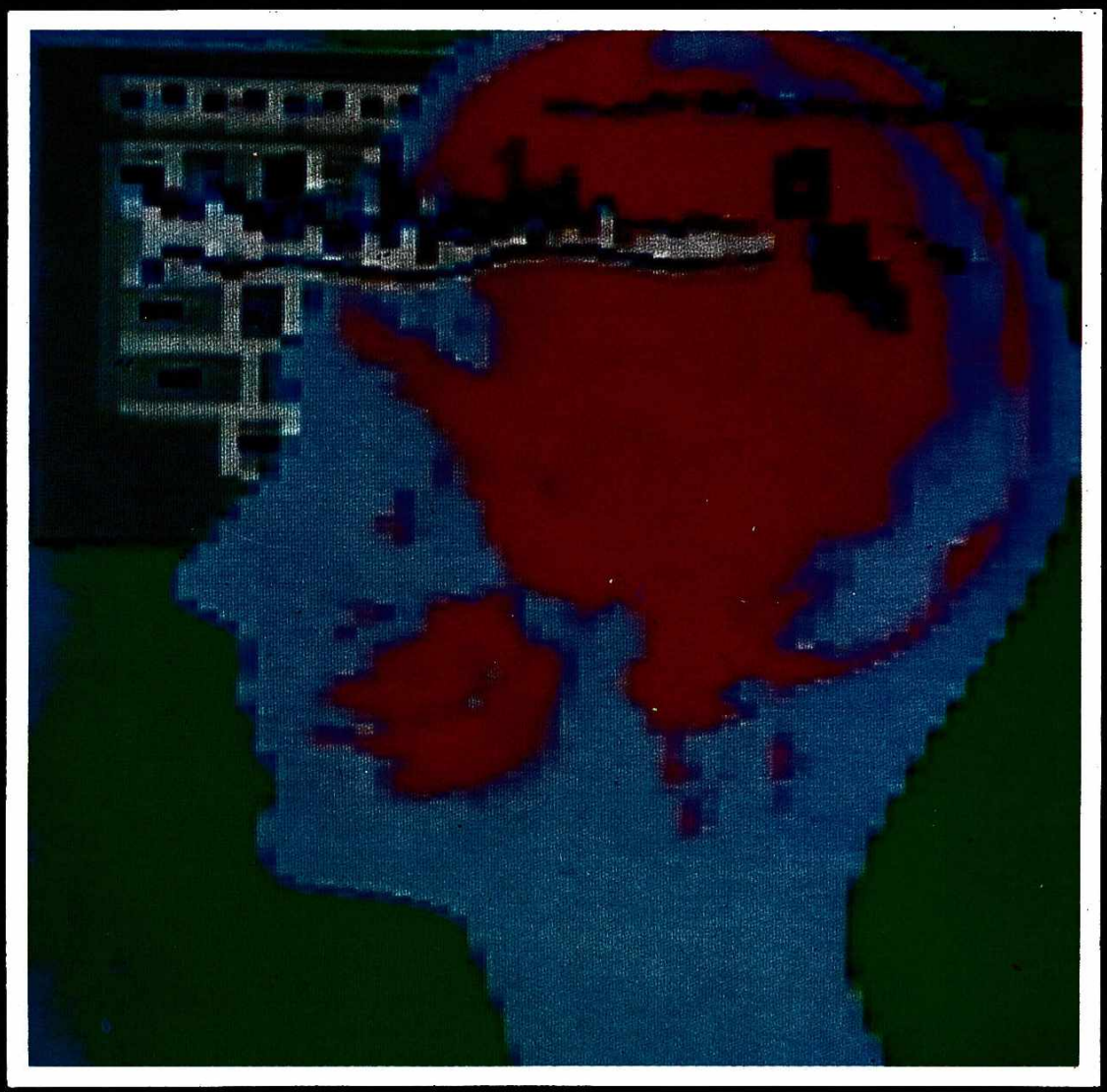
BIBLIOTECA BÁSICA

INFORMÁTICA

PROGRAMANDO COMO SE DEVE 9

algoritmos e outros elementos necessários



SECULO FUTURO

BIBLIOTECA BÁSICA

INFORMÁTICA

PROGRAMANDO COMO SE DEVE

9

algorítmos e outros elementos necessários

SECULO FUTURO

Diretor editor:
M.A.Nieto
Coordenação e supervisão técnica:
Engº Sergio Rocha Paggioli
Tradução:
Ideli Novo
Projeto:
Rainer K.E. Ladewig
Diretor de arte:
Duilio Sarto Fº
Studio editorial:
Auro Pereira da Silva (chefe), Susana M.Amaral Couto (revisão), Luiz Carlos Siqueira Lago (prod. gráfica), Antonio Carlos Martins, Rubens Tadeu Benedito
Fotocomposição, fotolito:
Omnicolor Gráfica e Propaganda Ltda - Rua Dr. Virgilio de Carvalho Pinto, 619 Pinheiros - CEP 05415 - São Paulo
Impressão
Editora Antártica S.A. - Av. Ramon Freire, 6920 (Pajaritos) - Santiago - Chile

ÍNDICE

PREFÁCIO

Neste volume defrontaremos com o delicadíssimo problema da programação. O tema está desenvolvido baseando-se especialmente no BASIC, ainda que tampouco faltarão esporádicas referências ao Pascal, linguagem da qual já conhecemos algo, graças ao número anterior da B.B.I. (Introdução ao Pascal).

Os temas que poderiam ser tratados, com o risco de repetir o dito e repetido em outros textos e nas revistas especializadas, são muitíssimos: desde o algoritmo de resolução mais adequado, até a melhor utilização da máquina da qual se dispõe, passando pelo estilo do programa, visto especialmente sob o perfil de suas características "user friendly" (manejo claro e simples). De fato, já se passaram os dias nos quais um programa servia somente para o uso de quem o havia redigido e que, portanto, em caso de problemas somente podiam aborrecer-se com eles mesmos (e já sabemos que conosco mesmo tendemos a ser especialmente indulgentes).

Entre tantas possibilidades nos decidimos concretizar ao máximo a problemática dos algorítmos resolutivos (ainda que sem nenhuma pretensão de desenvolver um tratado exaustivo). Procedemos através de exemplos de crescente complexidade e limitando-nos a um BASIC muito standard. Assim pretendemos alcançar dois objetivos:

- facilitar a compreensão da problemática que estabelece a realização e colocação em andamento de um programa;
- sair dos limites de um BASIC demasiadamente vinculado às interioridades do próprio microsistema.

Pensamos, sem falsa modéstia, que este tratamento será apreciado por todos os que queiram entender o que quer dizer PROGRAMAR.

Os dois últimos capítulos incluem problemas muito práticos que procuram fazer o mais rápido e eficiente que se possa o "caracol" que é o BASIC interpretado. Os temas e problemas tocados tornam-se muito típicos. Por exemplo, o problema do "garbage collection" ("identificação da informação inútil") das cadeias. É por isto que os conselhos que damos devem ser considerados praticamente de aplicação universal.

CAPÍTULO I

A LINGUAGEM CONFUSA DOS ALGORÍTMOS

É o software uma arte?

Esta pergunta, realmente crucial, afeta hoje em dia não somente aos usuários, mas também a uma grande quantidade de empresas interessadas em transformar esta atividade artesanal em industrial. "The art of programming": assim se intitula o talismã para o programador, uma monumental enciclopédia do software escrita por Knuth, um dos "gênios" nesta matéria. Em seus muitos volumes o autor faz a chamada, explicando-os e classificando-os, a maior parte dos tipos de algorítimos usados na programação. Muitas vezes acontece que principiantes e semi-experts copiam um procedimento de uma revista ou de um livro, que, por sua vez, está recolhido desta fonte sagrada do software.

Não deve escandalizar-se por isto. Além de que isto serve do ponto de vista geral. É muito dificíl, quando se tem um problema particular, encontrar uma receita adequada precisamente para nosso caso. Talvez exista em algum lugar (quase tudo já está inventado neste mundo), mas onde? Nestes casos não resta mais remédio que ajustá-las somente. Por desgraça, frequentemente surge a inquietante pergunta: se, estou disposto, por onde começo?

Na programação, efetivamente, existem ocasiões nas quais a flexebilidade e a liberdade de ação são traduzidas em tragédia, pior inclusive que a do célebre asno que, colocando diante de dois montes de feno, morreu de fome por não saber por qual decidir-se. No caso do software, mais que de dilema deveria falar-se de multidilemas (dilemas super ramificados); as regras (sentenças, loops, GOTO) na realidade são poucas, mas como podem ser combinadas adequadamente? Que nós saibamos, até agora ninguém conseguiu fazer outra coisa que não seja proporcionar exemplos, recomendando que, em casos semelhantes, se proceda "por anologia". Substancialmente, neste livro faremos o mesmo (se dissés-semos o contrário não seríamos honestos).

Em definitivo, tudo isto quer dizer que o software é uma arte? Segundo nosso ponto de vista sim, apesar dos desesperados esforços para produzí-los desta maneira, inclusive, "automática", mediante os mais diferentes métodos. Tomemos como exemplo o caso do Last One, programa que gera uma listagem BASIC baseando-se tão somente nas especificações do problema dadas pelo usuário. Depois de semanas inteiras aprendendo seu uso e empregando-o, quantos são os que se deram conta que a coisa anda sempre naqueles casos nos quais haviam sido ajustados perfeitamente sozinhos, por normais que fossem seus conhecimentos, com menos transportes e obtendo programas mais eficientes, velozes e compactos? Nos casos inéditos ou, o que é o mesmo, em nossos problemas reais, estamos outra vez como ao princípio: tela (ou a impressora) permanecem inativas: necessitam uma idéia, o mesmo que acontece a nós.

Como muitos já sabem, a idéia de partida, essa que pode chegar mediante a intuição criativa, por obra e graça da meditação transcendental, ou por qualquer outro meio por raro que possa parecer-nos é chamado "algorítimo", termo que deriva do nome do matemático árabe Al-Khuwarizmi. Um algorítmo é um conjunto de regras ou formas de atuar para a resolução de um problema.

A figura 1 mostra o esquema geral da gênesis e realização de um programa. Os especialistas oferecem infinidade e variantes deste esquema, mas em todas elas existe um ponto comum: o início é BUSCAR UM ALGORÍTMO RESOLUTÓRIO, este é o ponto mais delicado de todo o processamento.

Contudo, não queremos desanimar aos que se aproximam pela primeira vez a um computador. O software é uma arte, mas também pode ser aprendida; para as pessoas com força de vontade se trata de um desafio excitante e estimulante.

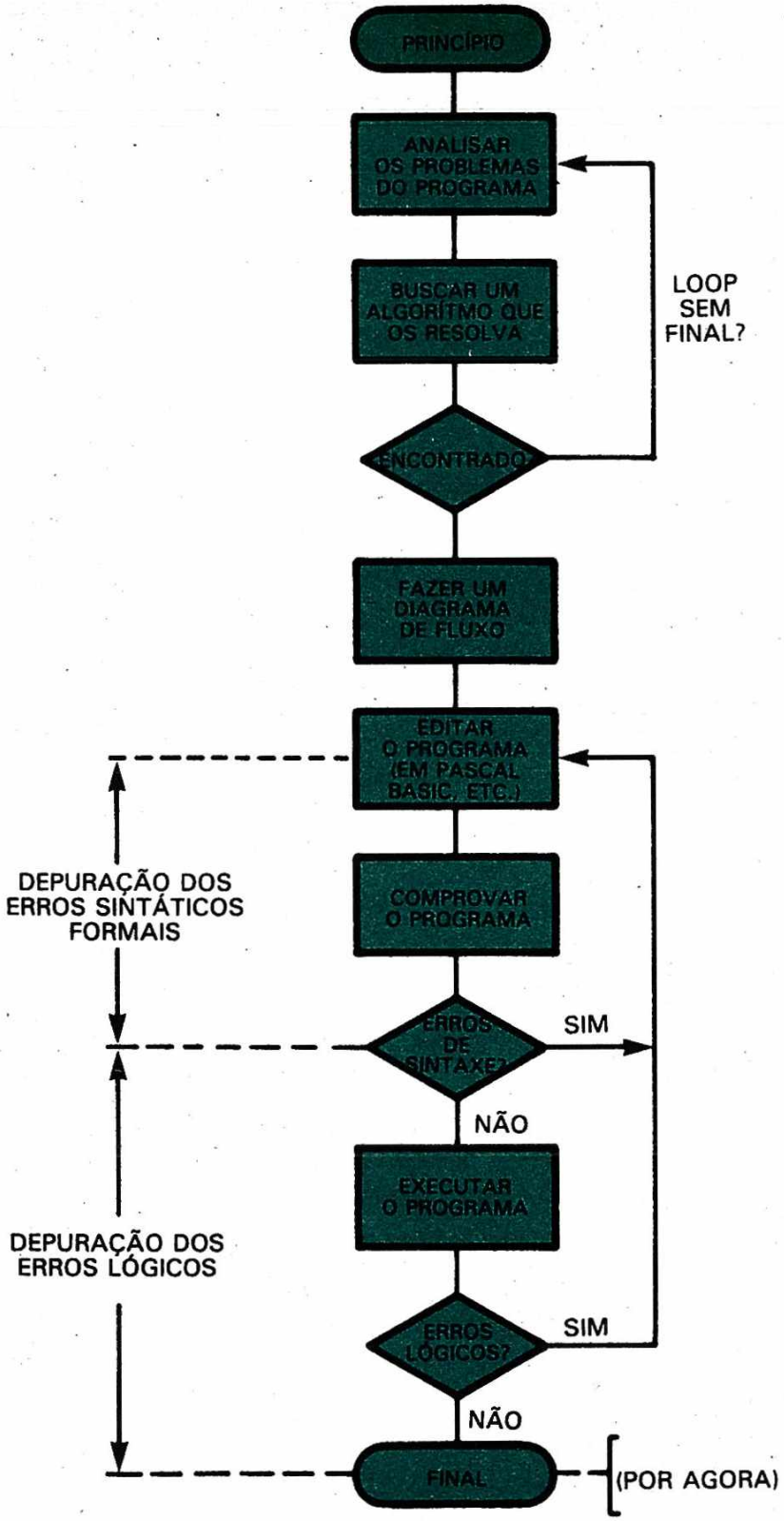


Fig. 1. O clássico ciclo de produção de software (simplificado) subdividido nas fases de implementação e debugging (provas e correção de erros). Para problemas desconhecidos as dificuldades maiores surgem ao princípio: imaginar um algorítmo é uma recomendação muito vaga. São necessárias intuição e experiência.

Dado que não parece possível elaborar uma teoria desde a qual "deduzir"- como ocrre com os sistemas de equações lineares e com a Algebra de Boole - Os casos particulares, mais simples é deixar de discursos e entrar de cheio nos exemplos que nos servirão mais ou menos como meio de estabelecer analogias.

Iniciamos pelos mais simples e clássicos, partindo daquele que tavez é o mais antigo de todos: o algorítmo de Euclides. Serve para falar o M.D.C (Máximo Divisor Comum) dos números naturais, isto é, inteiros positivos. Sejam estes X ou Y. O algorítmo euclidiano é definido desta forma:

passo 1 se X e Y são iguais, o M.D.C. é o valor comum.
passo 2 diminuir do maior o menor.
passo 3 substitua o maior pelo resultado anterior.
passo 4 volta outra vez ao passo 1.

