formado invertendo os bits e somando 1.

O hardware do computador realiza a subtracção somando o *complemento de dois* a outro número.

Um número de vírgula flutuante dispõe normalmente de cinco bytes no programa-intérprete BASIC.

Os «pointers» de endereços usados em código-máquina têm normalmente um comprimento de dois bytes. O byte de «menor ordem» é o menos significativo, e o de «maior ordem» é o mais significativo. Não existe uma posição estandardizada dos dois bytes.

O byte de maior ordem vale 256 vezes o seu valor «facial».

A maior parte das saídas por visor são ditas «memory mapped», sendo reservada para este fim uma certa área da memória, normalmente de 1000 ou mais células.

Uma outra área da RAM é reservada para o sistema operante. A alteracção destes «pointers» sensíveis pode conduzir à falha da máquina (crash). Com cuidado, é possível alterar o comportamento do sistema, se bem que não necessariamente para melhor...

A POKE é um operador híbrido. Armazena um número num endereço-máquina.

A PEEK é um outro operador híbrido. «Observa» o conteúdo de um endereço. Trata-se de uma função, o que significa que o endereço em causa deve estar entre parêntesis.

As instruções PEEK e POKE são usadas para forçar a saída de caracteres para o visor, alterar o sistema de

funcionamento e comandar dispositivos externos ligados ao port de saída.

A palavra-chave AND pode ser usada para limpar bits escolhidos colocando 0's numa máscara.

A tecla OR pode ser usada para dar o valor 1 a bits escolhidos, colocando 1's numa máscara.

As operações AND e OR não são aritméticas. Têm apenas a ver com os bits.

As instruções gráficas podem ser usadas de dois modos. As teclas gráficas são fáceis de usar, mas dão resultados pouco sofisticados. Os gráficos de alta resolução necessitam de palavras BASIC especiais, como PLOT x,y e DRAW x1,y1.

Como a realização de imagens gráficas de alta resolução exige o uso de um espaço de memória proporcional ao grau de resolução, é habitual dispor de um modo de escolha.

Os microcomputadores contêm normalmente geradores de números pseudo-aleatórios e relógios para sincronização e definição de atrasos.

É possível estruturar funções definidas pelo utilizador, diferentes das originais, recorrendo à instrução DEF FN A(X), mas estas são limitadas normalmente a uma linha do computador. No formato desta instrução, X é a variável *fictícia* (dummy variable), que pode ser substituída livremente quando se chama a função através de FN A.

Estas funções definidas pelo utilizador podem em certos casos ser usadas nos programas como alternativa às subrotinas.

Quando num dado programa se enfrente um problema grave de velocidade de execução é possível recorrer ao uso de subrotinas em código-máquina.

Existe em BASIC uma instrução, INKEY, que pode ser usada para obrigar a máquina a esperar pela entrada de uma informação por teclado.

Certas máquinas permitem renumerar as linhas de um programa de modo a melhorar a sua aparência.

8

FICHEIROS E REGISTOS

O computador moderno adapta-se perfeitamente às necessidades de armazenamento de informações e sua recuperação. O custo das memórias é hoje suficientemente baixo para permitir aos burocratas reunirem toda a informação útil ou inútil que desejem. Como todo este tema da organização de ficheiros está intimamente ligado ao das memórias, é lógico que comecemos por estudar as «caixas negras» disponíveis para uso como memória *de apoio*. A memória semicondutora RAM que existe no interior do computador actua com grande rapidez, mas devido ao facto de ser volátil e ao custo por byte não é de facto apropriada ao armazenamento de dados. Esta tarefa é portanto relegada para sistemas externos, ou «periféricos». O estudo resumido que faremos em seguida é suficiente para dar uma ideia do modo como se pode utilizar o BASIC neste campo.

Fita em cassette

Este é o tipo mais barato de memória de apoio, sendo de bastante confiança depois de se ter ajustado correctamente o nível dos comandos. A quantidade de dados que podem ser introduzidos num dos lados de uma fita de gravação C60 varia conforme o tipo de sistema de gravação empregue pela máquina. Para orien-

tação, diremos que é possível armazenar num dos lados cerca de 200 000 bytes. O problema da fita em cassette é ser muito lenta e o facto de apenas permitir um acesso *sequencial.* Para ter acesso a qualquer parte da fita, é necessário ler tudo o que se encontra antes, o que pode ser desnecessário.

Diskettes floppy

Neste caso a informação é gravada num pequeno disco maleável que, quando «chamado» pelo programa, roda a uma velocidade de aproximadamente 300 rotações por minuto até ser descoberto o bloco de informações relevante. Neste sentido, é ainda um sistema de acesso sequencial, mas a gravação é realizada numa de muitas pistas concêntricas, sendo cada uma delas acedida por uma cabeça radial de leitura/escrita que pode ou não lê-la integralmente. Neste sentido, a diskette pode ser considerada como um dispositivo de acesso aleatório. É possível descobrir em segundos qualquer bloco de dados, o que comparado com o gravador de cassettes é já de facto muito rápido. Conforme a densidade de gravação, uma diskette típica de cinco polegadas (12,5 centímetros) pode armazenar qualquer coisa entre 200 000 e 500 000 bytes num dos lados. Outras diskettes possuem maiores dimensões: oito polegadas.

O disco rígido Winchester

Ao contrário da diskette, que é substituível, o disco rígido é fixo, o que sob certos pontos de vista é uma desvantagem porque não possibilita o armazenamento de discos. No entanto, a velocidade e a capacidade destes discos compensa integralmente esta desvantagem. Um disco Winchester pode armazenar muito mais do que uma diskette, normalmente 10 milhões de bytes ou mais.

A unidade de discos permutável

Estas unidades correspondem à variedade «Rolls-Royce» do mercado... A velocidade, a capacidade e infelizmente o custo são astronómicos, sendo improvável que este tipo de disco venha alguma vez a ser associado aos microcomputadores.

A memória «de bolhas»

Um dos sonhos no mundo dos computadores tem sido a obtenção de uma memória de apoio barata e sem partes móveis. A memória «bubble» (de bolhas) armazena os bits em pequenas «ilhas» magnetizadas que se deslocam por forças magnéticas e não mecânicas. De um ponto de vista teórico este dispositivo é bastante prometedor, mas aparentemente têm-se sucedido os problemas práticos na sua concepção. Certas firmas gigantes gastaram muitos anos (e muitos milhões de dolares) desenvolvendo este sistema e acabaram por abandoná-lo. As poucas firmas que ainda o produzem não conseguem comercializá-lo a preços competitivos relativamente aos sistemas de armazenamento de informação mais tradicionais.

REGISTOS E FICHEIROS

O termo *ficheiro* é usado para cobrir muitos aspectos do armazenamento de informações, mas defini-lo-emos como um conjunto completo de informação colocado sob um título específico. Um ficheiro consistirá portanto num certo número de *registos* individualizados, e estes, por sua vez normalmente são divididos em *campos*. Por exemplo:

Nome do ficheiro: Clientes.

Nome	*Telefone*	*Crédito*	*Banco*
Fothergill J	655 4857	£3000	Lloyds

O ficheiro diz respeito a *clientes*, mas este registo particular refere-se a pormenores privados de um cliente chamado Fothergill J; existindo em cada registo quatro campos.

O campo-chave

Um dos campos possui um estatuto mais importante do que os outros, sendo designado por *campo-chave* porque, quando se procura um determinado registo num ficheiro, é normalmente este campo que serve para a sua identificação. A chave do exemplo anterior é «Nome», e normalmente ocupa o campo da esquerda.

Número de campos

Este número depende naturalmente do grau de pormenor em cada ficheiro, sendo o número dois e o máximo não excedendo normalmente dez. Não existe nenhuma boa razão para não incluir cem campos num registo se a capacidade da memória for suficiente para isso. Antes de decidir o número de campos, no entanto, não se esqueça de que alguém vai ter que sentar-se à frente do teclado e dar entrada à enorme quantidade de detalhes. As dimensões limitadas do visor impõem uma outra restrição. É inútil ter uma enorme quantidade de campos quando é difícil apresentar o conteúdo do registo no visor.

COMPONENTES DE UM SISTEMA DE FICHEIRO COMPUTARIZADO

Superficialmente, o programa deve poder armazenar ficheiros de dados e recuperar qualquer registo para exame. Existem no entanto outros aspectos que devem ser tidos

em conta num bom sistema de armazenamento de dados. O mais importante é certamente o módulo ou subrotina que permite ao operador criar um *novo* ficheiro.

Módulos para criar um ficheiro

Para criar um novo ficheiro o programa deve pedir ao operador que indique o nome do ficheiro, o seu número de registo, a quantidade de campos (colunas) que contém, o título de cada coluna e evidentemente as informações desejadas em cada campo. Toda a informação existente no ficheiro será guardado num quadro bidimensional A$(R,C). O subscrito R será o *número de registo*, e o subscrito C o *número de campo*. É conveniente associar registos a linhas e campos a colunas.

Os títulos das colunas podem ser convenientemente «escondidos» na linha de subscrito zero. Assim, A$(0,1) armazenará o título do campo chave (coluna 1). A$(0,2) armazenará o título da coluna 2, A$(0,3) o da coluna 3, etc. Nestas condições a linha 1 será o primeiro registo, o que é lógico.

O número de registos é guardado numa variável separada, de preferência uma variável de cadeia a fim de evitar uma mistura de modos quando chega o momento de enviar o ficheiro para a memória. O número de campos deve ser incluído também numa variável de cadeia. As variáveis escolhidas poderiam ser por exemplo Y$ para o número de registos (a dimensão vertical da matriz) e X$ para o número de campos (a dimensão horizontal da matriz). Ambas são necessárias para dimensionar o quadro, não apenas quando se cria o ficheiro mas também quando se o utiliza a partir da memória de apoio. Devemos recordar que quando se programa o módulo para recuperar o ficheiro se deve incluir Y$ e X$ como parte dos «pertences» do ficheiro..

Como os erros são impossíveis de evitar, é vulgar incluir no programa possibilidades de emenda e modifi-

cação. A procura de um determinado registo através do campo-chave, ou de um conjunto de registos que satisfaçam um critério qualquer, é uma capacidade muito útil. Deve haver portanto um método de recuperar um ficheiro de memória e carregá-lo no computador. Se o ficheiro é muito grande e a memória RAM é insuficiente para receber o ficheiro completo, é melhor carregar apenas o registo pretendido ou talvez dividir o ficheiro.

A formação de um sistema de armazenamento de dados sofisticado não é uma tarefa fácil mesmo para um programador experiente, e não seria conveniente incluir um programa detalhado sobre o assunto num livro com as dimensões deste. Em vez disso, o programa será dividido em módulos que podem ser ligados entre si ou modificados de acordo com as necessidades. A importância da validação das entradas e dos vários retoques que dão profissionalismo a um programa não será considerada visto que se parte do princípio de que as observações anteriores sobre estas técnicas terão sido assimiladas. Incluí-las poderia introduzir alguma confusão e obscurecer os aspectos essenciais. De qualquer modo, podem ser acrescentadas depois de a estrutura de cada módulo ter sido ensaiada.

Podem ocorrer duas situações no momento de criação do ficheiro. Se o operador sabe de avanço quantos registos virão a existir no ficheiro completo, o problema é simples porque é possível definir imediatamente a dimensão exacta. Por exemplo, se se sabe que existirão, digamos, 100 registos de cinco colunas cada, o dimensionamento do quadro pode ser definido por DIM$ (100,5). Se, por outro lado, o operador não tem nenhuma ideia do tamanho final do ficheiro quando o cria, necessitará de adivinhar esse tamanho, errando conscientemente para mais a fim de evitar uma mensagem de erro no caso de o ficheiro ultrapassar as dimensões previstas.

Mas existe ainda um outro problema. Como é que o computador sabe quando recebeu o último registo se não conhece desde início a dimensão exacta do ficheiro? Um

método simples consiste em empregar um truque conhecido por «sentinela», que o computador procurará no início de cada registo. A «sentinela» pode ser uma cadeia de caracteres como «FINISH», as letras FDF (EOF) significando «fim da ficha» (End of File), ou aliás qualquer conjunto de caracteres que não seja usado como dados. Os módulos que se seguem cobrem ambos os casos; primeiramente com conhecimento prévio das dimensões, depois sem ele.

Objectivo: criar um ficheiro novo, considerando que a sua dimensão é conhecida.

```
999  REM*CRIAR FICHEIRO DE DIMENSÃO
     CONHECIDA*
1000 CLS:REM*LIMPAR VISOR*
1010 INPUT «QUAL É O NOME DO FICHEIRO?»;
     NA$
1020 INPUT «QUANTOS REGISTOS?»;Y
1030 INPUT «QUANTOS CAMPOS?»;X
1040 DIM A$(Y,X):REM*DIMENSÕES DO
     FICHEIRO*
1050 CLS:REM*INDICAR TÍTULO DOS CAMPOS*
1060 FOR C = 1 TO X STEP 1
1070 PRINT «QUAL É O TÍTULO DO CAMPO?»C
1080 INPUT A$(0,C)
1090 NEXT C
1100 CLS:REM*INDICAR OS DADOS PARA CADA
     REGISTO*
1110 FOR R = 1 TO Y STEP 1
1120 PRINT «INICIAR INPUT DE DADOS PARA
     REGISTO»R:PRINT
1130 FOR C = 1 TO X
1140 PRINT «INDICAR»A$(0,C)
1150 INPUT A$(R,C)
1160 NEXT C
1170 PRINT
1180 NEXT R
1190 GOTO 500:REM*OPÇÕES PRINCIPAIS*
```

Objectivo: criar um ficheiro novo, considerando que a sua dimensão é desconhecida.

```
999  REM*CRIAR FICHEIRO DE DIMENSAO
     DESCONHECIDA*
1000 CLS:REM*LIMPAR VISOR*
1010 INPUT«QUAL É O NOME DO FICHEIRO?»;NA$
1020 INPUT«QUANTOS CAMPOS?»;X
1030 DIM A$(50,X):REM*AVALIAÇÃO*
1040 CLS:REM*INDICAR TÍTULOS DE CAMPOS*
1050 FOR C = 1 TO X STEP 1
1060 PRINT«QUAL É O TÍTULO DO CAMPO?»C
1070 INPUT A$(0,C)
1080 NEXT C
1090 CLS:REM*INDNCAR SENTINELA*
1100 PRINT«INICIAR INPUT DE DADOS. QUANDO
     TERMINAR»
1110 PRINT«O FICHEIRO, INDICAR 'EOF' NO
     CAMPO 1»
1120 PRINT
1130 R = 1
1140 PRINT«INICIAR INPUT DE DADOS PARA
     REGISTO»R:PRINT
1150 FOR C = 1 TO X
1160  PRINT «INDICAR»A$(0,C)
1170 INPUT E$:REM*VERIFICAR EOF F*
1180 IF E$ = «EOF» THEN 1240
1190 A$(R,C) = E$
1200 NEXT C
1210 PRINT
1220 R = R + 1:PRINT
1230 GOTO 2040
1240 REM*FINAL DO MÓDULO DE CRIAÇÃO DE
     FICHEIRO*
1250 GOTO 500:REM*OPÇÕES PRINCIPAIS*
```

Note que o loop exterior que determina o número de registo R é programado pelo método de incrementação, R = R + 1, na linha 1220, porque um loop FOR/

/NEXT deve possuir parâmetros definidos. Estes não existem no início devido ao facto de não se conhecer o valor de R. A verificação do «end of file» é realizado na linha 1180, levando o módulo a passa ao final do loop exterior na linha 1240.

Ambos os módulos de programa receberam números de linha a partir de 1000 porque nunca serão necessários simultaneamente no mesmo programa.

O módulo seguinte trata da apresentação de todo o ficheiro. É conveniente permitir uma escolha entre apresentar todo o ficheiro, um registo de cada vez, ou apenas um determinado registo. No entanto, quando se pede este último, surge um problema no que se refere à identificação. Por exemplo, pedimos o registo através de um campo-chave ou por um número de registo? Uma vez mais, seria bom poder-se escolher. Infelizmente, quanto maior é o número de escolhas permitidas num programa, maior é o número de linhas IF/THEN, e quando acrescentamos aos módulos as validações de entradas o conjunto começa a ser um tanto pesado. No entanto, por agora não nos preocuparemos com a validação de entradas a fim de não dificultar a compreensão deste tema.

Objectivo: apresentar um ficheiro.

```
1999 REM*APRESENTAR FICHEIRO*
2000 CLS:REM*LIMPAR VISOR*
2010 PRINT TAB(6) «OPÇÕES»:PRINT:PRINT
2020 PRINT«APRESENTAR TODO O FICHEIRO...
     1»:PRINT
2030 PRINT«APRESENTAR REGISTOS ATRAVÉS
     DE CAMPO-CHAVE.....2»:PRINT
2040 PRINT «APRESENTAR REGISTO ATRAVÉS
     DE NÚMERO DE REGISTO.....3»:PRINT
2050 PRINT «VOLTAR ÀS PRIMÁRIAS.....4»:PRINT
2060 PRINT:PRINT
2070 PRINT «INDICAR NÚMERO DA OPÇÃO
     DESEJADA»
2080 INPUT K
```

```
2090 ON K GOTO 2100, 2200, 2300, 500
2099 REM*APRESENTAR TODO O FICHEIRO*
2100 CLS:REM*LIMPAR VISOR*
2110 FOR R = 1 TO Y
2120 FOR C = 1 TO X
2130 PRINT A$(0,C) TAB(18)A$(R,C)
2140 NEXT C
2150 PRINT«.....»
2160 PRINT «CARREGAR BARRA DE ESPAÇOS
     PARA REGISTO SEGUINTE»
2170 GET K$
2180 IF K$ = «»THEN 2170
2190 IF K$ = «»THEN 2197
2195 GOTO 2160
2197 NEXT R
2198 GOTO 2000:REM*RETORNO À APRESENTAÇÃO
     DE OPÇÕES*
```

Note a linha 2130. Apresenta o título do campo e os dados relevantes na mesma linha. Como a passagem do texto se realizaria a uma velocidade demasiado grande para permitir a leitura de cada registo, o programa pára depois de cada registo e espera até se carregar na barra de espaços. A acção é dirigida pelas linhas 2160 a 2195. A aparência irregular dos números de linha no final deveu-se a erro do autor ao avaliar a extensão das opções na linha ON GOTO (2090): os quatro números deveriam ter sido mais espaçados. No entanto, serve para ilustrar o conselho dado mais atrás quanto ao espaçamento das linhas.

O módulo não está ainda completo. Mas vamos programar agora a segunda opção, que permite a apresentação de um dado registo pedido através do respectivo *campo-chave.*

```
2199 REM*APRESENTAR REGISTO POR
     CAMPO-CHAVE*
2200 CLS:REM*LIMPAR VISOR*
2210 PRINT«INDICAR CAMPO-CHAVE»
```

```
2220 INPUT F$
2230 FOR R = 1 TO Y
2240 IF A$(R,1) <> F$ THEN 2275:REM*NÃO
     CONCORDÂNCIA*
2250 FOR C = 1 TO X
2260 PRINT A$(0,C)TAB(18)A$(R,C)
2265 NEXT C
2270 GOTO 2280
2275 NEXT R
2280 PRINT «PREMIR QUALQUER TECLA»
2285 GET K$
2290 IF K$ = «» THEN 2285
2295 GOTO 2000:REM*RETORNO AS OPÇÕES*
```

O computador procura no ficheiro até que o campo-chave, A$(R,1) concorde com os dados pedidos em F$. O registo é então impresso pelo loop nas linhas 2250 a 2260, abandonando-se o loop exterior. Os registos que não concordam são impedidos de aparecerem na saída pelo salto da linha 2240 para 2275.