O procedimento descrito, se reflexionado bem, é delicioso. Muitos de nós teríamos resolvido o problema descompondo o número em fatores primos,escolhendo os comuns com o mínimo expoente como nos ensinaram, no colégio. Deve-se levar em conta que Euclides, como Al-Khuwarizmi não sabiam nada de computadores, e ao contrário supor:

- idealizar um processamento por aproximações sucessivas ou, como se costuma dizer em Informática, "iterativo".
- introduzir a idéia que nas linguagens de programação está expressa por sentenças do tipo X = X-Y (onde o sinal = deve ser entendido como "converter-se em").

Na figura 2 é representado o correspondente diagrama de fluxo; o programa em BASIC correspondente seria:

```
 5 DEFINT X, Y, A, B
10 INPUT X, Y:A = X:B = Y
20 IF A = B THEN 50
30 IF A > B THEN A = A-B: GOTO 20
40 B = B-A: GOTO 20
50 PRINT "0 M.D.C. DE ";X;"E ";Y
60 PRINT "E IGUAL A ";A:END
```

(NOTA: nos dialetos BASIC nos quais não exista a instru-

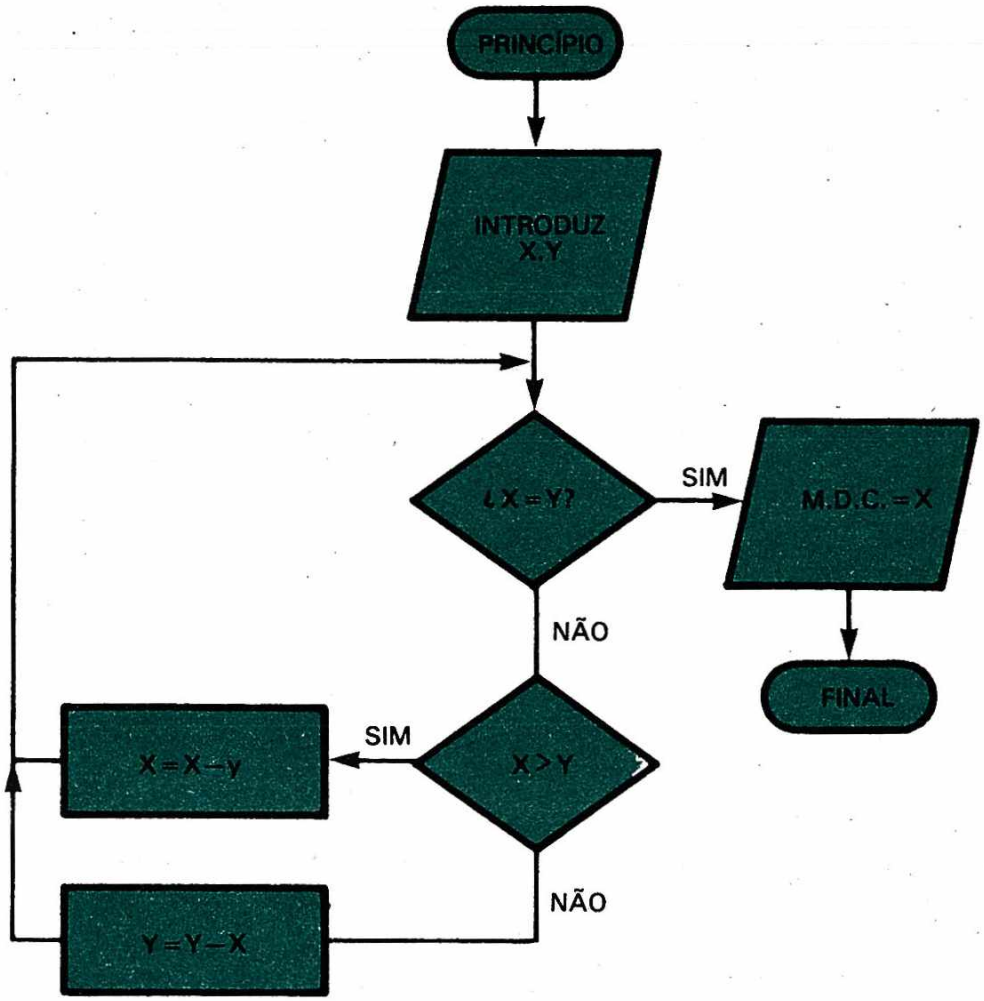


Fig. 2. Diagrama de fluxo do programa que aplica o algorítmo de Euclides para obter OM.D.C.

ção DEFINT para a definição explícita de variáveis inteiras, serão usadas variáveis tipo X%, Y%, ou então nos ajustaremos com as reais).

A comprovação empírica do algorítmo não é difícil. Por exemplo, com o par X = 30; Y = 12 os sucessivos valores de A e B elaborados pelo programa são:

A	B
30	12
18	12
6	12
6	6

Quanto à demostração, esta tem que ver com que o M.D.C. de dois números naturais A e B, com A > B, é o mesmo que o de A-B e B. Podem ser encontradas analogias com o cálculo do quociente inteiro, resultado de dividir dois números mediante a subtração repetida do divisor DSOR do dividendo DDO:

```
 5 DEFINT D
10 INPUT DSOR, DDO
20 R = DSOR:Q = 0:REM Q = QUOCIENTE;R = RESTO
30 IF R < DDO THEN 50
40 R = R - DDO: Q = Q + 1:GOTO 30
50 PRINT R, Q:END
```

Naturalmente, a divisão inteira é utilizada aproveitando o fato de que os computadores a "sabem fazer", e assim as 4 últimas linhas anteriores são reduzidas a uma só:

```
10 Q = INT(DDO/DSOR):R = DDO-Q + DSOR
```

Também o algorítmo de Euclides pode ser feito mais rápido por meio de divisões sucessivas. Este assunto, no entanto, deixamos como exercíicio, igualmente (para os mais avançados) a demostração de como funciona, acrescentando duas variáveis auxiliares "s" e "t", o seguinte (sub) programa. Para que tenha mais variedade está escrito em Pascal e serve para calcular o m.m.c. (mínimo múltiplo comum):

```
PROCEDURE minmultco (x,y:INTEGER);VAR m.m.c:INTEGER);
VAR s,t:INTEGER;
  s: = x;t:s: = y
    WHILE x <> y DO
     IF x > Y THEN BEGIN x: = x-y; s: = s + t END;
        ELSE BEGIN y: = y-x;t: = t + s END;
     m.m.c.: = (s + t) DIV 2 ( * DIV é a divisão inteira * );
  END;
```

Os que tentaram poderão agora comprovar sua solução ao problema de calcular o M.D.C. mediante divisões inteiras repetidas. Aqui está:

```
10 INPUT X,Y:A = X:B = Y
20 IF A   B THEN C = A:A = B:B = C
30 A = A-INT(A/B)*B
40 IF A      O THEN 20
50 PRINT "O M.M.C DE ";X ;"E";Y
60 PRINT "E: ";B
```

Como se vê, consiste simplesmente em obter repetidamente o resto da divisão inteira entre A e B, fazendo que substitua ao maior. Veja como na linha 20 o intercâmbio entre A e B (através da variável de serviço C) permite que em A sempre se tenha o maior do par. Ao final, o M.D.C é encontrado em B, por exemplo:

A	B
18	3
0	

M.D.C = 3

A	B
30	12
6	12
12	6
0	

M.D.C = 6

Portanto conseguimos revisar a Euclides e, ao mesmo tempo, proporcionamos uma variante de seu célebre algorítmo.

Antes de continuar com outros exemplos, nos parece oportuno introduzir uma importante reflexão geral (não chamaremos "princípio", pois seria demasiadamente enfático).

PRIMEIRA REFLEXÃO: o desenvolvimento de um determinado algorítmo (e seu corresponde programa) tem, frequentemente, um carácter evolutivo

Em outras palavras: não existe uma única solução; estudando e provando pode ser encontrada alguma variante, talvez melhor. Algumas vezes a nova solução favorece um determinado fator, por exemplo, a velocidade, em prejuizo de algum outro, por exemplo, a clareza ou o compacticidade do programa. O mais importante ,naturalmente, é que funcione.

Para ligar-nos ao tema vamos propor um exercício muito fácil: encontre um algorítmo para escrever a tabela dos quadrados dos números naturais, utilizando somente a soma (a multiplicação e a elevação à potência estão portanto proibidas, mas seria demasiadamente banal). A solução, por si não a encontram, se nós a dissermos:

```
10 IMPAR = 1:NC = 1
20 FOR N = 0 TO 100
30 PRINT N,NC
40 IMPAR = IMPAR + 2:NC = NC + IMPAR
50 NEXT N
```

Para entender algo bastará dar uma olhada à tabela seguinte:

N	IMPAR	NC
0	1	1
1	3	4
2	5	9
3	7	16

Compreenderá em seguida que o programa trabalha colocando em dia a série de números impares na variável IMPAR, e acrescentando o resultado ao valor anterior de NC. A demostração é baseada no desenvolvimento do binômio de Newton.

$$(N + 1)^2 = N^2 + 2N + 1$$

relação que é lida assim: o quadrado do elemento seguinte a N, isto é, N + 1 (primeiro membro), é igual ao quadrado de N mais o número impar (2N + 1).

Para elaborar a tabela dos quadrados e cubos, somente com a soma se pode partir da seguinte fórmula:

$$(N + 1)^3 = N^3 + 3N^2 + 3N + 1$$

levando em conta que 3N 3 N + N + N.

Na figura 3 foi reproduzido o diagrama de fluxo do programa BASIC que segue:

```
 5 INPUT MAX:CP = 2
10 FOR NP = 3 TO MAX STEP 2
20 LIM = SQR(NP) + 1
30 TEST = 3
40 IF TEST > LIM THEN 70
50 IF NP = INT(NP/TEST)*TEST THEN 80
60 TEST = TEST + 2:GOTO 40
70 PRINT NP:CP = CP + 1
80 NEXT NP
100 PRINT:PRINT "ENTRE 1 E";MAX;"TEM";
110 PRINT CP; "NUMEROS PRIMOS"
```

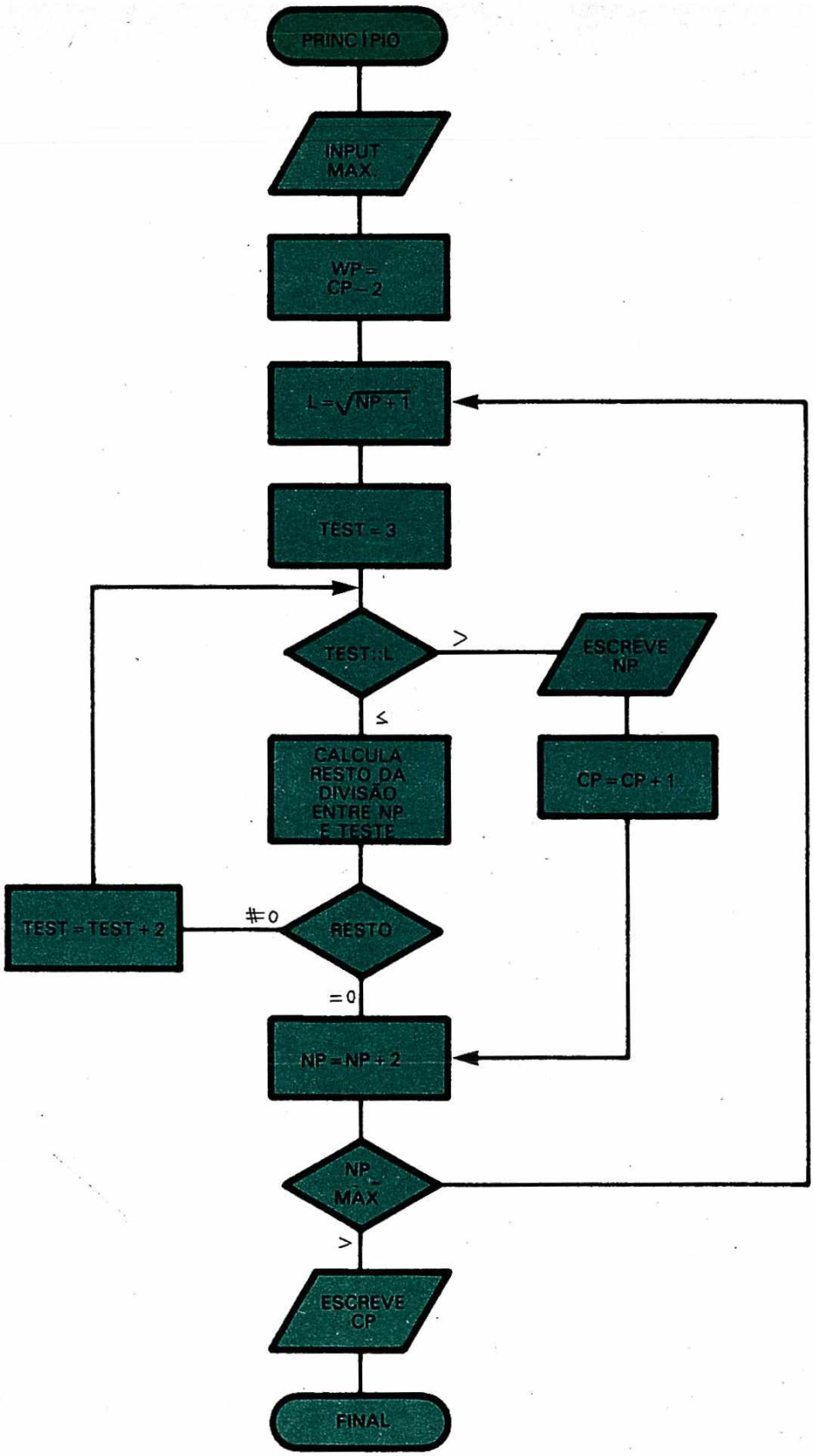


Fig. 3. Procedimento para encontrar e imprimir os números primos presentes entre 3 e MAX. O TESTE de divisibilidade de cada novo NP é feito com a série inteira de números ímpares desde 3.

Trata-se de um programa que gera e imprime os números primos seguintes de 1 e 2. Uma vez estabelecido na linha 5 o valor "MAX" ao qual se quer chegar, começa o loop (linhas 10-80) que recorre a série dos números impares: 3, 5, 7, 9,...Cada vez (linha 50) é realizada a prova da divisibilidade. Com o que? Não com os números primos anteriores, mas com a série dos números impares, desde 3 em diante, série mais ampla e que compreende a dos números primos (todo número primo deve ser impar). Com este fim, da linha 30 sai outro loop (quem o preferir pode realizá-lo também com FOR...NEXT), que gera sucessividade os números impares na variável TEST; se a prova de divisibilidade entre NP e TEST tem êxito, se salta ao próximo NP (linha 80), pois o atual não vale; senão, se volta a provar com TEST + 2.

Alguns pensarão que o problema parece bastante simples, mas entendem a linha 20? Nela é calculado o valor de LIM, sob o qual se faz variar TEST, como a raiz quadrada (por excesso) do NP em curso. De fato, se demostra que é supérfluo seguir adiante com a prova de divisibilidade se test supera LIM (é feito um salto à linha 70); então NP é proclamado primo e, portanto, é incrementado o conta-primos CP (tem que notar que ao princípio CP é colocado para contar o 1 e o 2).

O fato de basear a prova de divisibilidade na raiz quadrada (se 123 resultou indivisível por todos os impares desde 3 a 13, é inútil continuar com 15, 17, etc.), nos leva a nossa "primeira reflexão"; pois a primeira idéia seria fazer coincidir LIM com NP; no entanto, ao colocá-lo como no exemplo, melhora notavelmente a velocidade.