Programemos agora a terceira opção.

```
2299 REM*APRESENTAR REGISTO POR NÚMERO*
2300 CLS:REM*LIMPAR VISOR*
2310 INPUT«INDICAR NÚMERO DE REGISTO»;R
2320 FOR C = 1 TO X STEP 1
2330 PRINT A$(0,C)TAB(18)A$(R,C)
2340 NEXT C
2350 PRINT
2360 PRINT «PREMIR QUALQUER TECLA»
2370 GET K$
2380 IF K$ = «» THEN 2370
2390 GOTO 2000:REM*RETORNO AS OPÇÕES*
```

Note a relativa simplicidade deste módulo quando comparado com o anterior. Dir-se-ia que procurar registos pelo número é mais fácil, e portanto mais eficaz, do que procurar por campo-chave. Tende igualmente a permitir menos erros do ponto de vista do operador.

Antes de continuarmos com novos módulos, convém consolidarmos aqui o que já aprendemos. Se bem que a página da opção principal ainda não tenha sido programada, sê-lo-á a partir da linha 500. É mais seguro deixar isto para o fim. Temos até agora duas opções principais, a Criação do Ficheiro e a Apresentação do Ficheiro. Esta última possui por sua vez quatro sub-opções, sendo a quarta um simples retorno às opções principais.

Seguidamente programaremos a opção de modificações dos registos, que envolve a procura de um registo e a sua escrita de novo no quadro.

```
2999 REM*MODIFICAR REGISTO*
3000 CLS:REM*LIMPAR VISOR*
3010 PRINT «INDICAR NÚMERO DE REGISTO A
     MODIFICAR»
3020 INPUT R
3030 PRINT «INDICAR NÚMERO DE COLUNA»
3040 INPUT C
3050 PRINT C
3060 PRINT A$(0,C) TAB(18) A$(R,C):REM*
     IMPRIMIR DADOS EXISTENTES*
3070 PRINT:REM*ALTERAR DADOS*
3080 PRINT «INDICAR DADOS CORRECTO PARA»
     A$(0,C)
3090 INPUT A$(R,C)
3100 GOTO 500:REM*OPÇÕES PRINCIPAIS*
```

O registo e o campo específico a modificar no interior do registo são os primeiros elementos apresentados. Pode-se então alterar os dados desta coluna. Se for necessário alterar outros dados, a via é novamente pelas Opções Principais. Trata-se talvez de um inconveniente, mas tem o mérito da simplicidade.

Seria possível incluir aqui muitas outras opções, mas antes de tratarmos da leitura e escrita de ficheiros de e para memórias de apoio devemos incluir ainda um outro módulo. Este permite procurar num ficheiro todos

os registos que satisfaçam um determinado critério. Por exemplo, podemos querer descobrir quais as pessoas que têm mais de 60 anos de idade.

Objectivo: procurar uma concordância de dados.

```
3999 REM*PROCURAR DADOS*
4000 CLS:REM*LIMPAR VISOR*
4010 PRINT «QUAL É O TÍTULO DO CAMPO?»:
     PRINT
4020 INPUT H$
4030 PRINT«QUAL É O DADO PRETENDIDO?»
4040 INPUT D$
4050 FOR R = 1 TO Y STEP 1
4060 FOR C = 1 TO X
4070 IF A$(0,C) <> H$ OR A$(R,C) <> D$
     THEN 4090
4080 PRINT A$(R,1):REM*IMPRIMIR
     CAMPO-CHAVE*
4090 NEXT C
4100 NEXT R
4110 PRINT«PREMIR QUALQUER TECLA»
4120 GET K$
4130 IF K$ = «»THEN 4120
4140 GOTO 500:REM*OPÇÕES PRINCIPAIS*
```

A linha 4010 pede o «título do campo». Consideremos que o operador escreve «idade». A linha 4030 pede «dado pretendido». Suponhamos que o operador escreve «60». O loop FOR interior (linhas 4060 a 4090) examina cada coluna de um registo e quando descobre que a coluna desejada (idade) e os dados dessa coluna (60) pertencem ao mesmo registo (mesmo número «R»), imprime o campo-chave, que provavelmente será o nome da pessoa. Este procedimento é repetido para todos os registos pelo loop FOR exterior, de tal modo que se obtém uma lista de todas as pessoas com mais de 60 anos.

Estude cuidadosamente a linha 4070 porque é fácil trocar as condições. Se qualquer das condições for «ver-

dadeira» (título do campo diferente de H$ ou idade diferente de D$) a instrução PRINT A$(R,1) da linha 4080 não é considerada. Assim, estaremos a invalidar as pessoas que não têm as características desejadas. Os estudantes de lógica de Boole reconhecerão aqui a aplicação da lei de DeMorgan; não(A e B) é o mesmo que não(A) ou não(B).

É possível incluir muitas outras capacidades numa página de opção, mas espero que o leitor já seja capaz de as programar de acordo com as suas necessidades.

Para juntar os vários módulos, programaremos agora a página de Opções Principais, deixando no entanto espaço para os dois módulos finais relativos à escrita e leitura dos ficheiros a partir das memórias de apoio.

Objectivo: apresentar a página de opções.

```
500 CLS:REM*LIMPAR VISOR*
510 PRINT TAB(8)«OPÇÕES PRINCIPAIS»
520 PRINT«CRIAÇÃO DE FICHEIRO.....1»
530 PRINT«APRESENTAÇÃO DE FICHEIRO.....2»
540 PRINT«MODIFICAR REGISTO.....3»
550 PRINT«PROCURAR DADOS SEMELHANTES
    .....»
560 PRINT«OBTER FICHEIRO.....5»
570 PRINT«CARREGAR FICHEIRO.....6»
580 PRINT:PRINT:PRINT
590 INPUT«INDICAR NÚMERO DA OPÇÃO
    DESEJADA»;K
600 ON K GOTO 1000,2000,3000,4000,5000,6000
```

Os números que pertencem à ON GOTO na linha 600 ligam-se aos endereços iniciais anteriores. Assim, se escrevermos no teclado 2 (K = 2), o destino do salto executado pelo programa é o segundo número (2000). Recorde que o módulo de Criação de Ficheiro foi desenvolvido em duas formas diferentes, ambas com o mesmo número de linha (1000) de modo a que qualquer deles possa ser usado, mas não ambos.

ARMAZENAMENTO E CARGA DE FICHEIROS DE DADOS

Suponhamos que o leitor gasta uma hora escrevendo pacientemente nomes, endereços e números de telefone de amigos e conhecidos no âmbito da Opção Principal 1. Toda esta informação se dirige para a RAM, pelo que para a manter no sistema é necessário deixar o computador permanente ligado, sendo aliás impossível utilizá-lo simultaneamente para qualquer outra tarefa! É aqui que se torna essencial o uso de alguma forma de memória de apoio. Estes dispositivos, referidos anteriormente, podem ser usados não só para guardar programas mas também para armazenar quaisquer dados de que estes programas necessitem. Assim, distinguimos entre «ficheiros de programa» e «ficheiros de dados».

Na maior parte dos sistemas operativos dos microcomputadores a carga e a poupança dos programas (ficheiros de programas) é sempre um trabalho fácil para o operador. Normalmente, basta a instrução SAVE para armazenar um programa em fita. Algumas máquinas permitem-nos nomear um programa antes de o armazenar, usando a palavra SAVE seguida pelo nome entre aspas. SAVE «PROGRAMA DE REGISTOS» colocaria o nome em causa como título do ficheiro, na fita gravada, pelo que quando mais tarde necessitássemos desta poderíamos escrever LOAD «PROGRAMA DE REGISTOS». O computador procuraria então este programa na fita, carregando-o em seguida na RAM.

O processo usado para guardar e carregar ficheiros de dados é no entanto diferente porque neste campo existe muito pouca estandardização nos processos. Consequentemente só podemos indicar algumas regras gerais. Sobre a sua programação que pode variar de máquina para máquina os procedimentos usados dependem do tipo de sistema de input/output usado. Estes sistemas servem para enviar ou receber sinais por um conjunto de fios que ligam o computador ao dispositivo externo. Por exemplo, um sistema é designado por «bus» («canal»)

IEEE ou HPIB, sendo esta última designação usada para o Hewlett Packard Interface Bus. Trata-se de um sistema que permite ligar entre si até quinze dispositivos periféricos (incluindo o próprio computador). Cada dispositivo recebe um número, a que se chama *endereço principal*. Certos dispositivos, como os gravadores de cassettes, podem possuir duas funções distintas. Podemos escrever (armazenar) informações neles, ou ler informações neles contidas. Cada uma destas funções recebe então um *endereço secundário*, por exemplo 0 para o modo «ler» e 1 para o modo «escrever».

Qualquer que seja o sistema usado, a ordem inicial consistirá normalmente em OPEN (abrir) um ficheiro. A palavra OPEN é responsável por «acordar» o bus de input/output. Citaremos um exemplo do sistema de canal HPIB usado nos computadores CBM:

```
100  OPEN 1,1,0
```

abrirá o ficheiro número 1 no dispositivo 1 com o endereço secundário 0.

Em linguagem simples, isto significa «preparar o sistema de input/output para activar o gravador no modo ler». Os três parâmetros citados na instrução OPEN são respectivamente o número do ficheiro (uma escolha arbitrária), o número do dispositivo, e o endereço secundário. Pelo contrário, OPEN 27,1,1 abriria o ficheiro número 27 do gravador para o modo «escrever». Opcionalmente, pode-se acrescentar o nome de um ficheiro, contido entre aspas, à instrução OPEN; isto permitirá uma identificação mais fácil no futuro. Assim, OPEN 1,1,0 «BLOGGS» preparará o sistema para carregar um ficheiro chamado BLOGGS a partir da fita. Pode existir uma outra memória de apoio, por exemplo uma diskette, ligada ao sistema. Suponhamos que esta possui o número 8. OPEN 3,8,0 prepararia este dispositivo; número de ficheiro 3, dispositivo 8 e endereço secundário 0.

Quando a parte do programa que diz respeito à actividade da memória de apoio é terminada, o ficheiro deve

ser CLOSED (fechado) a fim de libertar o canal de input/output do dispositivo em causa. A palavra usada no sistema CBM é CLOSE n, onde n é o número do ficheiro (que deve ser o mesmo que se utilizou na instrução OPEN inicial). Assim, se tivermos aberto um ficheiro com o número 16, eventualmente teremos de escrever no fim uma instrução CLOSE 16. Qualquer tentativa de abrir um ficheiro já aberto provocará uma mensagem de erro enviada pelo sistema operativo.

Como temos de considerar um exemplo qualquer para escrevermos os nossos módulos de armazenamento e carga de ficheiros de dados, iremos supor que usamos o canal HPIB. As únicas linhas do programa que terão de ser modificadas, se o seu sistema for diferente, serão os formatos do OPEN ficheiro e de CLOSE ficheiro. Estas linhas podem ser alteradas de modo a concordarem com os requisitos indicados no manual da máquina.

PRINT # n, A$

Se se usa a palavra PRINT sem qualquer qualificativo, o sistema considera que nos referimos a imprimir no *visor*. Para informar o sistema de que não é esse o nosso objectivo, teremos de alterar de algum modo a instrução PRINT. Tal como acontece em vários sistemas usados actualmente, utilizaremos aqui PRINT # n, onde n é o número do ficheiro usado na instrução OPEN. A marca # que surge a seguir à palavra PRINT é um símbolo muito usado nos Estados Unidos para designar «número». De todos os outros pontos de vista esta instrução especial comporta-se exactamente do mesmo modo que a instrução PRINT normal, excepto evidentemente que «imprime» no periférico definido pela instrução OPEN.

INPUT # n, A$

Do mesmo modo, para informar o computador de que deve esperar um input vindo de um periférico em vez do habitual teclado, usam-se novamente a marca

indicada e o número do ficheiro. Ao contrário da INPUT de teclado, o computador já não espera neste caso pelo operador. Pode ser necessária a intervenção deste para carregar no botão «PLAY» do gravador de cassette, mas mesmo isto dependerá do hardware da máquina usada.

Vamos programar agora os dois módulos restantes, que empregam todas estas palavras-chave.

Objectivo: Demonstrar os módulos SAVE e LOAD.

```
4999 REM*POUPAR FICHEIRO*
5000 OPEN 1,1,1,NA$:REM*ABRIR FICHEIRO PARA
     ESCRITA*
5010 PRINT #1,Y
5020 PRINT #1,X
5030 FOR R = 0 TO Y
5040 FOR C = 1 TO X
5050 PRINT #1,A$(R,C)
5060 NEXT C
5070 NEXT R
5080 CLOSE 1
5090 GOTO 500; REM*OPÇÕES PRINCIPAIS*
5100 REM**********************************
5999 REM*CARREGAR FICHEIRO*
6000 CLR:REM*LIMPAR TODAS AS VARIÁVEIS*
6010 CLS:REM*LIMPAR VISOR*
6020 INPUT«QUAL É O NOME DO FICHEIRO?»;NA$
6030 OPEN 1,1,0,NA$:REM*ABRIR FICHEIRO PARA
     LEITURA*
6040 INPUT #1,Y
6050 INPUT #1,X
6060 DIM A$(Y,X)
6070 FOR R = 0 TO Y
6080 FOR C = 1 TO X
6090 INPUT #1,A$(R,C)
6100 NEXT C
6110 NEXT R
6120 CLOSE 1
6130 GOTO 500: REM*OPÇÕES PRINCIPAIS*
```

Antes de tentarmos fazer uma análise detalhada dos dois módulos, é necessário sublinhar uma vez mais que certas linhas devem ser consideradas provisórias e podem necessitar de algumas modificações, consoante a máquina usada. Estas linhas são a 5000 e a 6030, que tratam da abertura de um ficheiro. O método de fechar ficheiros pode também ser diferente, pelo que as linhas 5080 e 6120 poderão necessitar também de ser modificadas. A estrutura restante é praticamente standard e, julgamos, bastante fácil de compreender. No entanto, começando pelo módulo «Poupar ficheiro», note que a instrução OPEN possui o nome de ficheiro NA$ no final. Este nome teria sido escolhido ao usar a opção Criar Ficheiro. As primeiras variáveis a serem impressas na fita são os limites do ficheiro, Y e X, que foram igualmente definidos durante a sua fase de criação. O corpo principal deste ficheiro é em seguida impresso usando as redes de loops FOR em R e C. O primeiro loop FOR vai de R = 0 a R = Y. O subscrito 0 é evidentemente necessário dado que foi usado para gravar os títulos das colunas.

O segundo módulo, relativo ao carregamento de um ficheiro existente em fita, é semelhante ao anterior excepto no que se refere à instrução OPEN que possui o endereço principal 0 para READING (leitura). A instrução INPUT da linha 6020 também pede o nome de ficheiro pretendido. Este encontra-se na variável de cadeia NA$, e é incluído na instrução OPEN. A instrução CLR da linha 6000 requer algumas explicações. O leitor recordar-se-á de que uma instrução DIM não deve ser redimensionada. Dado que apareceu no início do programa uma instrução de dimensionamento (vários módulos antes), é necessário anulá-la antes de introduzir outra. A instrução CLR limpa todas as variáveis existentes no computador. Isto pode parecer demasiado drástico, mas neste caso não tem qualquer consequência porque, quando se carrega um ficheiro, todos os dados anteriores se tornam inúteis. A instrução DIM é evidentemente necessária na linha 6060 porque, mesmo que as antigas

dimensões ainda se encontrem no computador, não temos garantias de que sejam suficientes para o novo ficheiro que pretendemos carregar.

Utilização do sistema de armazenamento de dados

As mensagens no visor e a página de opções constituem já um guia para o operador, mas pode ser útil incluir mais alguns comentários.

Partindo do «zero», sem quaisquer ficheiros de dados, a primeira opção deve consistir em Criar Ficheiro. Até se dispor de um ficheiro na RAM da máquina, as outras opções darão necessariamente resultados estéreis. A primeira tarefa, depois de ter dado entrada a todos os módulos, consiste em criar um pequeno ficheiro de ensaio com um mínimo de dados. Um bom ficheiro para ensaio poderá ter a seguinte aparência.

NOME	*IDADE*	*NOME DO FICHEIRO: BLOGGS*
SMITH	23	
BROWN	60	
ROBINSON	23	

A criação do ficheiro consistirá então em dar entrada ao nome do ficheiro (para NA$), ao número de registos = = 3, e ao número de colunas = 2. Terminada esta tarefa, pode-se usar a opção «apresentar ficheiro», e de preferência, primeiramente a sub-opção «scroll» ficheiro. Obtem-se então algo do seguinte género:

```
NOME        SMITH
IDADE         23
.....
NOME        BROWN
IDADE         60
.....
NOME        ROBINSON
IDADE         23
```

Se o programa se comportar deste modo, estará demonstrado que tanto a opção «criação de ficheiro», como a de apresentação completa do ficheiro (Scroll), estão razoavelmente correctas. Se não obtiver este resultado, examine cuidadosamente o seu programa, procurando os erros possíveis, e experimente de novo. Não se esqueça no entanto de que quando carrega na tecla RUN o ficheiro por si criado é perdido, pelo que deve introduzi-lo novamente na máquina. Use sempre o mesmo ficheiro enquanto estiver a fazer ensaios... o hábito permitir-lhe-á trabalhar mais depressa.

Deve então verificar as sub-opções restantes antes de experimentar a opção principal «modificar registo». Depois de modificar, volte à sub-opção «apresentar ficheiro» para verificar se a máquina aceitou de facto a alteração. Para verificar a opção «procurar dados semelhantes», peça à máquina uma lista de todas as pessoas com 60 anos de idade. Deverá obter o seguinte:

NOME BROWN

Para verificar melhor, peça à máquina as pessoas com 23 anos de idade, o que deve produzir a seguinte resposta:

NOME SMITH
NOME ROBINSON

Finalmente, experimente as opções SAVE e LOAD. De início poderão dar origem a problemas, devido às variações nos sistemas de transporte da banda e às modificações que possam tornar-se necessárias para adaptar o nosso programa ao procedimento de «abertura de ficheiro» aceite pela sua máquina.

A diskette

Este dispositivo é um luxo caro se não tiver um mas uma autêntica necessidade se o tiver! São realmente modernos, pelo menos «post-Dartmouth», não sendo por-

tanto surpreendente que o seu modo de utilização esteja ainda menos estandardizado do que a humilde fita em cassette. Foram introduzidos diversos sistemas operativos capazes de comandar as diskettes. O preferido actualmente nos pequenos sistemas computorizados é chamado CP/M. A sua popularidade depende inteiramente do facto de ter sido escrito muito Software para ele. Afirma-se que está estandardizado, mas isto apenas significa que existem também as inevitáveis variantes. Seria arriscado tentar dar aqui exemplos BASIC.

Outras opções de ficheiros

O programa que construimos tão laboriosamente deve ser considerado apenas como o esboço de um sistema de armazenamento de dados. Depois do leitor começar a entrar neste assunto deve tentar acrescentar mais opções, como por exemplo ordenar uma determinada coluna — mais uma capacidade vantajosa. O campo-chave, que é muitas vezes o nome, poderia ser melhorado separando-o por ordem alfabética, através do módulo referido no capítulo 6. Existe no entanto uma armadilha; é inútil ordenar um dos campos, a menos que os restantes o sejam igualmente. Se se esquecer isto apenas se obtem o caos, visto que os dados que pertencem a cada registo ficarão mal colocados no ficheiro.

Uma outra capacidade útil será por exemplo a obtenção de totais e de médias para cada coluna. O módulo para obter isto conteria as linhas que se seguem, além da habitual pergunta ao operador sobre o campo que deseja totalizar:

```
INPUT«QUAL O NÚMERO DO CAMPO A
TOTALIZAR?»F
FOR R = 1 TO Y
T = T + A$(R,F)
NEXT R
PRINT«O TOTAL É» T
PRINT«A MÉDIA É»T/Y
```

Temas de ficheiros

Os exemplos dados disseram respeito apenas a nomes de pessoas e respectivas idades, etc. Existe evidentemente uma quantidade praticamente infinita de temas que merecem um ficheiro. Os que se interessam por electrónica, por exemplo, lucrarão se arquivarem em ficheiros dados sobre transístores sob a forma TIPO, POTÊNCIA, POLARIDADE, TENSÕES, etc. A opção principal 4 (procurar dados semelhantes) poderia então ser usada para obter uma listagem de todos os transístores com uma dada característica.