Ainda que o procedimento precedente possa tornar-se obrigatório por exemplo, nas calculadoras de bolso (algumas têm o BASIC mas carecem de memória, e, às vezes também da disponibilidade de vetores) não acontece assim com os computadores pessoais atuais. Vejamos pois a variante que utiliza arrays (em inglês array quer dizer literalmente formação; é usado com o significado de "vetor", tabela e similares).

```
 5 INPUT MAX:DIM PR(MAX):PR(1) = 1:
   PR(2) = 2:K = 3
10 FOR NP = 3 TO MAX STEP 2
20 LIM = SQR(NP) + 1
30 I = 3
40 IF PR(I) < LIM THEN 70
50 IF NP = INT(NP/PR(I)*PR(I) THEN 80
60 I = I + 1:GOTO 40
```

```
 70 PR(K) = NP:K = K+1:CP = CP + 1
 80 NEXT NP
100 PRINT:PRINT "ENTRE 1 E";MAX;"TEM";
100 PRINT CP; "NUMEROS PRIMOS"
```

Em vista de que este novo programa reflete muito acerca do anterior, nos parece suficiente deixar a comparação como exercício útil e limitar-nos a fazer notar que, colocando os habituais 1 e 2 nos dois primeiros postos do array PR dos números primos, CP partirá desde zero, ainda que também agora se atua com os NP ímpares e se faz o teste desde PR(3) em diante. Mas não terminamos. Os que desfrutam pensando, seguramente ouviram falar do Crivo de Eratóstenes.'

Em BASIC poderia ser expresso assim:

```
  5 VERDADEIRO = 1:FALSO = 0:INPUT MAX:DIM PR(2 * MAX)
 10 FOR I = 1 TO MAX:PR(I) = VERDADEIRO:NEXT I
 20 FOR X = 2 TO MAX
 30 IF NOT PR(X) THEN 70
 40 FOR Y = X TO MAX STEP X
 50 PR(Y + X) = FALSO
 60 NEXT Y
 70 NEXT X
 80 FOR I = 1 TO MAX:IF PR(I) THEN PRINT I :CP = CP + 1
 90 NEXT I
100 PRINT "TEM ";CP;"PRIMOS ATE ";MAX
```

O procedimento consiste em estabelecer inicialmente que todos os dados de uma matriz de valores boolleanos de amplitude MAX, são "VERDADEIRO" depois do qual, partindo de 2, serão situados como "FALSO" todos aqueles cujo índice seja múltiplo de cada número primo. A chave está na linha 50: PR(Y + X) que proporciona (para X = 2) os índices 4, 6, 8 (no caso de X = 3 tratar-se-á, ao contrário de 6, 9, 12, etc.). Ao acabar, a matriz-crivo estará reduzida a um coador, com valores "VERDADEIROS" (1) nos elementos primos, ou seja:

```
VALORES: 1, 1, 1, 0, 1, 0, 1, 0, 0,      1, 0, 0, 0, 0...
ÍNDICES:  1  2  3  4  5  6  7  8  9     23 24 25 26 27...
```

Note que, desta vez, os números (primos ou não primos) são os próprios índices. Assim, a impressão e conta dos primos (linha 80) consiste em considerar somente os índices dos elementos "VERDADEIROS". Por último, insistir na elegância derivada da uti-

lização dos booleanos: IF NOT PR ou IF PR são mais expressões elegantes que IF PR = 0 ou IF PR = 1 (nos dialetos BASIC nos quais os booleanos "true" não é 1, bastará estabelecer VERDADEIRO = -1).

O problema dos números primos é uma confirmação de nossa Primeira Reflexão, mas se pensarmos bem nos leva também à

SEGUNDA REFLEXÃO: qualquer algorítmo está intimamente ligado a uma estrutura de dados (adequada).

Esta idéia, verdadeiramente fundamental, é a base, entre outras, do texto de Niklaus Wirth (pai da linguagem Pascal), com o significativo título seguinte:

"Algorithm + data structures = programs"

De fato, é evidente que nos dois últimos exemplos, sem a estrutura "matriz" não teríamos podido fazer nada, ou teríamos tido que ajustar-nos, a duras penas, com mais trabalho e memória.

Vejamos agora outro pequeno exemplo sobre estes conceitos. Ainda com sua banalidade tem um valor histórico: mostra como o software de aplicação evoluciona tanto por aperfeiçoamentos sucessivos como pelo empurrão de sucessos do mundo real. No exemplo, a mudança das leis fiscais. A tabela da figura 4, representa uma possível taxa progressiva de IRPF (Imposto de Renda das Pessoas Físicas) representadas por meses, para o cálculo das retenções. Vamos supor agora que nosso computador não permita tabelas com índice. À primeira vista parece inevítavel recorrer a horríveis cadeias de IF/THEN como as seguintes:

```
200   IF IMP < = 2500  THEN IRPF = 0.1 *IMP
210   IF IMP > 25000 AND IMP = 33333 THEN
      IRPF = 2500 + 0.13 (IMP-25000)
220 IF IMP > 33333 AND IMP < = 41666 THEN
      IRPF = 3583 + 0.16*(IMP-33333)
.....ETC.....
```

A pouca elegância desta solução é indiscutível. Dito seja de passagem, tem que constatar que, inclusive problemas tão banais, NEM sempre encontram solução em uma fórmula, ainda que complicada. Então se requer um algorítmo, um procedimento por passos, seguramente com algumas iterações.

Não seria possível, inclusive com as limitações assinaladas, algo menos rude? Se nos fixarmos, observaremos que o aumento das faixas para impostos de rendas sujeitas a imposto e a das

porcentagens apresenta uma certa regularidade. De fato, até 50.000 o aumento na faixa é de 8.333, e logo de 12.500. Quanto às porcentagens, o aumento é mais regular: 3 em 3 pontos. O truque consiste em aplicar 10% do IRPF inicial (é o mínimo ao qual ninguém escapa) ao inteiro IMPosto e acresecenta um posterior 3% à diferença desta cifra (IMP) com respeito a um termo móvel PARAG. Este último, inicialmente igual a 25.000, é aumentado cada vez em 8.333, até 49.999 (também o Fisco aceita um erro de uma unidade...) e, desde então em 12.500. Tudo isto ficaria em Pascal:

```
PROCEDURE irpf (impon:REAL;VAR taxa:REAL);
VAR parag, incrl,incr2:REAL;
BEGIN
  taxa: = 0.1*impon;
  parag: = 25000;incr1: = 8333;incr2: = 12500;
  WHILE impon > = parag do
    BEGIN
    taxa: = taxa + 0.3 * (impon-parag);
      IF parag < 49999 THEN parag: = parag + incr1
        ELSE parag: = parag + incr2;
      END;
    END;
```

A tradução BASIC é banal:

```
100REM SUBROTINA IRPF
110 TAXA = 0.1*IMP:PAR = 25000
120 I1 = 8333:I2 = 12500
130 IF IMP < = PAR THEN 180
140 TAXA = TAXA + 0.3* (IMP-PAR)
150 V = (PAR <49999):W = NOT V
160 PAR = PAR + V*I1 + W*I2
170 GOTO 120
180 RETURN
```

Previna-se da "finura" booleana contida nas linhas 150 e 160. É apropriada para os dialetos BASIC nos quais o valor lógico "true" está representado com "1"(nos que forem "-1" será necessário substituir, nas linhas 160 os " + " por "-". As variáveis W e V são mutuamente exclusivas (quando uma é 0 a outra é 1, e vice-versa) assim que for verdadeira a condição (PAR <49.999) V = 1 e W = 0 pelo qual, de fato somente se acrescenta o incre-

mento I1 = 8.333 a "PAR". Em seguida, os papéis de W e V são trocados entre si e o que se acrescenta a PAR é I2 = 12.500. É um modo de substituir a construção IF/THEN/ELSE, da qual carecem muitos dialetos de computadores pessoais.

Pelas aparências, o programa anterior poderia parecer bem delineado, mas não é assim. Sua fragilidade fica a descoberto com o contínuo aumento da pressão fiscal, dado, que para bases de impostos mais altas, a regularidade da tabela seria um bênção.

É evidente que, ao final este assunto deveria ser resolvido com o uso de tabelas de tipo "correlacionadas". Em Pascal isto significa que tem que utilizar o tipo "record", subdividido em três campos: RENT, MIN e PORCENT.

Em BASIC nos ajustamos com três matrizes de igual nome e para correlacioná-las (é simples: como o mesmo índice...) teremos que ajustá-las a nós:

```
100 REM SUB IRPF COM TABELAS CORRELACIONADAS
110 DIM RENT (20);MIN (20);PORCENT(20)
120 FOR I = 1 TO 20
130 READ RENT (I), MIN(I), PORCENT(I)
140 NEXT
150 FOR I = 1 TO 20
160 IF IMP ≥ RENT(I) THEN NEXT I
170 IF I = 21 THEN PRINT "ERRO":STOP
180 TAXA = MIN(I) + (IMP(I)-MIN(I)) *PORCENT(I)
190 RETURN
200 DATA 25000,0,0.1,33333,2500,0.13
210 DATA 41666,3583,0.16,50000,4916,0.19
220 DATA.....ETC.....
```

Nas linhas a partir de 200 são carregados os DATA correspondentes às variáveis correlacionadas RENT, MIN e PORCENT. O primeiro trio, como se aprecia em seguida, está composto pelas faixas para imposto de renda RENT = 25.000, pelo imposto mínimo MIN = 0 e portanto por um (porcentagem já dividida por 100) PORCENT = 0.1. Nem estes nem os outros trios dos DATA coincidem com as linhas da tabela incluída na Figura 3, contrariamente ao que poderíamos ter esperado. Por quê? É simples. Para entendê-lo basta seguir o mecanismo da procura em tabelas. De fato, o processo (que, naturalmente necessita uma matriz-chave ordenada), funciona não "por igual" mas "por menor" e, neste caso, quando se encontra com uma situação de IMP menor (ou igual) a RENT

FAIXAS DE IMPOSTO DE RENDA	IMPOSTO MÍNIMO (BRUTO)	PORCENTAGEM %
0	0	10
25.000	2.500	13
33.333	3.583	16
41.666	4.916	19
50.000	6.500	22
62.500	9.250	25
. . . .	. . . .	. . . .

Fig. 4. Tabela de bases de imposto de uma fictícia classificação do IRPF, calculada por meses. Qaundo a base de imposto supera uma faixa e entra na seguinte é aplicado o imposto mínimo (contido na segunda coluna) mais o tanto por cento progressivo sobre a diferença.

tem que aplicar os MIN e PORCENT que, na Figura 4, estão na linha precedente. Portanto, vale a pena mudar estaş linhas tal e como se fez nos DATA. Assim, quando encontrarmos IMP 2500 o MIN será 0 e o PORCENT = 0.1; quando tivermos IMP 33333 MIN será = 2500 e PORCENT = 0.13, etc.

Uma última observação. Em teoria poderíamos ter prescindido do vetor MIN, deixando ao computador a aborrecida tarefa de calcular as distintas bases mínimas, mas é fácil proceder como fizemos. Em qualquer caso, se decidimos que não tenha, deveremos evitar que o cômputo, por exemplo, das 3583 unidades (como soma do MIN anterior, igual a 2500 , mais o 13% da diferença entre PORECENTagens contíguas, MIN = 0.13*(33333-25000) + 2500 = 3583) cada vez que devemos utilizá-lo, e fazê-lo realizar estes cálculos ao príncipio e de uma vez por todas.

O computador NÃO é uma máquina idealizada para realizar pequenos cálculos que já podemos saber de antemão (podemos obter os resultados com uma calculadora no caso de estarmos tão incertos). Usá-lo para isto é não tirar-lhe partido. Se, por exemplo, necessitamos em muitos lugares do programa raiz de 2, será conveniente calcular ao principio uma variável R2 = SQR(2) ou então colocar, onde for necessária, a constante 1.4142135.

Abandonemos aqui os problemas de cálculo. Nos próximos capítulos abordaremos temas que, esperamos, os ajudarão a compreender, como tem que delinear e escrever um programa concretamente.

CAPÍTULO II

LABIRINTOS, CLASSIFICAÇÕES E ESTADOS VARIÁVEIS

Ariadna, seu filho e Teseu

Dos tempos do colégio nos chegam as recordações dos mitos gregos, todos eles carregados de ressonâncias metafísicas e às vezes angustiosas. Tampouco faltam os mistérios, basta pensar em Édipo diante da Esfinge tratando de resolver seu enigma, em tempos nos quais não existiam as revistas de palavras cruzadas.

Quanto a Teseu, sabemos que seu êxito se deveu em parte a seu sex-appeal que lhe permitiu conquistar a Ariadna. E, que foi que lhe deu esta bela donzela? Evidentemente, um algorítmo. Fazendo notar que para sua segurança, Teseu teria que prover do carretel de fio de Ariadna para assinar as passagens por onde passasse, em termos um tanto esquemáticos este algorítmo pode ser expresso como segue:

```
CASE nó OF
     "Minotauro"     ASSASSINO-MONSTRO
     "Ciclo"         REBOBINA-FIO
     "Virgem"        DESENROLA-FIO
     "Ariadna"       STOP
     "Outro"         REBOBINA-FIO

END
```

Tudo isto está escrito em um Pascal "caseiro". São enume-

radas as condições nas quais pode se dar o nó (ou moradia do labirinto da qual saem duas ou mais passagens) e, junto a cada uma delas, a ação a cumprir. O significado desta condições é:

- "Minotauro" na moradia está o monstro
- "Ciclo" o fio de Ariadna se encontra em uma passagem
- "Virgem" nenhuma passagem foi percorrida (exceto aquela de onde se vem)
- "Ariadna" no novo nó está a donzela, ou então estamos outra vez na entrada
- "Outro nenhuma das condições anteriores.

Não é dificil demonstrar (e,a posteriori, ficaria algo intuitivo) que o algorítmo funciona; inclusive no caso de que o monstro seja inacessível (então porque esteja encerrado em uma moradia, ou então porque tenha sido assassinado por outros) e implica além disso, a volta do herói. Para nossos leitores e utilizando termos informáticos, devemos mencionar que esta recuperação do fio, percorrendo seus própios passos, se chama "backtraking" e é desenvolvida mediante uma estrutura de dados denominada pilha (stack) na qual são acumulados os dados para posteriormente extraí-los um a um, começando pelo último introduzido.

Isto poderia sugerir aos mais adiantados a realização de um pequeno programa que simule o jogo do labirinto. Portanto, nós aproveitamos a ocasião para expressar a

TERCEIRA REFLEXÃO: qualquer programa reflete uma situação de estados variáveis

Também poderia ser dito que o programa é uma máquina de estados variáveis. Aqui torna-se obrigatório citar a célebre Máquina de Turing, computador idealizado (como a Máquina de Carnot em Termodinâmica). A cada passo sucessivo de elaboração o computador assume uma determinada configuração ou "estado"; as variáveis que estão na memória tomam então, sucessivamente, valores que, por sua vez (por exemplo, com a presença de instruções condicionais do tipo IF, CASE, ON...GOSUB, etc.), determinam o estado futuro. Em todo este processo o software é um pouco o motor inalterável (pelo menos enquanto for excluído o caso, novíssimo dos programas que se automodificam) e a sequência de instruções determina, a priori, o movimento total. Na prática, isto significa que ao programar tem que desenvolver duas tarefas:

- compreender como deve ser movida a máquina de estados

variáveis;
- prever no programa as regras adequadas.

Ou seja, programar significa prever, ou melhor, escrever, o que deve ser feito sob todos os pontos de vista, sem esquecer que o computador é uma máquina sequencial, isto é, que executa as instruções de uma em uma. A história de Teseu e Ariadna é significativa com respeito a isto.