Uma aplicação muito útil de um sistema de armazenamento de dados consiste em arquivar artigos de revistas. É frustrante procurar em pilhas de revistas uma qualquer jóia de sabedoria... Seria fácil (depois de estar terminado o trabalho de criação do ficheiro) procurar qualquer artigo através de um campo-chave apropriado.

RESUMO

Os ficheiros de dados são armazenados em memórias externas de apoio.

A fita em cassette é barata mas lenta, apenas permitindo por outro lado um acesso sequencial.

As diskettes são muito mais rápidas do que a fita porque permitem um acesso aleatório. As diskettes podem ser removidas da sua unidade de leitura, podendo ser armazenadas à parte.

Os discos Winchester (fixos) armazenam quantidades de dados enormes e permitem um acesso muito rápido. Normalmente o disco é fixo, não podendo ser armazenado à parte.

A memória «de bolhas» é um tipo de memória periférica não mecânica, mas ainda não conseguiu grande popularidade.

Um *ficheiro* é uma colecção de registos sob um mesmo título geral.

Um *campo* é um dos vários elementos de cada registo.

O *campo-chave* é normalmente o campo 1, sendo ele que identifica o registo.

As dimensões de um ficheiro dependem do número de registo e dos número de campos no interior de cada registo. Num dado ficheiro, o número de campos é normalmente uma constante.

Os componentes de um sistema de armazenamento de dados incluem sempre: (a) a criação do ficheiro; (b) a sua apresentação sob várias formas; (c) a modificação dos registos; (d) a procura de dados específicos no ficheiro; (e) o armazenamento do ficheiro na memória de apoio e o seu carregamento. Muitas outras capacidades podem ser acrescentadas consoante o espaço de que o programa possa dispor.

O corpo do ficheiro é mantido numa variável de quadro bidimensional com a forma A$(R,C), onde R é o número de registo e C o campo.

Os títulos dos ficheiros podem ser armazenados no subscrito 0 (R).

Se é conhecida a dimensão final do ficheiro no momento em que é criado, torna-se possível dimensionar rigorosamente a variável de quadro. Se esta dimensão for desconhecida, é necessário dimensioná-la de modo inteligente.

Qualquer programa com um comprimento razoável, como é o caso de um programa de armazenamento de dados, deve ser construído sob a forma de um conjunto de módulos, cada um deles capaz de ser verificado individualmente.

A recuperação de qualquer registo específico é realizada através do campo-chave ou do número de registo.

Se se pretende obter o output dum ficheiro completo, não se deve permitir que o computador o passe continuamente no visor. A máquina deve apresentar um registo de cada vez, só passando ao seguinte quando o operador carrega numa tecla para esse efeito.

Deve existir uma página de Opções Principais. Se algumas derem origem a sub-opções, uma destas deve permitir ao operador voltar à página de Opções Principais.

As palavras-chave BASIC que tratam da armazenagem e carga dos ficheiros de dados variam bastante. Os métodos de abertura e fecho de um ficheiro formam áreas não «conformistas», pelo que só podem ser resolvidos estudando o manual de cada máquina.

O método usado para armazenar e carregar *programas* é normalmente muito mais fácil do que o de armazenar e carregar ficheiros de dados.

PRINT #n,A$ é a instrução normalmente usada para «imprimir» na memória de apoio em vez de o fazer no visor. O parâmetro *n* é o número do ficheiro, arbitrariamente escolhido.

Se existe mais do que um tipo de memória de apoio no mesmo sistema, cada um desses tipos deve corresponder a um «número de periférico» que o possa identificar devidamente.

INPUT#n,A$ é a instrução complementar da anterior PRINT #n,A$. Permite a passagem da informação enviada pela memória de apoio para a variável A$.

9

LINHAS GERAIS DA PROGRAMAÇÃO

Estilo de programação

Depois de programar durante algum tempo desenvolve-se um certo estilo próprio. O facto de este estilo ser «bom» ou «mau» não é muito importante nas fases iniciais. É mais importante desenvolver a confiança que se tem na linguagem. Muitos afirmam a importância da «estrutura» e insistem em que os recém-vindos à programação devem aderir rigidamente a um conjunto de normas. A palavra *estrutura* tem um sentido específico, cobrindo um conjunto de regras e regulamentos impostos por conhecedores. Essas regras têm sem dúvida excelentes intenções, mas se forem tomadas demasiado a sério (e demasiado cedo) servirão apenas para diminuir o entusiasmo do pobre principiante. «Não desenvolva maus hábitos no início» é um daqueles exemplos irritantes de pomposidade pedante, servindo apenas como incitamento à rebelião... O principal problema que se sente ao principiar é conseguir fazer funcionar os nossos programas. Adaptá-los aos ditames da programação estruturada pode aprender-se gradualmente.

O que é a «programação estruturada»? Significa programar de tal modo que torne fácil a qualquer outra pessoa compreender o nosso programa. Significa evitar o uso excessivo de instruções GOTO, não obrigando os olhos a andarem para cima e para baixo repetidamente seguindo o rumo do programa. Significa dividir o pro-

grama em módulos que possam ser verificados individualmente, cada um deles com um ponto de entrada e outro de saída. Significa ainda muito mais, mas são estes os ingredientes essenciais. A sua filosofia global baseia-se na ideia de que a compreensibilidade e a possibilidade de fazer correcções são os objectivos principais; todas as outras considerações dependem destas, incluindo até a economia em posições de memória. Infelizmente, os defensores da estrutura deixam-se por vezes arrastar, tentando transformá-la numa espécie de religião em vez de aceitar aquilo que ela de facto é... um conjunto de regras de bom senso, que se desenvolvem naturalmente com a experiência.

Em vez de dar uma lista de exemplos daquilo que é uma boa e uma má estrutura, bastar-nos-á examinar a linha 1180 do nosso módulo de Criação do Ficheiro. O salto para fora do loop FOR antes de terminar, para a linha 1240, é do ponto de vista dos «estruturalistas» um crime hediondo (aproveitamos aliás para lhes apresentar as nossas desculpas). Se bem que isto não justifique o facto, devemos referir que o BASIC não se adapta bem à aplicação de princípios de estruturação. Existem no entanto algumas palavras-chave extra, em certas versões, que permitem melhorar esta linguagem. O novo microcomputador BBC (Acorn) utiliza um BASIC bastante avançado, incluindo certas características normalmente reservadas a linguagens de nível superior.

IF THEN ELSE é um exemplo de uma instrução que se presta a uma melhor estruturação do que a simples IF THEN. A principal diferença consiste na possibilidade de definir uma acção alternativa se a condição for falsa, em vez de passar automaticamente à linha seguinte. Por exemplo:

```
100  IF T = 235 THEN PRINT «MUITO ELEVADO»
     ELSE PRINT «OK»
```

Esta estrutura evita uma instrução GOTO.

Descoberta de erros

Existem dois tipos de erros: de origem sintática ou de origem lógica. Os erros de sintaxe são aqueles que ofendem a gramática da linguagem, e são os mais fáceis de resolver. Uma coisa simples, como por exemplo o uso de uma vírgula em vez de um ponto e vírgula, pode provocar uma mensagem de erro. Esquecer as aspas numa instrução PRINT é um erro bastante vulgar. Palavras-chave mal escritas são um outro tipo de erro muito comum.

Um tipo de «crime» sintático tem a ver com o uso de palavras-chave «embebidas». A maior parte dos BASIC's não gostam de encontrar palavras em partes «inesperadas» de uma instrução. Assim, se escrevermos a linha:

```
100  TO = D + 2
```

que superficialmente não parece prejudicial, a resposta é uma mensagem de erro porque, apesar de TO ser um nome legítimo de variável no sentido de possuir dois caracteres, é igualmente uma palavra BASIC. TO faz parte da estrutura do loop FOR. Este tipo de erro, no entanto, é aceite com alguma tolerância em certas versões de BASIC.

Os erros de adaptação são fáceis de ocorrer por serem esquecidas as aspas de um dos lados, por exemplo, ou por mistura de variáveis em cadeia e variáveis numéricas.

Como exercício, estudemos as seguintes linhas tentando descobrir erros de sintaxe, erros de adaptação, etc.

```
100  A = 56;B = 34
110  S$ = 'INCORRECTO'
120  IF A = BC + K THEN — 500
130  A$ = C$*5
140  FOR F = G TO K$ + 1 STEP — 1
150  GOTO K
```

```
160 ON A$ GOTO 1000,2000,3000
170 PRINT G,CK,G$:TAB(3)J
180 K = SQR((3 + Y))K
190 J = SQR(F$)
200 «PERIGO» = G$
```

Os erros aqui contidos são os seguintes:

100 O ponto e vírgula está errado; deve ser dois pontos.
110 O uso de uma só «aspa» é errado; devem ser duas aspas.
120 Não se pode usar um número de linha negativo.
130 As variáveis de cadeia não podem ser multiplicadas.
140 K$ é ilegal; deve ser uma variável numérica.
150 A maior parte dos BASIC's não permitem o uso de uma variável como número de linha.
160 A$ é proibida numa instrução ON GOTO.
170 TAB sem a PRINT correspondente.
180 A multiplicação implícita por K não é válida: tem de ter um asterisco.
190 Não é aceite a raiz quadrada de uma variável de cadeia.
200 «PERIGO» é uma variável inválida à esquerda do sinal de igual.

Um tipo especial de erros diz respeito aos limites numéricos. Em todas as máquinas existem limites impostos à grandeza de certas variáveis. Estes erros são do tipo «quantidade ilegal», existindo diversas maneiras de poderem manifestar-se.

1. Simples «overflow» numérico no interior de uma variável. Existe sempre um número máximo que não deve ser excedido. Felizmente, o limite é normalmente superior aos requisitos vulgares, sendo em geral da ordem de 10^{30}, o que em notação científica de computador equivale a 1E30. Existem poucos problemas de física

ou matemática que exijam números superiores a este. De facto, os matemáticos consideram 10^{100} como a maior dimensão prática de um número.

2. «Overflow» inconsciente causado pela tentativa de dividir por zero. É um erro muito aborrecido, que muitas vezes demora bastante tempo a ser descoberto. Em termos simples, dividir por zero é ilegal em matemática porque mesmo 1 dividido por zero conduz a um absurdo: o infinito. Como exemplo deste erro, consideremos a seguinte equação de Einstein, que relaciona a massa (M), a velocidade (V) e a velocidade da luz (c)

$$M = \frac{M_0}{\sqrt{1 - \left(\frac{V}{C}\right)^2}}$$

O termo M_0 é a massa inicial e M representa a massa obtida à medida que a velocidade do corpo aumenta. Enquanto v se mantém inferior a c (que corresponde a 3×10^8 metros), a equação é supreendente mas não prejudicial. Mas se v chegar a igualar c, o denominador transforma-se em 0 e a massa torna-se infinita! Num programa de computador, esta condição é rejeitada quando a equação produz um resultado para além do limite numérico da máquina. Para ilustrar este assunto, consideremos que se escreviam as seguintes linhas num programa de jogo «Star Trek»:

```
1000 FOR V = 1 TO C STEP 1E6
1010 K = (V/C) ↑ 2
1020 M = M0/(SQR(1 — K))
1030 PRINT V,M
1040 NEXT V
```

V aumenta em passos de um milhão de metros/segundo, e o computador nada tem a dizer sobre o assunto até ao momento em que V atinge 299 milhões de metros/segundo. No entanto, na última execução do loop, V atinge o mesmo valor que C, o que torna K = 1. A linha

1020 coloca agora o problema de dividir M0 por zero. Isto leva o computador a queixar-se. O modo mais simples de resolver o problema consiste evidentemente em escrever de novo a primeira linha de modo a que o limite seja um a menos:

```
1000 FOR V = 1 TO C — 1E6 STEP 1E6
```

A solução nem sempre é tão óbvia, e em certos casos nunca chega a ser satisfatória. Se, por exemplo, o aumento de V fosse programado pelo método de incrementação (M = M + 1) em vez de o ser através de um loop FOR NEXT, seria necessário incluir uma instrução IF THEN para determinar o momento de saída do loop. Isto envolveria uma decisão quanto ao ponto até onde V se pode aproximar de 0. Isto dependeria por outro lado da dimensão do incremento I, que nem sempre é conhecido. Neste caso poder-se-ia recorrer a uma solução grosseira mas eficaz que consiste em escrever a instrução IF THEN do seguinte modo:

```
IF V = C — 1E/30 THEN etc, etc.
```

A condição de «overflow» é assim evitada no último momento, mas evitando o corte drástico de V quando está ainda a uma diferença de um milhão de metros por segundo de C. As equações deste tipo aproximam-se do limite assintoticamente, e só ganham interesse quando o limite é quase atingido.

3. As variáveis inteiras estão sempre limitadas a um número muito mais reduzido do que as variáveis de vírgula flutuante. Os inteiros têm normalmente um comprimento de dois bytes. Significa isto que o equivalente binário do número cabe em dois bytes quando representado na forma «complemento de dois». Tendo em conta que o bit da extremidade esquerda (o bit mais significativo (msb)) é reservado para o sinal restam apenas quinze bits para o número positivo máximo, que

é então $2^{15} - 1$, ou seja, 32 767. O menor inteiro negativo é 32768 (mais um).

4. Existe sempre um limite para o número máximo de caracteres que podem existir numa variável de cadeia. É típico o uso de um só byte, e como os números negativos são neste caso um absurdo, é possível utilizá-lo na totalidade. Isto significa que uma variável de cadeia está normalmente limitada a $2^8 - 1$, ou seja, a um máximo de 255 caracteres. Este limite pode ser particularmente frustrante, em especial nos programas de acesso a registos, porque em muitas ocasiões um registo exige mais do que 255 caracteres. Uma palavra de aviso... não pense que a concatenação constitui uma forma de ultrapassar este limite. É inútil escrever A$ = B$ + C$, porque A$ continua a só poder armazenar o valor 255.

As funções TAB(n) estão igualmente sujeitas a este limite máximo de 255 para o valor de n. Podem também existir limites semelhantes quanto aos parâmetros de dimensionamento das variáveis de quadro. Deve-se verificar quais são estes limites consultando o manual fornecido juntamente com a máquina.

Erros de lógica

Como se referiu no primeiro capítulo deste livro, os computadores são incapazes de tomarem qualquer iniciativa sozinhos. Qualquer que seja a linha por nós escrita, se o computador não encontrar nada de errado na sintaxe ou grandeza usadas, ou quaisquer outros desvios, fará aquilo que lhe disserem. Não faz a mais pequena ideia do que possa ser o simples «bom senso».

Se for construído um módulo complicado para calcular qualquer coisa e se esquecer a instrução PRINT, o computador não «parte do princípio» de que era isso que nós queríamos... Pode-se então perder muito tempo procurando erros na equação usada no módulo, quando o verdadeiro problema reside em algo de completamente trivial. Os seres humanos não estão habituados a dar

ordens a um nível «atómico» — elas são de natureza «molecular». Por exemplo, «Abra aqui um buraco com 30 cm de profundidade» teria normalmente como resultado a abertura do buraco... Parte-se do princípio de que o receptor da ordem sabe como se abre um buraco. Para programar um computador para uma tarefa semelhante, mesmo no caso de possuir dispositivos mecânicos periféricos, seria necessário perder muito tempo e gastar muitas linhas só para explicar ao estúpido computador onde deveria pôr a pá para levantar o primeiro pedaço de terra...

O erro talvez mais frequente, consiste em saltar para uma linha errada numa instrução GOTO. A GOTO conduz facilmente a este tipo de erros, sendo por essa razão que os «estruturalistas» a odeiam. No entanto, tentar evitar completamente esta instrução quando se trabalha em BASIC é como tentar beber um copo de cerveja sem tocar com os lábios no vidro! É possível, mas muito restritivo... Tenha bastante cuidado ao escrever os números GOTO, e de preferência verifique-os várias vezes. É muito mais fácil adquirir a certeza no momento em que se escreve a linha do que depois descobrir o erro quando o programa não funciona. Um outro erro muito vulgar, muitas vezes difícil de descobrir, consiste em escrever outra instrução na mesma linha em que se encontra uma instrução GOTO, *depois* desta. Por exemplo:

```
100  A = B + C:GOTO 500:PRINT«VOLTS»
```

A instrução PRINT *nunca* seria executada

As condições no interior das instruções IF/THEN exigem um cuidado especial. Existem aqui duas possibilidades diferentes de erro; resultados inesperados e resultados impossíveis. IF A = 30 AND A = 50 THEN, etc., é obviamente uma condição impossível e deve ter uma palavra OR em vez da palavra AND. IF K > 30 OR K < 80 THEN, etc., é obviamente concebida para apanhar

todos os valores de K superiores a 30 e inferiores a 80. Mas falhará. A palavra a empregar é AND e não OR, senão a instrução deixará passar todos os números inferiores a 80.

Velocidade de execução

Fala-se muito da velocidade de execução. Na maior parte dos programas a questão da velocidade tem apenas um interesse académico. A velocidade de execução é suficientemente grande na maior parte dos casos para poder ser considerada instantânea pelos seres humanos. No entanto, devemos admitir uma vez mais que o BASIC, dado que é normalmente interpretado e não compilado, pode ser um tanto lento. Esta deficiência pode manifestar-se ao executar loops FOR, e em especial quando estes contêm instruções POKE sobre a «screen memory» destinadas a provocar uma ilusão de objectos em movimento. Certas equações com muitos parêntesis uns dentro dos outros também são executadas lentamente. Alguns dos truques que podem ser usados para aumentar a velocidade de execução de um programa já foram mencionados, mas em seguida apresentaremos algumas ideias gerais a respeitar no caso de o problema da velocidade se tornar crítico.

1. Tanto quanto possível, realize as operações aritméticas *antes* de entrar num loop:

```
1000 FOR A = 1 TO N STEP K — C
1010 S = SQR(G + F ↑ 3.2)*A
1020 PRINT A,S
1030 NEXT A
```

Este exemplo ilustra a repetição desnecessária que pode ser incluída num loop. Porque razão não realizar a operação K — C antes de o iniciar, em vez de obrigar o computador a fazer as contas N vezes? A linha 1010 constitui de igual modo um crime de perda de tempo,

porque G + F ↑ 3.2 deveria ter sido calculada antes da entrada no loop:

```
980  T = K — C
990  S1 = G + F ↑ 3.2
1000 FOR A = 1 TO N STEP T
1010 S = S1*A
1020 PRINT A,S
1030 NEXT A
```

2. Utilize variáveis sempre que possível. Isto significa avaliar as constantes uma vez e só uma, no início do programa.

3. Elimine os espaços e as instruções REM desnecessários. Isto permitirá aumentar um pouco a velocidade, apesar de raramente justificar a diminuição de compreensibilidade do programa.

4. Diminua a quantidade de texto a enviar para o visor. Pode haver exageros neste aspecto, acabando-se por ver apenas no visor uma espécie de jogo de abreviaturas semelhante ao que se observa em certos documentos oficiais... Isto pode tornar-se contraproducente devido à necessidade de utilizar uma página de informação extra a fim de explicar as abreviaturas.

5. Utilize subrotinas em código-máquina nas áreas que fazem perder mais tempo. Infelizmente esta cura, do ponto de vista do principiante, pode ser mais prejudicial do que a doença... O código-máquina é ultra-rápido (em certos casos centenas de vezes mais rápido) mas as técnicas de programação a utilizar nesse caso requerem muita paciência e bastante experiência do ofício. Estes métodos estão fora do âmbito deste livro. Os que desejam aprender código-máquina e possuem a necessária natureza masoquista devem consultar um texto especializado no assunto.