Para concretizar de forma simples o conceito, examinemos agora um labirinto simplificado como o da figura 1. O objetivo é o seguinte: se trata de realizar um jogo do tipo "aventuras", no qual o usuário tem que chegar ao aposento do tesouro (T) e voltar à entrada. Desta vez, portanto a tarefa de não perder-se (e sem o fio de Ariadna!) é responsabilidade do homen enquanto que o computador é limitado a fazer o papel de notário. A máquina de estados variáveis que utilizamos pode representar-se com o grafo da figura 2.

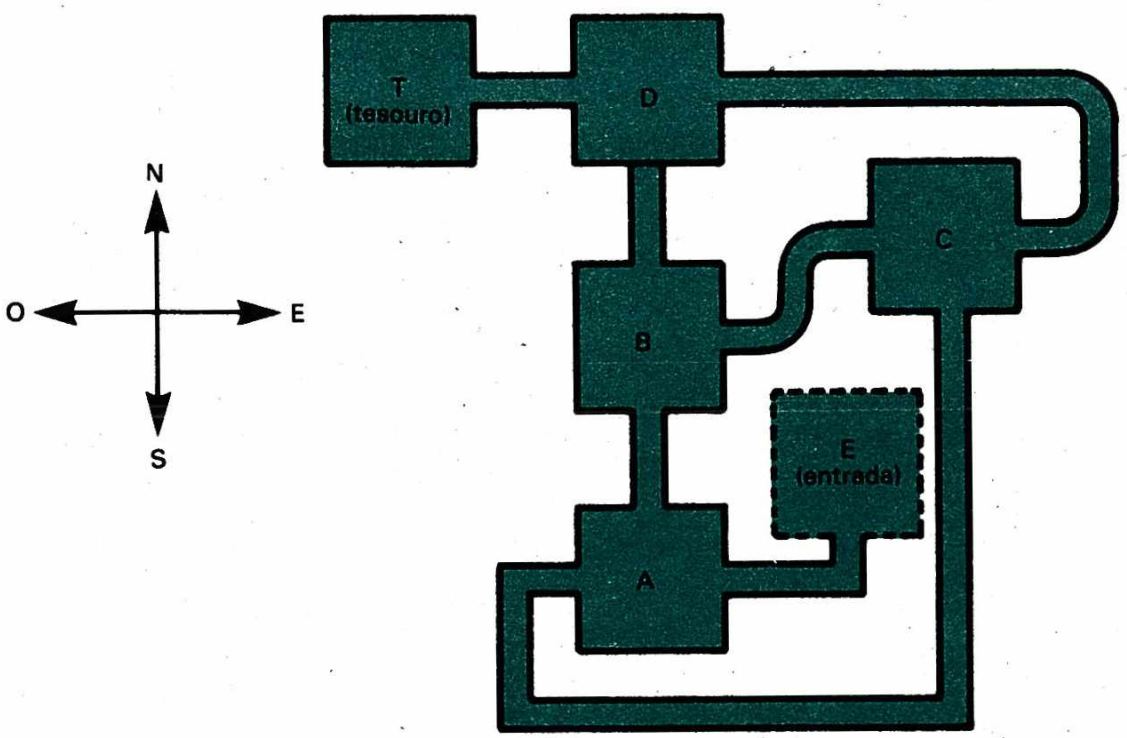


Fig. 1. O simples jogo do labirinto. Para maior simplicidade as portas somente têm 4 orientações; admite a possibilidade de voltar a entrar por uma direção distinta à usada quando saimos, em causa da tortuosidade das passagens

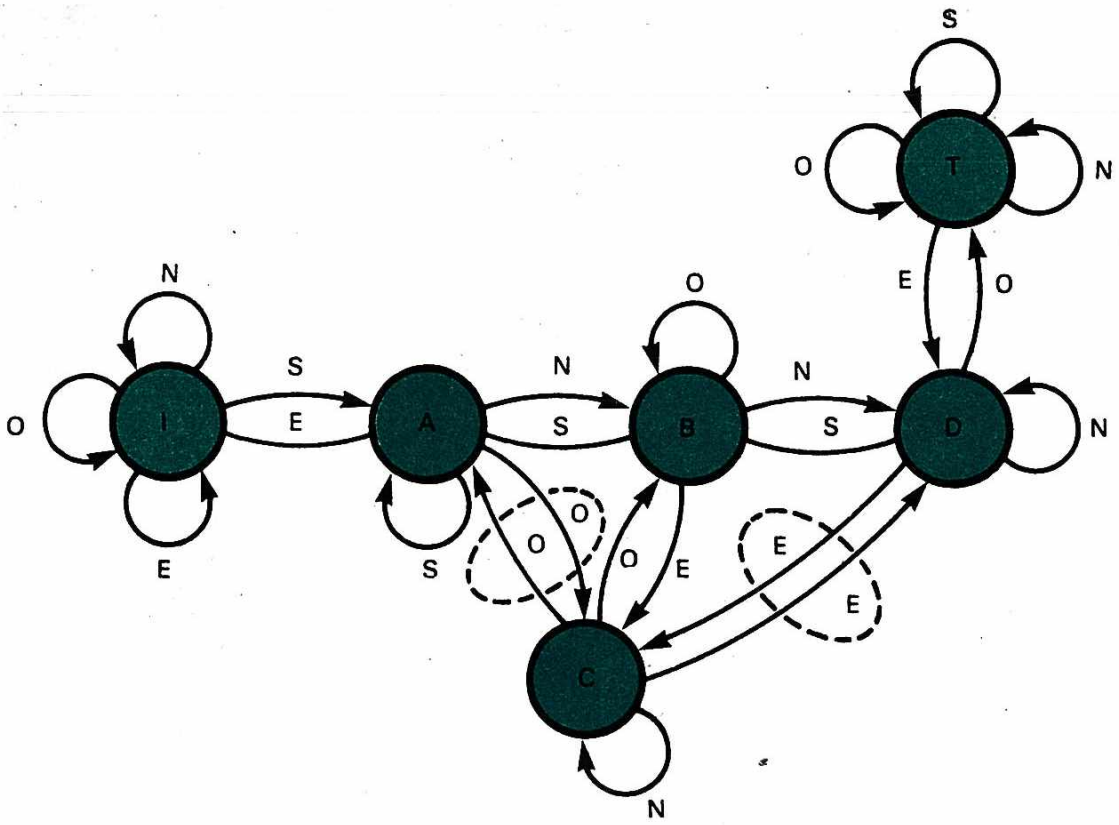


Fig. 2. Grafo que representa a máquina de estados variáveis do labirinto. Cada nó é uma moradia, cada arco (transição) corresponde a uma passagem. Os pares fechados por traços indicam a possibilidade de voltar a entrar em um sentido qualquer.

O que é um grafo? perguntarão alguns de vocês. Antes de contestar queremos precisar que cada aposento pode ter até quatro portas, tantos quantos pontos cardeais, e que alguns acessos podem estar fechados ou murados. Com o dito podemos esclarecer que um grafo representa, com um círculo pequeno (nó) cada estado do sistema; os "arcos" orientados, cada um assinalado com a chave da ação atribuida, expressam as possíveis "transições" de um estado a outro. Assim, na figura citada, cada nó representa um aposento, enquanto que as quatro flechas que saem dali representam as passagens e correspondem às respectivas escolhas feitas pelo usuário (terminam no aposento o qual dá acesso à respectivas pasagem). Quando a passagem está fechada, a flexa volta ao mesmo nó, evidenciando que o estado não mudou. Note também, comparando as figuras 1 e 2, que a flecha de volta nem sempre tem o sentido oposto ao de ida, desde o ponto de vista da rosa dos ventos, por exemplo, de A se vai a C desde o oeste mas desde C se vol-

ta a A pelo sul. Isto depende da tortuosidade de algumas passagens, e deveria aumentar a desorientação segundo os planos do diabólico arquiteto Dédalo.

Neste exemplo a correspondência entre nós-estados e arcos-passagens é estreitíssima, em outros pode ser mais abstrata; por exemplo, podem fazer corresponder a um nó o estado de uma centralzinha automática de telecomonicações e, a um arco, a chegada de um sinal particular, conforme a um certo "protocolo" de comunicação que fará reagir a primeira de uma forma determinada. Rodeando-nos ao tema dos jogos, nos de tipo estratégico cada nó evidencia estados nos quais se dão coisas (como a quantidade de dinheiro, a felicidade, as naves espaciais) possuidas por cada jogador, o número de bens ou planetas conquistados, etc., enquanto as escolhas (flechas orientadas), estão ligadas às ações decididas (como o envio de naves a outros planetas, a compra de ações da Telefônica ou similares, etc.). O jogo pode ser complicado à vontade, mas o conceito é em essência, idêntico.

Ordenação por "bolhas"

Vamos agora deixar um tempo de reflexão para que os leitores pensem sobre o anteriormente exposto. Somente comentaremos antes, para os mais inquietos e impacientes, que um grafo do tipo que vimos é equivalente a um correntíssimo diagrama de fluxo e que, em geral, cada flecha orientada pode ser substituída por um IF THEN...(pelo menos em teoria: antecipamos que existe uma solução mais elegante, mas não diremos mais nada). Nesta espécie de intervalo nos exercitamos no tema da Primeira Reflexão, relacionada com o carácter evolutivo do software.

Tomemos pois o diagrama de fluxo da figura 3, referente à clássica ordenação pelo "método da bolha". (em inglês "bubble sort") aplicada a uma matriz de números A. O método é denominado assim porque as translações que sofrem os dados recordam o movimento ascedente das bolhas em líquido. O programa correspondente em BASIC seria como segue:

```
 5 INPUT N:REM O usuario especifica o número de elementos
10 DIM A(N)
20 FOR I = 1 TO N
30 A (I) = INT (RND(1)*1000 + 1):PRINT A(I);"   ";
40 NEXT I
50 FOR I = 1 TO N-1
60 IF A(I) < = A(I + 1 ) THEN 80
```

```
 70 C = A(I):A(I) = A(I + 1): A (I + 1) = C = :W = 1
 80 NEXT I
 90 IF W THEN W = 0:GOTO 50
100 PRINT:PRINT
110 FOR I = 1 TO N:PRINT A(I);"   ";NEXT I
120 END
```

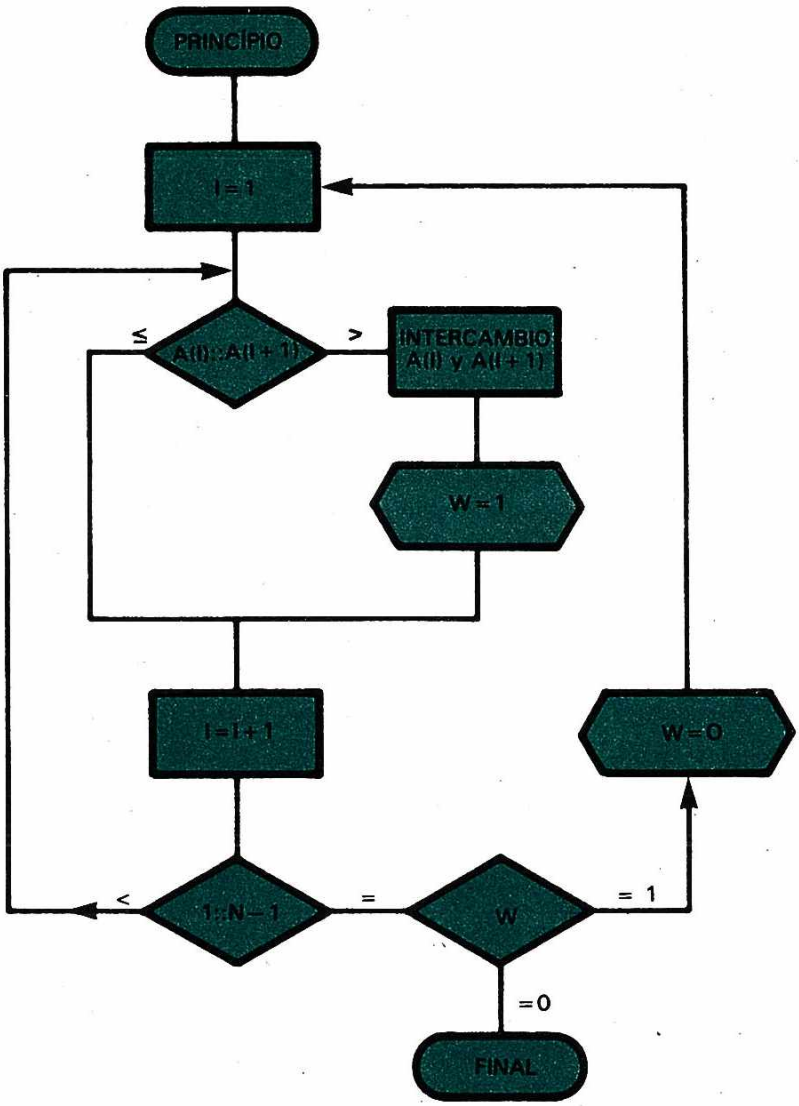


Fig. 3. Diagrama da clássica "ordenação por bolhas" (Bubble Sort).

As linhas 30 e 110 imprimem os números da listagem (gerados ao acaso no campo dos inteiros desde 1 até 1000) antes e depois da operação. O ordenamento (ou"reordenamento") com al-

gorítmo das bolhas consiste, como talvez alguém já sabia, em examinar de dois em dois os elementos contínuos, de índice I e I + 1. Se estão deslocados (aqui a hipótese é que se deseja uma ordem ascedente) muda-os usando uma variável auxiliar C e além disso é ativa uma variável booleana W (tenha presente que em alguns dos dialetos de BASIC o valor lógico "verdadeiro" corresponde a-1, em vez de a 1). Tudo isto está na linha 70. Ao acabar o ciclo FOR/NEXT é checado W e se está em "on" volta a iniciar o ciclo para dar outra passada de possíveis mudanças. Se ao contrário , W é 0 quer dizer que cada elemento estava seguido por outro valor maior, pelo que tudo está correto e o ordenamento para.

Para estudar outras variantes e,raciocinando, pode chegar a soluções alternativas, possivelmente melhores, propomos aqui dois exercícios. Para que sirva de "aquecimento" iniciaremos com uma pergunta muita simples: é o desviador W verdadeiramente indispensável? Resposta: em absoluto. Na linha 70 é suficiente substituir W = 1 por um simples GOTO 50 (e, naturalmente, eliminar a linha 90, que agora torna-se inútil). Isto equivale, no entanto, a ter que iniciar desde o princípio cada vez que é feito um intercâmbio, de forma que, quando se vai do NEXT, tudo esteja em seu lugar.

Aqui os dois exercícios mencionados

1• realize o programa usando dois índices distintos (I e J), fazendo rodar mais rapidamente o segundo que o primeiro (portanto, dois ciclos FOR/NEXT aninhados um no outro) e comparando sistematicamente A(I) com A(J) para intercambiá-los se o primeiro é maior;
2• comparando sempre elementos adjacentes, mas, quando é encontrado um par fora do lugar, realize uma série de intercâmbios para trás, até chegar a um par bem colocado, voltando então a varrer para frente "desde o ponto no qual havíamos ficado".

O importante é verificar se, com estas variantes, se consegue ganhar algo de velocidade. De fato, a ordenação de bolhas se torna muito lento ao aumentar o número de elementos. Para ser mais exatos, é demostrada que a média de tempo de uma ordenação deste tipo cresce com o quadrado da dimensão (ou "potência") do conjunto, porque dobrando-o são produzidos tempos quádruplos. A primeira sugestão que lhes damos é a de tentar aplicar, de alguma forma, o princípio do divide e vencerás. A receita é simples:

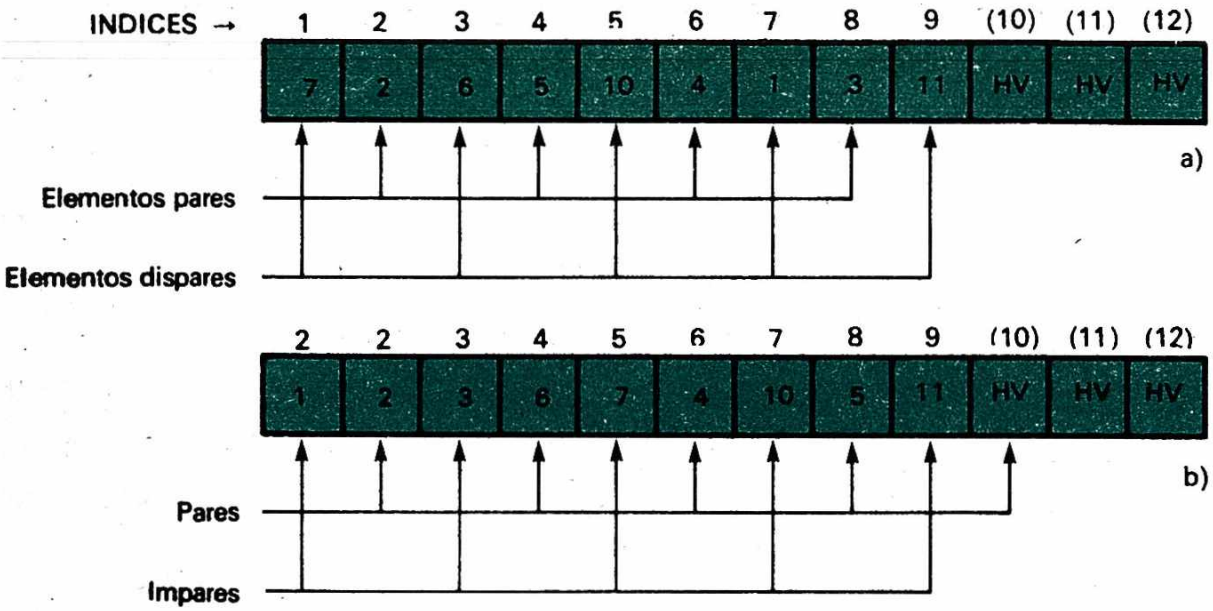


Fig. 4. Situação da matriz que deve ser reordenada, antes a) e depois b) da semi-ordenação dos elementos compostos pares e impares. Note o "tampão" HV acrescentado no décimo lugar para o número de elementos

1) faz-se a classificação de duas metades em matrizes distintas;
2) realiza-se a fusão das duas semimatrizes reordenadas.

Este processo está ilustrado na figura 4. Em a) é representada a subdivisão escolhida (entre elementos de lugar par e impar). Em b) é representada a situação depois dos reordenamentos parciais (de bolha) sobre os números originais (como em a). Note também que, sendo neste caso a potência igual a 9, foi acrescentado um décimo elemento de valor HV. Serve para quadricular as contas, fazendo assim que a semimatriz par tenha cinco elementos, igual à impar. Mas, o que significa HV? Vem do inglês "High Value" (valor alto); é utilizado como "tampão" ao final de um conjunto (ou file) e tem que assumir um valor maior que o mais alto possível do conjunto. Em nosso pequeno exemplo são utilizados valores de 1 a 1000, pelo que bastará colocar HV = 9999, enquanto que no caso de dados alfabéticos será suficiente com HV$ = "ZZ" (melhor ainda, o maior número de "Z" possíveis) para estar tranquilos.No caso de uma classificação decrescente se falará do correspondentes LV (Low Values- por exemplo-999...9).Mas vejamos a listagem:

```
5 HV = 9999
10 INPUT N:DIM A(N + 3), B(N + 1)
20 FOR I = 1 TO N:A(I) = INT (RND(1)*1000 + 1):NEXT I
30 IF INT (N/2)*2   N THEN N = N + 1:A(N) = HV
40 A(N + 1) = HV:A(N + 2) = HV
50 GOTO 120:REM Salta a ROTINA 60, de classificação de bolha
60 FOR I = X TO Y-1 STEP 2
70 IF A(I)   = A(I-1) THEN 90
90 NEXT I
100 IF W THEN W = 0:GOTO 60
110 RETURN
120 REM Reordenação dos elementos pares
130 X = 2:Y = N: GOSUB 60
140 REM Reordenação dos impares
150 X = 1:Y=N-1:GOSUB 60
160...(Segue mais adiante)...
```

Na linha 30 é feito um teste de paridade: se, por exemplo, N = 9 a INT (N/2)*2 nos dá 8 <9 porque é acrescentado um elemento de valor HV = 9999. Quanto à classificação por bolha, está descrita sob a forma de uma subrotina "paramétrica" (desde a linha 60à110).Os parâmetros são as variáveis "X" e "Y" (valores iniciais e finais de submatriz); à parte do STEP 2 (que é óbvio, já que se tem que trabalhar com os pares ou com os ímpares) tudo continua como vimos. É nas linhas 130 e 150 de onde os já citados parâmetros são fixados para que a subrotina da linha 60 faça seu trabalho de reordenação dos elementos pares e impares; fazendo isto, conseguimos escrever as instruções adequadas uma só vez. Uma vez chegados a este ponto devemos realizar a fusão das duas semimatrizes em uma única matriz B. Eí-la aqui:

```
160 REM Fusao (MERGE)
170 X = 1: Y = 2
180 FOR Z = 1 TO N
190 IF A(X) > A(Y) THEN B(Z) = A(X):X = X + 2:GOTO 210
200 B(Z) = A(Y):Y = Y + 2
210 NEXT Z
220 FOR I = 1 TO N:PRINT B(I);"   ";:NEXT I
230 END
```

A fusão é banal. Pressupõe dois arquivos de partida (varridos com índices "X" e "Y") ordenados ambos do mesmo modo; procede sempre com ordenação ascendente, usando a matriz que tem

o elemento menor e aumentando o índice deste. Quanto aos "tampões" (outros os chamam "sentinelas"), servem para manter o sentido da comparação entre elementos A (x) e A (y), até o final. Desta maneira, ao alcançar o elemento valor 9999 em uma semimatriz, o programa completará automaticamente o carregamento dos elementos residuais da outra, todos inferiores a 9999. Se não tivéssemos adotado o critério dos tampões , a casuística teria sido bastante complicada; basta examinar a figura 5 para acreditar; em a) e em b) foram representados os fluxos da fusão com e sem "tampões". A figura demonstra que - ainda que exista gostos para tudo - aqueles que preferem evitar situações demasiadamente complicadas, também poderiam seguir adiante.

Falando de gostos, é muito provável que aqui alguém pergunte: que necessidade havia de subdividir os elementos em pares e impares? Não era mais simples pegar simplesmente a primeira e a segunda metade? Efetivamente, mas fazendo isto, o truque dos tampões não poderia ter sido aplicado, a menos que destacássemos um lugar à esquerda à metade da formação: frequentemente adianta para quem persegue fins de velocidade!

Depois de experimentar a redução de tempo conseguida com este método (apesar do acréscimo da fusão, com uma dezena de elementos já se nota que é um processamento cuja escolha depende linearmente do número de elementos em jogo) recordaremos rapidamente que a classificação, junto à fusão e ao "search" (procura de um elemento determinado), constitui um dos capítulos mais densos da informática. Destacaremos duas questões que daqui são deduzidas:

- o caráter evolutivo do software e dos algorítmo;
- a importância de uma estrutura de dados adequada.

Quer dizer, é uma repetição das que chamamos Primeira e Segunda Reflexão.

Em relação com a primeira, animamos aos mais adiantados a afiançar-se na criação de outros algorítmos deriváveis dos primeiros.

Outros dois tipos de ordenação

Para completar o tema, no mínimo é obrigatório a alusão a outros dois tipos de ordenação "clássicos": a classificação por mínimos e a classificação rápida. O primeiro é parecida com a "bolha" mas é um pouco mais rápido; consiste em varrer ao primei-

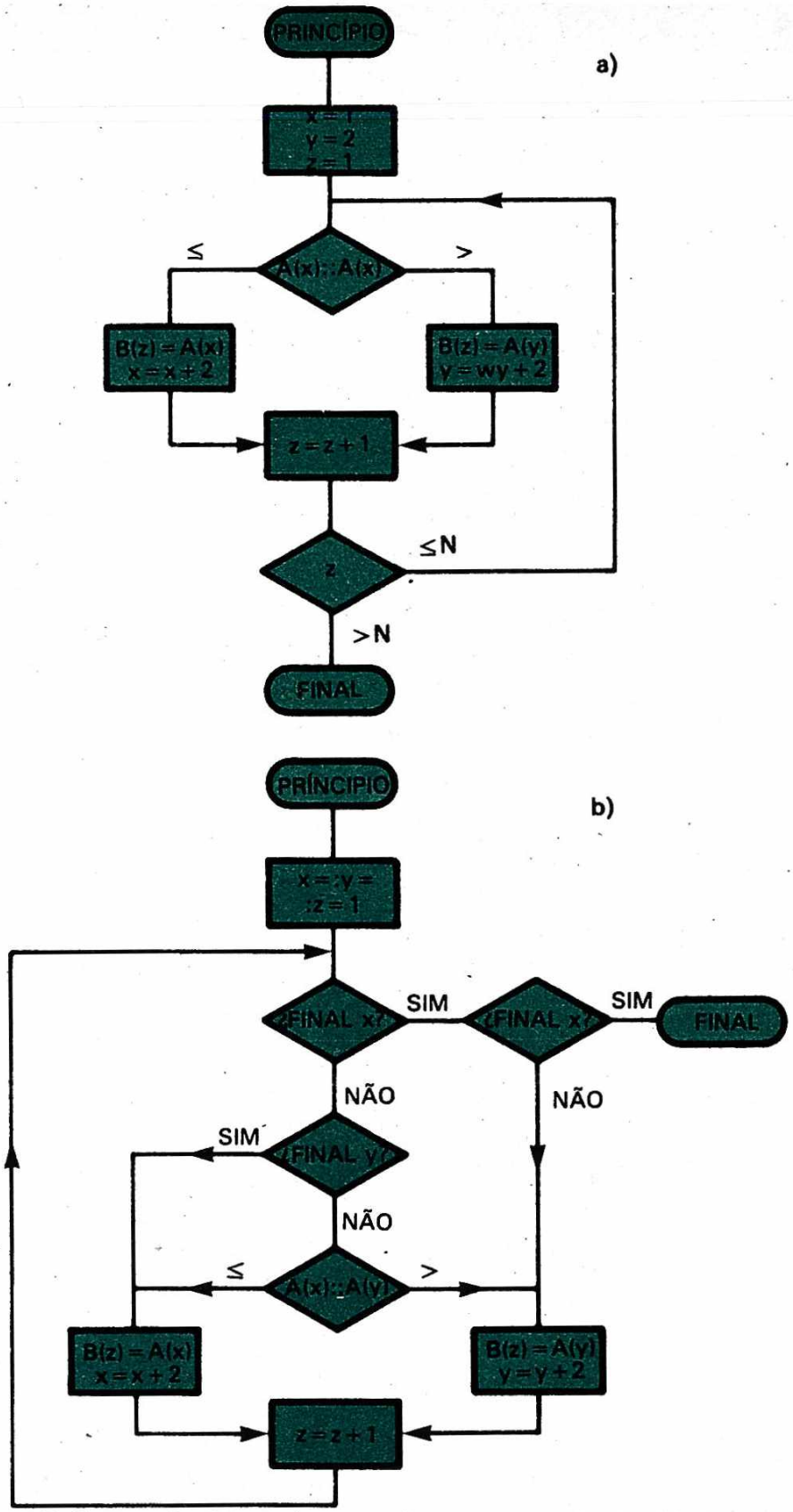


Fig. 5. Fusão de duas matrizes. A primeira solução (a) é realizada mediante a adoção dos tampões HV, e é mais simples que a outra (b) na qual se tem que ter em conta as distintas possibilidades de um final parcial para cada matriz.

ro giro todo o conjunto, encontrando o mínimo, que será colocado no primeiro lugar. Depois o procedimento com os N-1, N-2...2, elementos sucessivamente residuais.

Quanto à classificação rápida, deve seu nome ao assombro de seu inventor Hoare, ante sua velocidade. O procedimento "P" consta dos seguintes passos:

- encontrar o elemento do centro da matriz: seja X seu valor;
- buscar desde o primeiro elemento em diante um elemento maior que X;
- buscar desde o último para trás, um elemento menor que X;
- mudar o lugar dos dois elementos encontrados;
- continuar até que (da direita ou esquerda) se chegue ao elemento central.

A verdadeira classificação rápida consiste na aplicação reiterada de P na matriz inteira, depois em suas duas metades, nas várias metades da metade e assim sucessivamente, até chegar a porções de tão somente dois elementos. Complicado? Bom, é por isto que aconselhamos algum texto especializado para apronfundar nela, enquanto que, para exercitar-nos, nos limitaremos à classificação dos mínimos, que é mais fácil. De todas as formas, os que souberam se apronfundar na matéria, darão conta que, sendo teimosos, com as matrizes se pode chegar longe até certo ponto. Somente pensando em novas estruturas se pode tentar otimizar outras funções, entre as quais estão a de pôr em dia ou o cancelamento de um elemento, ou a soma de um novo lugar adequado. Colocamos aqui um exemplo (figura 6) aludindo à estrutura de árvore binária. Nela cada "nó" contém um dado e é "pai" de um nó esquerdo menor e de um nó direto maior. O parentesco está fixado pelos ponteiros correspondentes, que são campos de cada nó que indicam a colocação dos filhos direto e esquerdo. Esta estrutura torna particularmente fácil o ato de "pôr em dia". Por exemplo, ao acrescentar um dado, não é necessário, como com as matrizes, reordenar toda a formação, perdendo assim um montão de tempo; basta "visitar" a árvore para saber onde tem que pendurar o novo filho; na figura 6b) é mostrada, com linha tracejada, a visita que tem que fazer para colocar um novo dado (13) que tem de ser acrescentado.à árvore de a).