Economia de memória

Em geral, as atitudes tomadas para aumentar a velocidade de execução de um programa permitirão igual-

mente poupar posições de memória. Tente manter reduzido o número de variáveis usando, sempre que possível, a mesma variável mais do que uma vez. Esta técnica tem a vantagem adicional de tornar o programa mais fácil de seguir e de emendar. Certas variáveis devem manter-se «exclusivistas» durante todo o programa, mas outras têm uma natureza apenas temporária. Depois de usadas, podem ser definidas como tendo um conteúdo que se tornou inútil.

É fácil tornarmo-nos fanáticos nesta questão da economia de memória. Certas pessoas têm um orgulho imenso em programarem recorrendo a uma quantidade mínima de memória. Se o programa é comprido e a memória é limitada, é de facto essencial realizar economias; mas se pode gastar muito espaço de memória, porquê preocupar-se em economizar?

Como programar

Aprender a linguagem, pensar cuidadosamente sobre o problema, desenhar ou esboçar um fluxograma ou qualquer outro auxiliar sob a forma de diagrama e começar a trabalhar. Tal como muitos bons conselhos, no entanto, este não passa de um conjunto de «clichés»... Mas é de facto um bom conselho, o suficiente para merecer ser discutido com algum pormenor.

Considerando que sabemos já a linguagem, passamos à fase de «pensar cuidadosamente» sobre o problema Um grande perigo neste ponto consiste em correr para a máquina e iniciar uma furiosa sessão de batida no teclado, sem pensar absolutamente nada antes. Este hábito, que é compreensível, não dará certamente bom resultado; assemelha-se a compor música sentado ao piano... O resultado de dividir um problema em passos executáveis separadamente é designado por algoritmo. Vale a pena dispender bastante tempo com os algoritmos. Habitue-se a rejeitar o primeiro; raramente é o melhor. Pense no assunto durante horas, até mesmo dias, mas não se

atire para o teclado até ter definido completamente a sua estratégia de resolução do problema.

A *tática* é formada pelas linhas de programação escritas em BASIC e o ideal seria escrevê-la primeiro no papel. É no entanto provável que poucas pessoas (incluindo profissionais) se dêem a este trabalho porque, no fim de contas, o teclado de um computador é de facto uma máquina de escrever e não vale a pena escrever tudo duas vezes. Depois de cada módulo ter sido colocado na máquina e ensaiado, passe-o para as memórias de apoio imediatamente, não vá faltar a electricidade de repente e perder-se todo o seu trabalho...

Ensaio do programa

Ensaie-o quantas vezes quiser para sua satisfação própria mas não dê grande importância à validade dos resultados. O programador não é a melhor pessoa para experimentar um programa. As mensagens no visor serão sempre óbvias para a pessoa que as compôs; o problema consiste em saber se alguém, funcionando como cobaia as considerará igualmente óbvias... Se o programa pede uma entrada feita pelo operador, a tal «cobaia» é a melhor pessoa para a fazer. Números estranhos como —1, etc., não são considerados muitas vezes pelo programador porque em sua opinião ninguém se lembraria de os comunicar à máquina. Mas esta razão não é válida.

Um programador profissional não produz, ou não deveria produzir, um «estoiro» da máquina. Para discutir isto, necessitamos de três interpretações diferentes do que possa ser esse «crash».

1. Um sistema «estoira». Trata-se da situação catastrófica em que o comando por teclado é perdido. Deve-se quase sempre a instruções POKE que interferem com o sistema operativo. O remédio para isto, como se explicou anteriormente, consiste em «reset» ou desligar e ligar novamente a máquina, se bem que obviamente o programa que se encontra na RAM seja perdido entretanto.

2. Um «estoiro» de comando. Não é catastrófico mas é aborrecido. Significa que o computador saltou para fora do programa e passou a estar disponível para a entrada de ordens.. Estes «estoiros» podem ser causados pelo próprio operador, se este carregar inadvertidamente na tecla STOP. Para evitar isto, descubra qual é o número POKE que inibe a função STOP. Utilize este número num ponto qualquer perto do início do programa, mas não se esqueça de que deve deixar ao operador a opção de stop se o desejar. Este é um outro número POKE que permite recuperar a função STOP.

Uma outra razão para um «estoiro» deste tipo é o uso da tecla RETURN (em resposta a um pedido INPUT) *antes* de dar entrada aos dados pedidos. Em certas máquinas, isto pode provocar um crash a nível de comando. Este «estoiro» pode ser benigno ou não, conforme a natureza do programa. Se for necessário dar entrada a um número reduzido de dados, não será difícil RUN novamente o programa desde o princípio. Em certos programas, no entanto, particularmente nos módulos de Criação de Ficheiro, é francamente aborrecido, depois de comunicar o centésimo dado, descobrir que a máquina «estoira»... Pode ser possível continuar o programa (a partir do ponto onde «estoirou») escrevendo CONT (continue), mas não devemos partir do princípio de que o operador sabe isto. A maior parte dos BASIC «estoiram» nesta situação só quando se carrega prematuramente na tecla RETURN em resposta ao pedido de INPUT de uma variável numérica. Se a INPUT for uma variável de cadeia, considera-se que o operador deu entrada a uma «cadeia nula», pelo que a máquina não causará quebras a nível de comando. Se for este o caso da sua máquina, a regra para evitar este tipo de «estoiros» (chash) consiste em assegurar que todos os INPUTS sejam variáveis de cadeia... Se se pretende uma entrada numérica, pode-se convertê-la usando uma instrução VAL:

```
100 INPUT K$
110 K = VAL(K$)
```

Assim, o caracter escrito no teclado é tratado como uma cadeia no que se refere à instrução de INPUT (não havendo portanto «estoiro») sendo depois convertido numa variável numérica pela instrução da linha 110. De facto é bom adoptar esta regra em todos os casos; dar entrada sempre a uma variável de cadeia, qualquer que seja o tipo de variável de facto necessária.

3. Um «estoiro» por erro. Não existe qualquer desculpa para escrever um programa que de súbito apresente uma mensagem de erro. Se bem que isto constitua de facto um «estoiro» do tipo anterior, merece um tratamento separado. Pode ser sempre evitado desde que se utilize a apropriada validação de entradas, de tal modo que seja qual for o dado indicado pelo operador nunca haja a possibilidade de surgirem mensagens de erro.

Portabilidade

A portabilidade implica a possibilidade de transferir um programa de um tipo de máquina para outro. Uma «portabilidade» absoluta é pouco mais do que um sonho inútil, dado que são tantas as variáveis em causa... No entanto, não há qualquer razão para não tentarmos pelo menos aproximar-nos deste ideal. A primeira coisa a discutir é se esta qualidade é ou não bastante importante. Se apenas pretendemos utilizar os programas na nossa máquina ou em máquinas do mesmo tipo, não é de facto necessário considerar o assunto. Mas há sempre a possibilidade de você querer cobrir os custos do seu sistema computorizado tentando vender programas a uma revista; se for este o caso, é conveniente usar codificações que evitem, tanto quanto possível, palavras-chave ou outras operações que sabe serem apenas possíveis na sua máquina. Por exemplo:

1. Evite as instruções POKE e PEEK a menos que explique na documentação que acompanha o programa, o que pretende conseguir com elas. Isto permitirá

aos utilizadores de outras máquinas realizarem as alterações necessárias.

2. Inclua muitas REM's. Não tenha medo de exceder-se no seu emprego.

3. Evite usar palavras-chave BASIC que são descritas pelos fabricantes como «muito potentes». Isto significa normalmente que não estão estandardizadas.

Existem algumas variantes na sintaxe das palavras de tratamento de cadeias de caracteres, LEFT$( ), RIGHT$( ), MID$( ). As diferenças dizem normalmente respeito ao significado dos parâmetros que se encontram no interior dos parêntesis. Algumas consideram que a primeira posição numérica é o 0, outras que é o 1. Isto significa que a sub-cadeia desejada pode estar errada em uma posição. Algumas permitem o uso de um parâmetro 0, outras apresentam uma mensagem de erro se tal acontecer.

CONCLUSÃO

Este livro é literalmente um guia para principiantes. Deve ser continuado pela leitura de todos os livros mais avançados sobre BASIC a que possa deitar mão. Os registos sobre programação são também bastante valiosos, particularmente no que se refere a porem-no ao corrente das novidades. Constituem além disso um meio útil para trocar ideias sobre programação. Ninguém nesta actividade pode manter-se sozinho; todos aproveitamos com os conhecimentos e a inspiração dos outros. Se tiver lido este livro sem possuir um computador ou ter acesso a um, talvez sinta desejo de o comprar. É um hobby intelectual bastante agradável, e pode conduzir a uma carreira profissional, particularmente se o leitor ainda é jovem.

APÊNDICE A

CÓDIGO DE CARACTERES ASCII

Por uma questão de simplicidade, muitos dos caracteres especiais de controlo não são indicados aqui, juntamente com as letras minúsculas (códigos 97 a 122).