Voltando ao labirinto

Mas desçamos das árvores e dos frutos que podiam ser colhi-

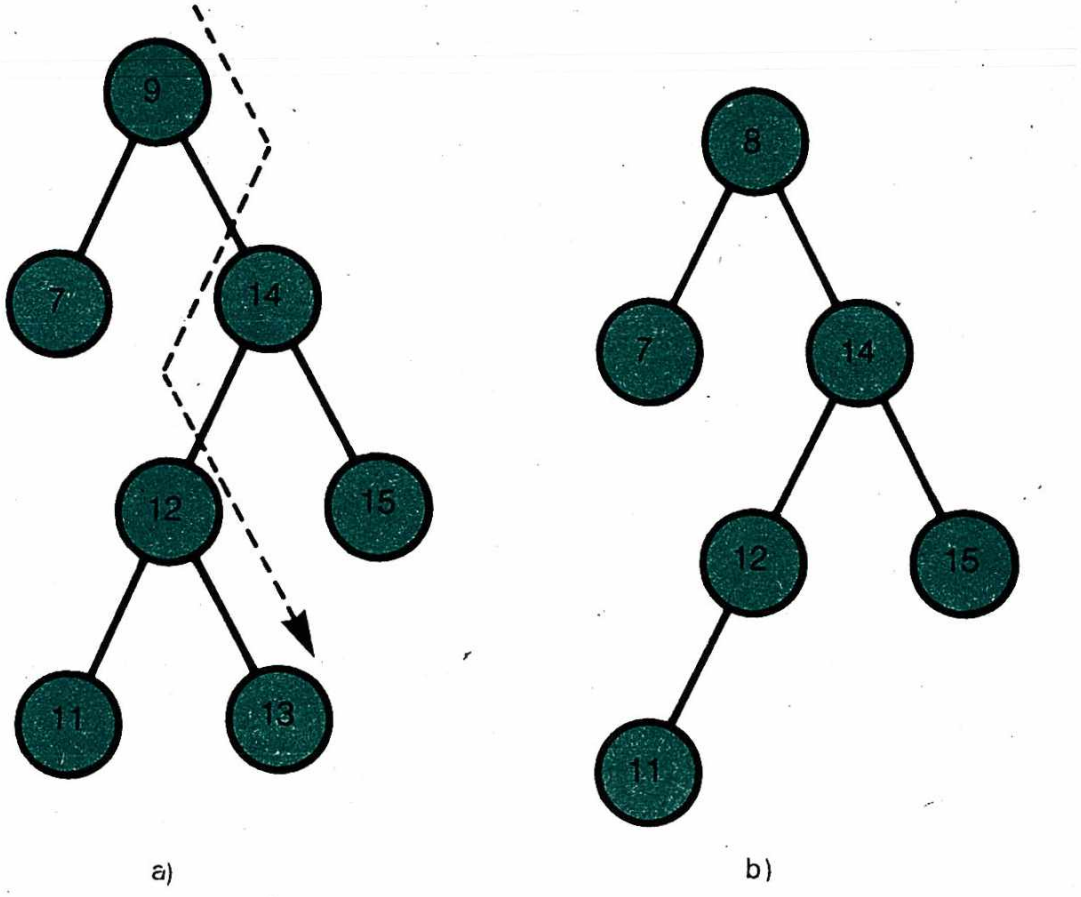


Fig. 6. Estrutura de dados geral em uma árvore binária. Os elementos maiores e menores são os "filhos" direito e esquerdo de cada nó (utilizando ponteiros). Em b) é observada "a visita" que tem de ser realizada para encontrar a colocação de um novo dado (13).

dos de suas frondosas e intricadas ramificações. Teram notado que nos exemplos do "intervalo", não traimos a comparação da Terceira Reflexão, a da máquina de estados variáveis. Na realidade apareceu,mas subtendida, intrínseca a mesma natureza de cada algorítmo. Ao contrário, em problemas como o do labirinto, a idéia da máquina de estados varíaveis, mais que útil é indispensável. Aqui nos propomos misturar adequadamente a Terceira Reflexão com a Segunda.

Dado que já desenhamos o grafo com os estados e suas possíveis transações, a pergunta que devemos fazer é: que estrutura de dados pode ser mais conveniente para descrever nossa máquina de estados variáveis e seu desenvolvimento? Os apressados e os principiantes provalvelmente o resolveriam assim (expressando-o em pseudo-Pascal):

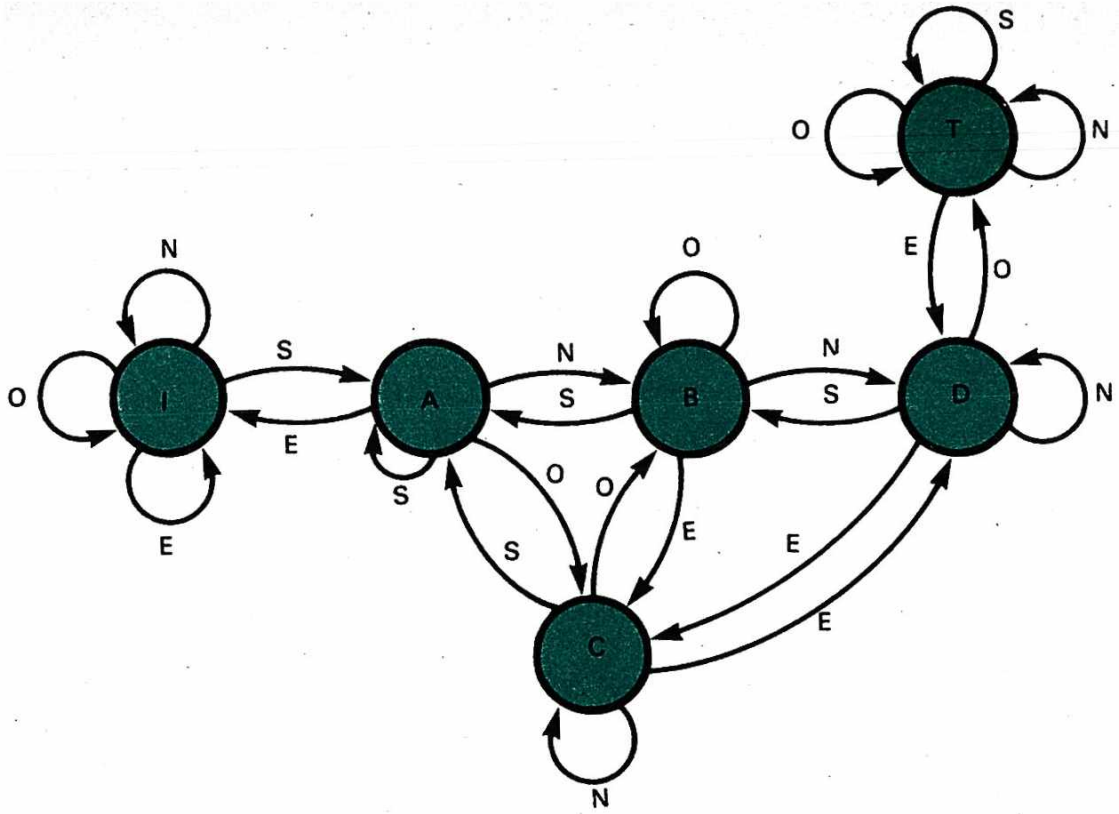
```
.....
IF moradia-A AND resp: = ' N ' THEN
moradia-B
ELSE IF moradia-A AND resp = ' S ' THEN
WRITE ('não existe caminho')
•
•
ELSE IF moradia-B AND resp = ' N ' THEN
moradia-D
.....
```

Os mais refinados talvez adotariam uma construção CASE, mais elegante, mas seria igualmete um biscato, ainda que funcionasse, ao menos por dois bons motivos:

- enquanto o labirinto se torna um pouco complicado, o comprimento e legibilidade do conjunto vai sendo levada pela corrente;
- algo contudo mais trágico: se demônio nos faz acrescentar uma só moradia mais, nos arriscamos a ter que escrever tudo desde o princípio outra vez.

Em definitivo: -é necessária uma verdadeira estrutura de dados. A que propomos está ilustrada na figura 7, onde, por comodidade, voltamos a desenhar nosso grafo. O miolo da questão é a tabela de duas dimensões (ou "matrizes", como normalmente é chamada) denominada UNI (por UNIões), cujas filas representam as diferentes moradias, enquanto que as colunas estão asssociadas aos 4 pontos cardeais. Por exemplo, na intersecção da linha 3 com a coluna 2 é encontrado o valor UNI (3,2) = 2, que indica que se nos encontramos na moradia 3 (a B do grafo) e escolhemos a porta situada ao sul chegaremos à moradia 2 (a A do grafo). Ao paciente leitor será fácil encontrar todas as outras correspondências, evidentes nos diversos compartimentos. Em essência estas contém "ponteiros"; obviamente, um ponteiro 0 corresponde às flechas do grafo fechadas sobre si mesmas (portas muradas). Quanto aos vetores alfanuméricos S T $ e DIR$ servem, o primeiro para associar as direções 1, 2, 3 e 4 na ordem: N, S, E e O, (ainda que também poderiam ser previstos os nomes inteiros) e, o segundo para atribuir um nome de fantasia - "ENTRADA", "QUADRA", "SALA TESOURO", etc. - aos diferentes ambientes do mini-labirinto.

Não nos resta agora mais que voltar a formular o problema: redigir um programa que, tomando as escolhas do usuário (N, S,



		LEE					HAB$		DIR$
(E)	1	0	2	0	0	1	A entrada	1	N
(A)	2	3	0	1	4	2	O salão	2	S
(B)	3	5	2	4	0	3	As cavalariças	3	E
(C)	4	0	2	5	3	4	A cozinha	1	O
(H)	5	0	3	4	6	5	O compartimento		
(T)	6	0	0	5	0	6	A sala do tesouro		
		1 (N)	2 (S)	3 (E)	4 (O)				

LEE (3,2) = 2

Fig. 7. A estrutura de dados, adequadamente organizada, reflete a máquina de estados variáveis do labirinto, tornando além disso mais limpo e geral o programa.

E e O) assinale, cada vez, os traços (passagens) TI percursos de ida e os TV percursos de volta do compartimento do Tesouro (T), partindo do compartimento de entrada (E).

Damos por certo que vocês o tentarão. Tudo fica mais simples uma vez que foi definida a Máquina de estados variáveis e a estrutura de dados. Não autorizamos aos mais negligentes a que olhem a solução que estamos a ponto de dar, sem tentá-lo antes ao menos uma vez.

```
1 REM DADOS E ATRIBUICOES INICIAIS
10 DATA 0,2,0,0,3,0,1,4,5,2,4,0
20 DATA 0,2,5,3,0,3,4,6,0,0,5,0
30 DATA "NA ENTRADA","NO SALAO","NO QUARTO"
40 DATA "NA COZINHA", "NA ANTESALA","NA SALA DO TESOURO"
50 DATA "N","S","E","O"
100 DIM UNI(6,4),ST$(6),DIR$(4)
110 FOR H=1 TO 6
120 FOR K=1 TO 4:READ UNI(H,K):NEXT K
130 NEXT H
140 FOR I=1 TO 6:READ ST$(I):NEXT I
150 FOR I=1 TO 4:READ DIR$(I):NEXT I
160 REM
200 REM INICIALIZACOES DIVERSAS
210 SW=0:H=1:TI=0:TV=0:HF=6
220 CLS:PRINT "ESTAS ";ST$(H)
225 REM
230 REM EXAME DE CONDICOES-LIMITE
240 IF SW=1 AND H=1 THEN GOSUB 3000:GOTO 260:REM FINAL
250 GOTO 290:REM CASO NORMAL
260 INPUT "QUERES TENTAR OUTRA EXPLORACAO";R$
270 IF LEFT$(R$,1)="S" OR LEFT$(R$,1)="s" THEN 200
280 END
290 IF H=HF THEN GOSUB 2000:REM TESOURO ALCANCADO!
295 REM ESCOLHA DO USUARIO
300 REM
310 INPUT "PARA ONDE QUERES IR? (N,S,E,O) ";R$
320 FOR K=1 TO 4
330 IF LEFT$ (R$,1)=DIR$(K) THEN 360:REM ENCONTRADA COLUNA
340 NEXT K
350 GOTO 300:REM SINTAXE ERRADA, REPITA INPUT
360 IF SW=0 THEN TI=TI+1
370 IF SW=1 THEN TV=TV+1
380 IF UNI(H,K)<>0 THEN 410
390 PRINT "NAO SE PODE IR PARA O ";R$
400 FOR R=1 TO 1000:NEXT R:GOTO 220:REM PAUSA E AO PRINCIPIO
```

```
410 H=UNI(H,K):GOTO 220:REM RETOMA O CICLO
420 REM INICIO SUBROTINAS
1990 REM TESOURO CONSEGUIDO
2000 SW=1:PRINT:PRINT "O TESOURO E TEU!!"
2010 REM OUTRAS INSTRUCOES AD LIBITUM
2980 RETURN
2990 REM FINAL
3000 CLS:PRINT "FIM DO JOGO"
3010 PRINT:PRINT "PERCORREU";TI;"PASSAGENS";
3020 PRINT " PARA A IDA ";TV;" PARA A VOLTA"
3030 REM OUTRAS INSTRUCOES AD LIBITUM
4000 RETURN
```

Mais curto que o esperado. Verdade? O mérito de que seja tão compacto vem dado seguramente pela extrema simplicidade da micro-aventura, Inclusive tem que admitir que ao programa faltam algumas "cartas" para constituir verdadeiramente o esqueleto de um jogo de "aventuras" mais desenvolvido: seria necessário acrescentar coisas como uma interface mais cuidado com o usuário e além disso, fazê-lo mais interessante em determinados aspectos (por exemplo, situações de perigo, enigmas e similares) nem todos facilmente reconvertíveis a estruturas de dados do tipo das que aparecem na figura anterior (ainda que...quem sabe?). Por outro lado, este problema se refere, realmente aos limites da engenharia do Software, sobretudo sua eterna busca da chamada "reutilidade" dos programas. A prática demonstra, além disso, que quase 70% dos casos nos quais tem que fazer frente a um problema novo, ainda que seja parecido aos precedentes, é necessário voltar a escrever novamente grande parte do programa.

Ainda assim, a solução proposta permite acrescentar moradias e fazer variações, à vontade, das uniões recíprocas: é suficiente modificar adequadamente os DATA e as linhas iniciais, nas quais são carregados os parâmetros nas diversas matrizes. E, sobretudo, foram proporcionadas algumas idéias que poderão ser reutilizadas, op rtunamente trocadas, em contextos parecidos, ou inclusive, à primeira vista distintos. Em suma, o saber não ocupa lugar, e isto também se aplica à moderna Arte da Programação.

Quanto à explicação do programa nos limitaremos ao essencial para não ofender a maioria de vocês, que supomos conhecem o BASIC (e senão, basta que procurem os volumes 5, 6 e 7 da B.B.I). Será suficiente seguir passo a passo as instruções para compreeder o que vai sucedendo. A linha "chave" é a 410:

410 H = UNI(H,K): GOTO 220

O índice H, corresponde à linha da matriz UNI, constitui, na prática, o estado de nossa máquina de estados variáveis; com a instrução já vista, é extraído de UNI o novo valor (a nova moradia), a qual nos leva à direção (coluna) escolhida pelo usuário. Quanto ao índice K desta coluna, determinando segundo a resposta R $, dada na linha 310 em termos alfanuméricos (N, S, E e O, ou então Norte, SUL, Este, Oeste, pois somente conta a primeira letra extraída com LEFT $ (R$,1) na linha 330), se estabelece numericamente desde as linhas 320 até a 340. A procura falha (não contentamos com um ponto cardeal) se o ciclo FOR...NEXT é completado, enquanto que se tem êxito é interrompida antecipadamente, o que significa que o primeiro caracter de R$ é um dos que estão contidos no pequeno vetor DIR$(N, S, E, e O).

Terminamos com estas pequenas anotações: CLS, comando de limpeza da tela, pode ser substituído pelo equivalente em outros BASIC (por exemplo HOME, no Aplle). O loop de atraso, realizado com um loop "ocioso" na linha 400, serve para manter o tempo necessário à frase "NÃO SE PODE IR ATÉ ELE"; R$. A variável HF, inicialmente igual a 6, determina, sobre a condição H = HF, a chegada ao tesouro; no caso de uma ampliação do jogo, admite a possibilidade de variar seu valor, para ajustá-lo ao novo jogo, unicamente ao princípio (linha 210). As REM das linhas 2010 e 3030 servem para recordar que as respctivas subrotinas podem ser trocadas à vontade (por exemplo, para acrescentar gráficos e/ou sons). Ao jogar com o programa alguns de vocês notarão que, se voltar a entrar pela segunda vez no compartimento do tesouro, é repetida a mensagem "O TESOURO É TEU", que pode tornar-se trágico para quem talvez, esteja morrendo de fome tentando encontrar a saída. Quem não gostar disso, poderá remediá-lo sem muita dificuldade.