ASC II Código (decimal)	Caracter	ASC II Código (decimal)	Caracter	ASC II Código (decimal)	Caracter
0	Cadeia nula	52	4	75	K
10	Alimentar linha	53	5	76	L
13	Retorno ao início da linha	54	6	77	M
32	Espaço	55	7	78	N
33	!	56	8	79	O
34	"	57	9	80	P
35	#			81	Q
36	$	58	:	82	R
37	%	59	;	83	S
38	&	60	<	84	T
39	'	61	=	85	U
40	(	62	>	86	V
41	)	63	?	87	W
42	*	64	@	88	X
43	+			89	Y
44	,	65	A	90	Z
45	−	66	B		
46	.	67	C		
47	/	68	D	91	[
		69	E	92	\
		70	F	93	]
48	0	71	G	94	↑
49	1	72	H	95	←
50	2	73	I	96	Espaco
51	3	74	J	127	Eliminar

APÊNDICE B

LISTA DE PALAVRAS-CHAVE DO BRASIL

Palavra-Chave	*Objectivo*	*Exemplo*
LIST	Apresenta todas as linhas do programa	LIST
LIST n	Apresenta apenas a linha número n	LIST 50
LIST —n	Apresenta as linhas até n	LIST—450
LIST n—	Apresenta as linhas a partir de n	LIST 670—
LIST n—m	Apresenta as linha entre n e m	LIST 340—570
RUN	Executa o programa	RUN
RUN n	Executa o programa a partir da linha n	RUN 200
REM	Observação ou comentário não executado pela máquina	100 REM *DETERMINAR MAXIMO*
LET	Prefixa uma instrução de atribuição, normalmente opcional	110 LET K = 467.589 110 G = K + S/D 110 G = G + 1
SIN(X)	Calcula o seno de X, sendo este dado em radianos	350 K = SIN(X) 350 B = SIN(X + G) 350 R = SIN(2.37)
COS(X)	Calcula o coseno de X, sendo este dado em Radianos	350 K = COS(X) 500 S = COS(R + SIN(X))
TAN(X)	Calcula a tangente de X, sendo esta dada em Radianos	500 S = TAN(X) 500 S = TAN(X + B*.01)
ATN(X)	Calcula o arco cuja tangente é X. O ângulo é dado em radianos	F = ATN(X) G = S + ATN(1 + X) H = ATN(1 + SIN(X))

Palavra-Chave	*Objectivo*	*Exemplo*
INT(X)	Corta o número de vírgula flutuante em X para o inteiro mais pequeno seguinte	J = INT(X) G = INT(100*X + 0.5) F = INT(SIN(X)*32)
ABS(X)	Converte X para o seu valor absoluto (se X é negativo passa-o a positivo)	F = ABS(X) J = 5 + ABS(X + B — C) G = ABS(G)
EXP(X)	Calcula ex. O símbolo é a base dos log. neperianos, aproximadamente 2,718	K = EXP(X) G = EXP(X) + EXP(Y) F = 5 + EXP(SIN(G — 7))
LOG(X)	Calcula o logaritmo de X na base e.	F = LOG(X) G = 3*LOG(X/F + C)
SQR(X)	Calcula a raiz quadrada de X. X deve ser positivo	R = SQR(X) F = INT(SQR(X) + J)
DIM A(n)	Reserva n posições de memória para o quadro subscrito A	DIM A(30) DIM F(T)
DIM A(n,m)	Reserva n e m posições de memória para o quadro bidimensional A	DIM A(20,5) DIM AS(50,80)
PRINT X	Imprime no visor, começando numa linha nova à esquerda, o valor de X	PRINT X PRINT X/2 PRINT SQR(X + B)
PRINT X;Y	Imprime o valor de X e Y na mesma linha com um espaço entre ambos	PRINT X;Y PRINT X + 4;Y*B
PRINT X,Y	Imprime o valor de X na primeira zona e o valor de Y na segunda zona da mesma linha	PRINT X,Y PRINT K,G/B

Palavra-Chave	*Objectivo*	*Exemplo*
PRINT X;	Imprime valor de X e impede retorno e mudança de linha, garantindo que a próxima impressão é na mesma linha	PRINT X; PRINT Y Estas duas instruções imprimem X e Y na mesma linha
PRINT« »	Imprime os caracteres dentro das aspas, mas não estas	PRINT«ERRADO» PRINT«456/75B» PRINT«.....»
PRINT A$	Imprime o conteúdo da variável de cadeia A$	PINT K$ PRINT D$,X
PRINT	Imprime uma linha em branco	
TAB(N)	Usada no interior de uma instrução PRINT; assegura que a variável seguinte seja impressa a partir da coluna N	PRINT TAB(6)X PRINT Y TAB(13)X PRINT TAB(7)«SIM» PRINT TAB(R/3)X
SPC(N)	Deixa N espaços antes de imprimir a variável seguinte	PRINT SPC(5)X PRINT X SPC(T)Y
GET X	Coloca um caracter único do teclado na variável X (certos BASIC's esperam, outros não, pelo operador)	GET K GET K$
INPUT X	Espera até serem fornecidos dados pelo teclado e se ter carregado na tecla RETURN	INPUT X INPUT D$
INPUT« »;X	Espera, apresenta a mensagem dentro das aspas até entrarem dados e ser carregada a tecla RETURN	INPUT«ENTRE»;X INPUT«AGORA»;D$

Palavra-Chave	*Objectivo*	*Exemplo*
GOTO n	Passar para a linha de número n	GOTO 7650
IF THEN	Usada para permitir acções alternativas. A acção depois de THEN só é executada se a condição é verdadeira	IF A = B THEN 468 IF B = C + D THEN PRINT «SIM» IF A$ = «NAO» THEN G = 3.4: GOTO 570
AND	Usada para impor duas condições numa instrução IF THEN	IF A = 30 AND B = 50 THEN 300 IF S = 5 AND B > 4 THEN PRINT«FIM»
OR	Usada para impor uma de duas condições numa instrução IF THEN	IF S = 5 OR S = 9 THEN 580 IF G$ = «ERRADO» OR K = 56 THEN 400
ON N GOTO	Passa para a enésima linha da série de linhas que se seguem à parte GOTO	ON N GOTO 1000,4500,300,5000 (se N = 1, o programa passa para 1000; se N = 4, passa para 5000)
FOR NEXT	A parte FOR é usada no início de um loop. A NEXT é a demarcação no seu final. A estrutura FOR é: FOR N = S TO F STEP I. N é a variável de loop que se inicia em A, aumenta em incrementos de I e atinge F. A parte NEXT deve ser seguida em certos BASIC's pela variável de loop	FOR N = 1 TO 20 STEP 1 FOR S = 5 TO 56 STEP 0.1 FOR F = T + R TO K*B STEP 4 FOR F = 20 TO 6 STEP —1 FOR N = SQR(T) TO H STEP J/3.2 NEXT N

Palavra-Chave	*Objectivo*	*Exemplo*
GOSUB n	Passar à subrotina situada na linha n. «Chama» a subrotina.	GOSUB 1000
RETURN	Esta instrução deve terminar todas as subrotinas. Passa o comando para a instrução que se segue ao chamamento da subrotina	RETURN
ON GOSUB	Semelhante a ON GOTO mas os números de linha referem-se ao início das subrotinas	ON N GOSUB 1000,3000,5000, 300 ON K GOSUB 450,5780
DATA	Parte da estrutura READ/DATA. DATA é seguida por items de dados separados por vírgulas	DATA 45.6,3000,45, — 8,346.78 DATA LONDRES, JUGOSLÁVIA, PORTUGAL
READ X	Lê para a variável numérica X o items de dados sequencial seguinte	READ X READ J
READ X,Y,Z	Lê para X,Y e Z três items de dados	READ B,J,H2
READ A(N)	Lê para o quadro subscrito A(N) o item de dado sequencial seguinte	READ A(N) READ F$(N)
READ A$,B$	Lê para as variáveis de cadeia	READ A$,B$,G$,H$
RESTORE	Depois de os items de dados serem READ uma vez não podem sê-lo de novo a menos que se use RESTORE para levar o «pointer» de dados a 0.	RESTORE
CLR	Limpa todas as variáveis numéricas para zero, e as de cadeia para «cadeia nula»	CLR

Palavra-Chave	*Objectivo*	*Exemplo*
LEN(A$)	Calcula o comprimento da variável de cadeia (quantos caracteres contém, incluindo espaços)	L = LEN(A$)
LEFT$(AS,n)	Separa os n caracteres da esquerda de A$	B$ = LEFT$(A$,3) B$ = LEFT$(K$,L — N)
CHR$(N)	Descobre o caracter que tem o código ASCII N	B$ = CHR$(N) F$ = CHR$(56)
RIGHT$(A$,n)	Separa os n caracteres da direita de A$	D$ = RIGHT$(A$,4) F$ = RIGHT$(C$,L + 1)
MID$(A$,n,m)	Tira m caracteres a partir do caracter n	F$ = MID$(A$,5,2) F$ = MID$(K$,J — 2,K — B)
VAL(A$)	Descobre o valor numérico da variante de cadeia até ao primeiro caracter não numérico. Se o primeiro não for numérico, o valor é zero	K = VAL(A$) K = VAL(G$) —3 D = VAL(D$) *VAL(F$)
STR$(X)	Converte a variável numérica X para a forma de variável de cadeia	D$ = STR$(X)
ASC(«C»)	Descobre o código ASCII do caracter entre aspas	F = ASC(«Z»)
ASC(«FGH»)	Descobre o código ASCII do primeiro caracter da cadeia	K = ASC(D$)
POKE n,m	Armazena o número m no endereço-máquina decimal n	POKE 214,56 POKE M, (K — Y)

Palavra-Chave	*Objectivo*	*Exemplo*
PEEK(n)	Descobre o conteúdo do endereço-máquina n	F = PEEK(67) D = PEEK(F + L/Y)
RND(n)	Produz um número «aleatório». O significado de n varia nos diferentes BASIC's	K = RND(1) K = INT(K*RND(1) + 1)
DEF FN F(X)	Função definida pelo utilizador. F é o nome da função e X é a variável fictícia usada apenas para definir a função	DEF FN K(V) = V/R + 2 DEF FN T(Y) = R + SIN(Y)
FN F(T)	Chama a função anteriormente definida com o nome F. T é o argumento	K = FN F(7)

ÍNDICE

Este livro acabou de se imprimir
em 1983
para a
EDITORIAL PRESENÇA, LDA.
na
Tipografia Nunes, Lda. — 4000 Porto

Concebido para principiantes, este livro segue dois objectivos primordiais. Primeiro, ir de encontro aos desejos daqueles muitos leitores fascinados pela nova tecnologia e que, pretendendo dominá-la não possuem ainda suficiente experiência neste campo.
Segundo, minorar as faltas que se têm feito sentir de uma obra sobre o BASIC aplicado aos microcomputadores, hoje já tão popularizados.
E como o único método de aprendizagem da programação de computadores é programar, esta obra encoraja o leitor a escrever os seus próprios programas em vez de se limitar a copiar as «obras primas» de outros.

EDITORIAL PRESENÇA / MARTINS FONTES