Por último a variável booleana SW, que é ativada quando alcançado o tesouro (linha 2000), permite determinar comodamente na linha 240 o final do jogo: este termina quando encontramos no compartimente 1 com SW = 1, condições que testemunham o final do pesadelo.

CAPÍTULO III

OS JOGOS COM ESTRATÉGIAS VENCEDORAS

A selva informática dos jogos

A informática revela um insuspeito parentesco com a botânica, ainda que somente seja porque o objeto que encontramos com maior freqüência é a árvore, muitas vezes colocada ao contrário com a copa para baixo e as raízes para cima. Já falamos fugazmente dos dados.

Agora, depois de ter sugerido que estruturas estão à ordem do dia nos sistemas operacionais (por exemplo no célebre sistema operacional Unix, a estrutura dos arquivos é arborescente) passaremos a tratar outra importante utilização das árvores: aquela em jogo...na Teoria dos Jogos e nada mais e nada menos na nova e fascinante disciplina da Inteligência Artificial. Entre os jogos mais simples que tomam em consideração esta teoria, estão aqueles nos quais dois jogadores se alternam nos movimentos, seguindo as regras que definem sua própria licitude e as situacões de vitória, derrota e empate.

Agora então, pode ser demonstrado facilmente que em muitos destes jogos, aos quais podemos chamar mecânicos, também podemos contradizer ao sentido comun, pois um dos dois jogadores está predestinado a ganhar ou, pelo menos, a obter uma igualdade (naqueles que admitam um empate). Efetivamente, somente se ficar equivocado, ignorando a "estratégia vencedora" que está ao seu alcance, permite ao contrário dispor por sua vez de outra estratégia vencedora. As damas, o xadrez a muitos outros perten-

cem a esta categoria, e somente a astronômica quantidade de combinações em jogos nos dois primeiros impedem conhecer para isto uma só estratégia matematicamente vencedora; nem sequer se sabe se tal estratégia é previlégio do que inicia ou do que joga em segundo lugar.

Um jogo com estratégia vencedora: o Nim ou Marienbad

Para entender melhor este capítulo, colocaremos um exemplo clássico que será objeto de um problema bastante instrutivo e inclusive, acreditamos, original.

O jogo de Nim é um jogo matemático suficientemente simples como para que se possa delinear uma estratégia vencedora. Existe quem o chama "Marienbad"(na realidade Nim e Marienbad são o mesmo jogo, mas com regras opostas). Para simplificar não faremos nenhuma distinção entre os dois. O jogo em questão era um motivo repetido freqüentemente no filme do diretor francês Alain Resnais "L'annèe derniére á Marienbad" (O ano passado em Marienbad), um filme misterioso da Nouvelle Vague. Nela, protagonista, G. Albertazzi, perdia sistematicamente neste jogo de palitos - nos ricos salões da estação termal - com o marido de sua amante. Desafortunado no jogo, afortunado em amores? Que seja! A questão era simplesmente, que o enigmático e pouco fascinante adversário, conhecia a chave da estratégia vencedora do Nim, a mesma que nos permitirá programar o computador de maneira que a máquina vença inexoravelmente (e sem o consolo de contrapartidas eróticas).

O Marienbad consiste em várias filas de objetos (palitos, velas, botões, etc.) Na Figura 1 vemos, à direita, a configuração que é adotada habitualmente, enquanto que à esquerda está a que utilizamos em nosso programa. Como veremos depois, existem infinitas combinações possíveis e o mesmo de jogos, ganhos alguns por quem começa e outros por quem move em segundo lugar.

As regras são muito simples: os jogadores são alternados nas jogadas e, em cada uma, podem pegar desde um até todos os palitos da mesma fila. Perde quem tem que pegar o último.

Examinemos um Marienbad muito simples (Figura 2-a) e vamos chamá-lo 3-2-1, pelo número de peças de suas filas. A árvore do jogo está desentranhada na mesma figura, em b. Está claro que se trata de uma árvore ramificada: a raiz constitui a configuração inicial e as ramas as sucessivas jogadas, cada uma das quais conduz à nova configuração. A árvore é um grafo particular da má-

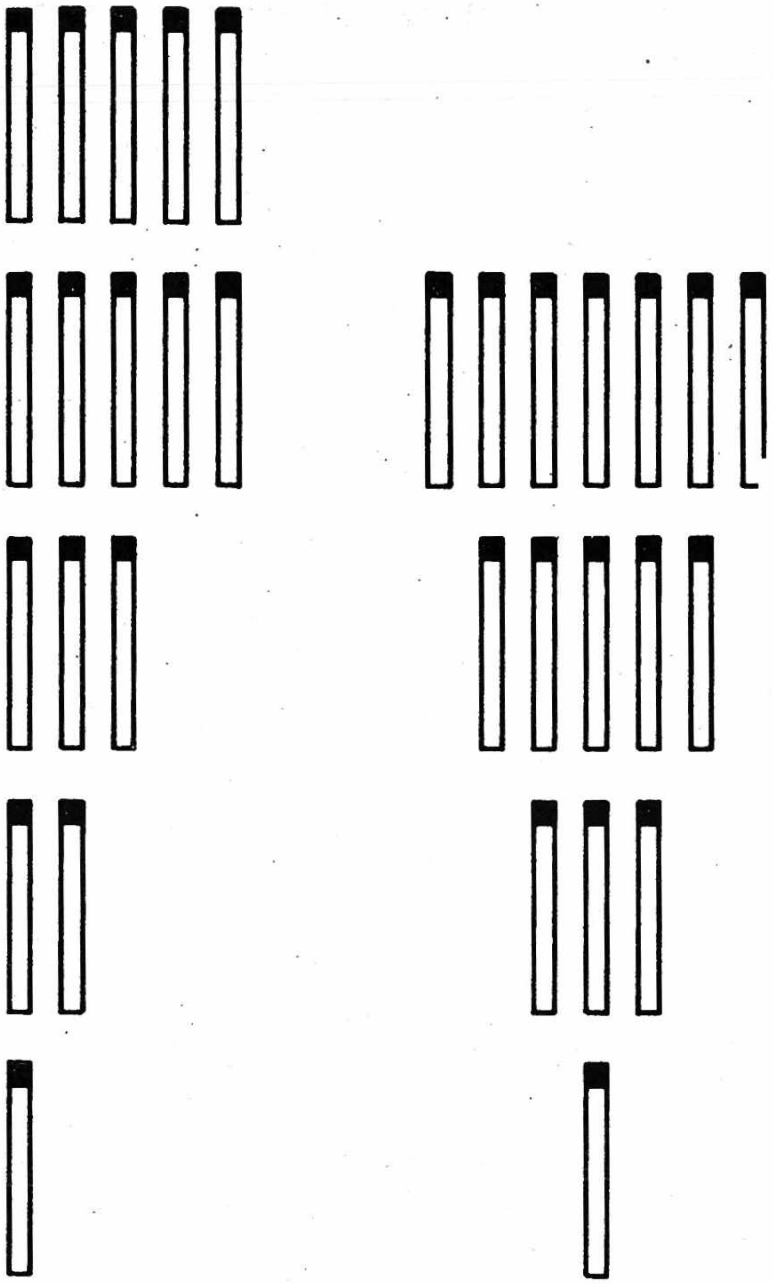


Fig. 1. Duas configurações do Nim ou Marienbad. À esquerda a usada em nosso programa, à direita a que é utilizada normalmente para jogar. As combinações possíveis são infinitas.

quina de estados variáveis do jogo. Na realidade, por motivos de espaço e de simplificação, não representamos grande parte das jogadas evidentemente perdedoras para o segundo jogador. Com as notas n/m, escritas ao lado dos diferentes arcos, foi indicada a escolha de tomar n objetos da fila m, enquanto que dentro de cada nó (estado) estão representados verticalmente os números resi-

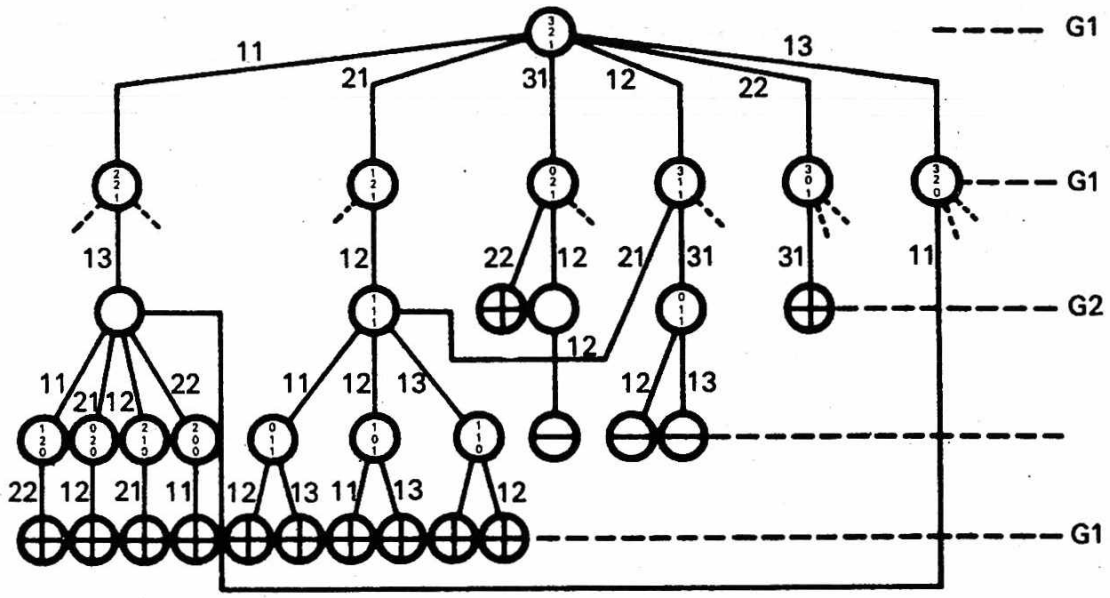


Fig. 2. Árvore simplificada do Marienbad 3-2-1 (a). Para maior simplicidade foram omitidas muitas escolhas, especialmente as perdedoras para o segundo jogador J2. Ganha "matematicamente" este último.

duais das três filas. Por razões tipográficas, no texto que segue escrerevemos, ao contrário: 2-2-1, 3-0-1, etc. Finalmente, note que os nós estão reagrupados em níveis, J1 e J2 alternadamente, que denotam aqueles nos quais a jogada corresponde ao primeiro ou ao segundo jogador.

Seguir o desenvolvimento do micro-Nim da figura não é difícil, e menos ainda ajudando-se, por exemplo, com palitos reais. Analisando assim todas as combinações possíveis, nos daremos conta em seguida que o segundo jogador ganha "por nariz". Por exemplo, se depois da jogada 2/1 do primeiro, o segundo se decide por 1/2 deixando na mesa 1-1-1, qualquer jogada que faça o outro deixará dois palitos em filas diferentes: uma destas pegará o segundo, obrigando ao adversário a última, perdendo a partida.

Para abreviar, observamos que no grafo (ou árvore) da figura mencionada são indicados:

- com traço grosso as jogadas vencedoras do segundo jogador;
- os nós finais, caracterizados todos por um estado 1-0-0, 0-1-0 ou 0-0-1, são pequenos círculos que encerram um sinal " + " ou "-", conforme vença o segundo ou o primeiro jogador (pode ocorrer isto se o outro se equivoca, por exemplo, com um 1/1 na

configuração 3-2-1, com o que primeiro, se não é tonto, replica com 1/1 e ganha).

- partindo desde baixo, foram colocados oportunamente sinais " + " ao lado dos nós que são revelados nós-raiz de subárvores vencedoras para o segundo jogador: subindo gradualmente é fácil constatar que todas as subárvores que se referem à raiz da árvore inteiro têm a marca " + ", de onde todo o Marienbad 3-2-1 é vencedor para o segundo jogador. Note também que os nós 2-2-0 e 1-1-1 foram alcançados ambos através de dois caminhos diferentes.

Nosso algorítmo para o nim

Tudo o que vimos no exemplo precedente pode ser generalizado. Nos jogos do tipo que estamos examinando, a vitória é exclusiva de um "predestinado" que, cada vez que vai jogar, só tem

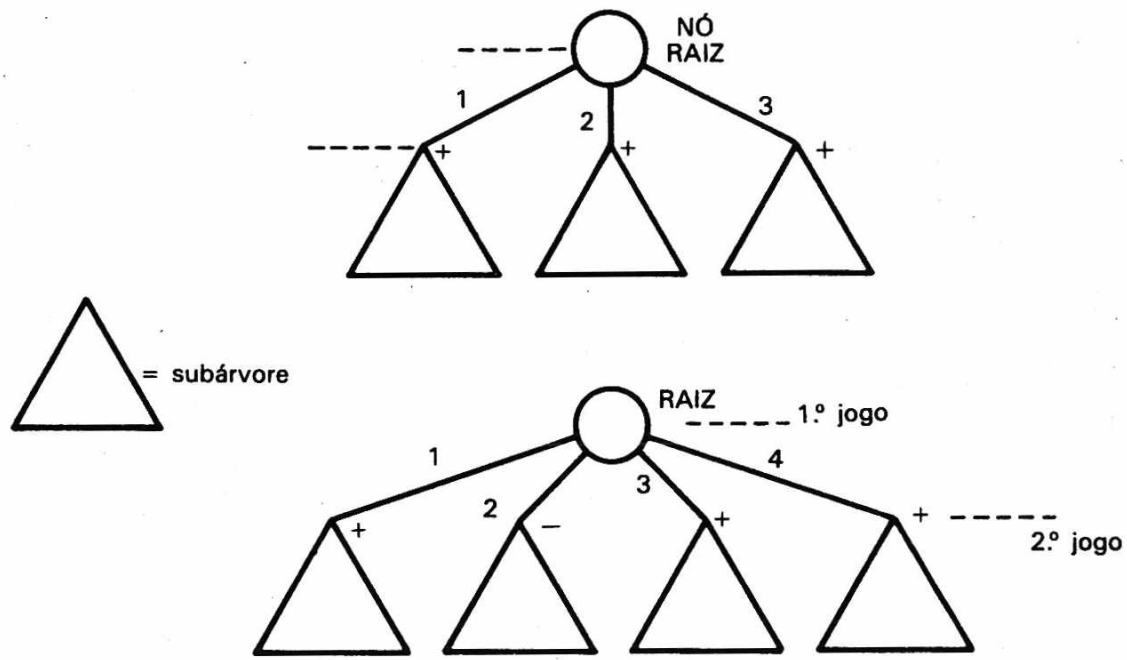


Fig. 3. A árvore de um jogo pode ser subdividida em subárvores (e subjogos), cada um dos quais terá uma combinação vencedora para um só jogador entre J1 e J2. Se todos os subjogos que saem de um único nó-raiz são combinações vencedoras para J2, como sucede em (a), J1 não tem possibilidade de ganhar (exceto se existe algum erro por parte de J2). Se, ao contrário, existe também uma subárvore com combinação vencedora para o primeiro jogador (b), este assegura a vitória tomando esta escolha (2).

que evitar as jogadas que mudam a situação, oferecendo a vitória matemática ao adversário. Em algum ou em todos os nós pode acontecer que a "jogada feliz" seja uma somente, mas não importa, é suficiente.

A demonstração do teorema já vimos empiricamente no parágrafo anterior: é baseado na decomposição de árvore do jogo em subárvores (iniciando por baixo), cada um dos quais é ganhador para um dos jogadores e perdedor para o outro. A árvore completa será composta - partindo de baixo - de subárvores, que saem da raiz, marcadas com " + " ou "-", conforme seja o vencedor dos respectivos subjogos o primeiro ou o segundo jogador. Agora então, se chegar a uma situação como a da Figura 3-a, está claro que, seja qual for a jogada que faça o primeiro, estará predestinado à derrota, salvo generosidade ou ignorância do adversário. Se ao contrário, como em b, tem pelo menos uma subárvore marcada com "-" é com esta jogada com a qual o primeiro se assegura a vitória.

Voltamos a Marienbad (hoje em dia esta cidade se chama Mariansk e Lazne...). É o seu um jogo um pouco mais sério e amplo que o mísero 3-2-1 visto até agora. Fica fácil imaginar a frondosa e intrincada que chegará a ser, sua árvore correspondente. Geralmente, os jogos de computador são baseados na exploração, mais ou menos exaustiva, da subárvore das jogadas seguintes, mas fazer isto lateralmente é muito complicado e lento, exceto para os jogos simples, que tem certas limitações: podadas, limitações da profundidade, ou seja, dos níveis das jogadas e outros critérios. De todas as maneiras, do mecanismo de exploração de árvores trataremos no próximo capítulo. Aqui vamos evitá-lo, adotando outro método, mais que nada para iludir o choque repentino com uma série de delicados conceitos informáticos, como as pilhas e o backtracking.

Utilizaremos uma idéia que, pelos menos neste caso específico, simplifica o procedimento e talvez aumenta a velocidade. Naceu da observação psicológica de como atua o ser humano quando joga Nim ou outros jogos parecidos. Nossa mente não é capaz de baixar demasiados níveis porque, mais ou menos conscientemente, o que fazemos é analisar a configuração (em inglês se diria "pattern") conseqüente a cada uma das jogadas possíveis nesse nível: se é reconhecida como uma das vencedoras (ou como "vantajosa" nos jogos muito complicados) a jogada é efetuada, senão é examinada outra (escolhida normalmente ao acaso). Agora então, se tiver o repertório de todas as configurações vencedoras, bastaria a análise de todas as vezes somente até o primeiro nível.

Explicaremos com um exemplo. Vamos supor que na mesa

temos 4-2-1. Pois bem, quem conhece que 3-2-1 é vencedora (para ele) não duvidará em jogar 1/1. E se tivesse sido 2-1-5? Não é preciso muito para deduzir, por analogia, que 1/3 produz 2-1-3 e que este modelo é totalmente "equivalente" a 3-2-1, assim como a 1-2-3 ou 1-3-2. Comprovamos assim que não vale a pena resolver todas as configurações: será suficiente reconduzir as equivalentes a uma única comum. O critério é simples e baseado no fato que:

- os zeros não contam;
- a ordem dos dados tampouco conta.

Portanto será conveniente fazer referência a uma classificação pré-estabelecida. Nós adotaremos a decrescente.

Chegados a este ponto construiremos manualmente uma pequena base de dados das configurações vencedoras (repetimos pela última vez: "vencedoras" para quem as deixa na mesa).

O procedimento para construir estas configurações é mecânico e como veremos, pode ser confiada a um programa computadorizado. Vai desde o mais simples até o mais complexo (em informática se diria que é do tipo "bottom up"), partindo portanto da configuração mais elementar possível, ou seja, 1. Como dissemos antes este 1 está no lugar de todas as possíveis (1-0-0-...-0, 0-1-0...-0, 0-0-0-...-1). Limitemo-nos, por simplicidade, a cinco linhas e façamos girar as cifras de forma que:

- se respeite a ordem decrescente ao representar o estado;
- cresçam, até onde seja possível, as linhas inferiores e as linhas vazias;
- depois de uma cofiguração m-m-m-...-m se passa a...-m + 1-0-0-...-0 (que, pelo dito, será abreviada com...-m + 1, omitindo os zeros).

Em base nestes critérios, depois de 3-2-1 vem 3-2-2 enquanto que a 4-3-2 seguem 4-3-2-1 e 4-3-2-1-1 . Concretizando teremos:

1; 1-1; 1-1-1; 1-1-1-1; 1-1-1-1-1;
2; 2-1; 2-1-1; 2-1-1-1; 2-1-1-1-1;
2-2; 2-2-1; 2-2- 1 1; 2-2-1-1-1; 2-2-2; 2-2-2-1; 2-2-2-1-1;
2-2-2-2; 2-2-2-2-1; 2-2-2-2-2;
3; 3-1; 3-1-1; 3-1-1-1; 3-1-1-1-1;
3-2; 3-2-1; 3-2-1-1; 3-2-1-1-1; 3-2-2-; 3-2-2-1; 3-2-2-1-1;
3-2-2-2; 3-2-2-2-1; 3-2-2-2-2;
3-3; 3-3-1; 3-3-1-1; 3-3-1-1-1; 3-3-2; ...etc.

A procura das configurações adequadas é parecida em tudo à dos números primos sucessivos: cada nova configuração se faz a prova da "invencibilidade", comparando-a com as configurações vencedoras encontradas até esse momento. Se, partindo de uma das jogadas permitidas, é possível reconduzí-la a uma das vencedoras, evidentemente tratar-se-à de uma configuração vencedora, mas para o adversário.

Portanto a descarta. Ao contrário, será incluída a que sai intata do teste, para todas as jogadas possíveis. Seguindo este critério encontraremos em seguida as primeiras configurações vencedoras:

1; 1-1-1; 1-1-1-1-1;
2-2; 2-2-1-1; 2-2-2-2;
3-3; 3-2-1
...

Seguindo com este aborrecido exercício são examinadas as configurações que seguem, ao lado das quais será colocado, entre parêntesis, "VENCEDORA" ou, em caso contrário, as configurações vencedoras às quais pode ser reduzida:

3-2-1-1 (3-2-1ou2-2-1-1)
3-2-1-1-1 (VENCEDORA)
3-2-2 (3-2-1ou2-2)
3-2-2-1- (3-2-1ou2-2-1-1)
3-2-2-1-1 (2-2-1-1ou3-2-1-1-1)
3-2-2-2 (2-2-2-2)
3-2-2-2-1 (VENCEDORA)
3-2-2-2-2 (2-2-2-2ou3-2-2-2-1)
3-3 (VENCEDORA)
.....

Nos DATA das linhas 4 a 12 do programa da Figura 4 são representadas posteriores configurações Nim vencedoras. Quem desejar pode tentar encontrar outras. Quanto mais combinações vencedoras memorizarmos mais possibilidades teremos de ganhar facilmente dos ignorantes e desprevinidos.

```
2 REM O ANO PASSADO EM MARIENBAND (ALIAS "NIM")
4 DATA " 1"," 1 1 1"," 1 1 1 1 1"," 2 2"," 2 2 1 1"," 2 2 2 2"
6 DATA " 3 2 1"," 3 2 1 1 1"," 3 2 2 2 1"," 3 3"," 3 3 1 1"," 3 3 2 2"
8 DATA " 3 3 3 2 1"," 3 3 3 3"," 4 4","4 4 1 1"," 4 4 2 2"," 4 4 3 2 1"," 4 4 3 3"
```

```
10 DATA " 4 4 4 4"," 5 4 1"," 5 4 1 1 1"," 5 4 2 2 1"," 5 4 3 2"," 5 4 3 2 1"
12 DATA " 5 4 4 4 1"," 5 5"," 5 5 1 1"," 5 5 2 2"
14 DATA 5,5,3,2,1 : REM DADOS DE CONFIGURACAO INICIAL
100 LIM=29 : NL=5 : DIM CV$(LIM),S(NL) : VERDADEIRO=-1 : FALSO=0
110 FOR I=1 TO LIM : READ CV$(I) : NEXT I
120 FOR R=1 TO NL : READ S(R) : NEXT R
130 HOME : PRINT "TEM VARIAS FILAS DE PALITOS" : PRINT
140 PRINT "  FILA NUMERO","  PALITOS   " : PRINT
150 FOR I=1 TO NL : PRINT "      ";I,"    ";S(I) : NEXT I
160 PRINT : INPUT "ESCOLHA LINHA:";L : IF L<1 OR L>NL THEN 160
170 INPUT "ESCOLHA PALITOS : ";NC : IF NC<1 OR NC>S(L) THEN 160
180 S(L)=S(L)-NC
190 REM O COMPUTADOR ESTUDA A JOGADA
200 GOSUB 500 : REM COPIA S NA MATRIZ-TRABALHO W
210 FOR I=1 TO NL
220 IF S(I)=0 THEN 360 : REM VE A NEXT SALTANDO A LINHA VAZIA
230 FOR PROVA=1 TO S(I)
240 W(I)=W(I)-PROVA
250 GOSUB 1000 : REM DOUBLE-SORT DE W (NOVO)
260 GOSUB 1200 : REM TRANSF. DE W EM CADEIA C$
270 REM BUSCA DE C$ NA MATRIZ CV$
280 ENCU=FALSO
290 FOR K=1 TO LIM
300 IF C$=CV$(K) THEN ENCU=VERDADEIRO
310 NEXT K
320 IF ENCU THEN NC=PROVA : LINHA=I : PROVA=S(I) : GOTO 340
330 GOSUB 500 : REM REESTABELECE S EM W
340 NEXT PROVA
350 IF ENCU THEN I=NL
360 NEXT I
370 IF NOT ENCU THEN PRINT "ERRO NOS DATAS" : STOP
380 PRINT : PRINT"EU, COMPUTADOR, TIRO";NC;" PALITOS"
390 PRINT "DA LINHA ";LINHA
400 S(LINHA)=S(LINHA)-NC : FOR K=1 TO 1000 : NEXT K
410 IF C$<>" 1" THEN 140
420 PRINT : PRINT "...FICANDO NA MESA"
430 PRINT "UM SO PALITO, GANHEI!"
440 PRINT : INPUT "OUTRA PARTIDA (S/N)";R$
450 IF LEFT$(R$,1)="S" THEN RESTORE : GOTO 110
460 END
```

```
490 REM COPIA DE S EM W
500 FOR K=1 TO NL : W(K)=S(K) : NEXT K : RETURN
1000 FOR K=2 TO NL
1010 IF W(K)>W(K-1) THEN C=W(K) : W(K)=W(K-1) : W(K-1)=C : K=1
1020 NEXT K : RETURN
1200 REM TRANSFORMACAO MATRIZ W EM CADEIA C$
1210 C$="" : FOR K=1 TO NL
1220 IF W(K)=0 THEN K=NL : GOTO 1240 : REM OMITE ZEROS
1230 C$=C$+STR$(W(K))
1240 NEXT K : RETURN
```

Fig. 4. Listagem do programa que sempre ganha o Marienbad 5-5-3-2-1 (realizado no BASIC do Apple).

O programa

Uma vez definido o algorítmo e, como ele, sua respectiva máquina de estados variáveis arbórea (note que desta vez o algorítmo a subentende e, em certo sentido, a sobrepassa) fica por meditar uma estrutura de dados apropriada. Depois disto fazer um programa corretamente será somente questão de paciência, ainda que necessitará bastante, porque inclusive uma pequena falha (esquecer o RETURN em uma subrotina, equivocar as modalidades executivas de uma instrução BASIC, etc.) será suficiente para fazer cambalear ao mais comprovado dos algorítmos. Afinal poderemos assombrar (moderadamente) a nossos amigos colocando-os diante de uma máquina que sempre ganha.

Sem pretender que seja uma solução ótima, sugerimos a matriz e a cadeia como estruturas de dados apropriadas. De fato, nos parece que:

- a matriz (de cinco elementos no nosso caso) é cômoda para realizar a classificação (descendente) e as diferentes "descarnaduras" de elementos;
- a cadeia torna-se uma representação sintética das configurações, especialmente com a finalidade de uma comparação imediata (sobre a ocupação de espaço e a extensão do método a jogos Nim mais amplos haveria muito que discutir).

Não nos resta mais que dissecar, pedaço por pedaço, o programa proposto. Geralmente, reduzimos tudo a seus termos essenciais, como comentários muito concisos e interface-usuário espartano. Quem tem vontade poderá, por exemplo, usar os gráficos (para representar visualmente os palitos) e outras facilidades dificilmente

transportáveis de um micro a outro.

Os DATA iniciais representam as configurações vencedoras. Encontramos mais simplicidade substituindo o traço separador pelo espaço em branco (em inglês blank), o que assegura uma boa transferibilidade entre diferentes dialetos BASIC. De todas as formas tem que ter cuidado e fazer preceder a cadeia por um espaço. Este foi feito em atenção ao BASIC Microsoft e derivados nos quais a função STR$(X), coloca, sistematicamente, um espaço diante do número X, ao convertê-lo em cadeia (ou seja, transforma 1234 em " 1234"). O programa da figura para alternar dialetos de similar popularidade, é referente ao BASIC da Apple. De todas as maneiras, as modificações que devem ser feitas são mínimas. Os microsoftistas somente terão de mudar, na linha 100, VERDADEIRO = 1 por VERDADEIRO = -1 e a linha 1230 assim:

```
1230 C$ = C$ + SRT$(W(K))
```

Voltando à listagem (linha 110), as configurações vencedoras são carregadas em seguida na matriz CV$. Imediatamente são colocadas nos seis compartimentos do vetor de estado (S) os valores 5, 5, 3, 2, 1, que constituem o Nim de calibre médio graças ao qual o computador, jogando como segundo, nos vence incansavelmente.

Desde a linha 130 à 160 a máquina propõe ao adversário humano que mova, depois de mostrar-lhe a situação inicial com o quadro seguinte:

NUMERO DE FILAS	PALITOS
1	5
2	5
3	3
4	2
5	1

Um detalhe: na linha 170 o controle IF NC < 1 OR NC > S(L), cuja missão é impedir que o usuário faça trapaça escolhendo zero palitos ou um número maior que os presentes. Não se fecha com

THEN 170, como poderia ser esperado. O THEN 160, envia à escolha de outra linha e serve para o caso de que S(L) = 0. Se tivéssemos feito THEN 170 estaríamos apanhados em um loop sem fim.

Depois de que o computador tomou nota de nossa jogada (linha 180), tirando os NC palitos da linha L escolhida, é iniciado o estudo da jogada. Para maior comodidade, são citados aqui as linhas correspondentes:

```
190 REM O COMPUTADOR ESTUDA A JOGADA
200 GOSUB 500 : REM COPIA S NA MATRIZ-TRABALHO W
210 FOR I=1 TO NL
220 IF S(I)=0 THEN 360 : REM VE A NEXT SALTANDO A LINHA VAZIA
230 FOR PROVA=1 TO S(I)
240 W(I)=W(I)-PROVA
250 GOSUB 1000 : REM DOUBLE-SORT DE W (NOVO)
260 GOSUB 1200 : REM TRANSF. DE W EM CADEIA C$
270 REM BUSCA DE C$ NA MATRIZ CV$
280 ENCU=FALSO
290 FOR K=1 TO LIM
300 IF C$=CV$(K) THEN ENCU=VERDADEIRO
310 NEXT K
320 IF ENCU THEN NC=PROVA : LINHA=I : PROVA=S(I) : GOTO 340
330 GOSUB 500 : REM REESTABELECE S EM W
340 NEXT PROVA
350 IF ENCU THEN I=NL
360 NEXT I
```

Para que fique mais evidente foi utilizada na listagem um escalonamento, que isola melhor os ciclos mais internos, com índices PROVA e K, "aninhados" dentro do loop principal, de índice I.

O coputador depois de ter copiado S em W (na subrotina 500) prova a tirar, linha por linha, um número de palitos crescente PROVA desde 1 até o número S(I) presente na linha I. Então faz a ordenação de W e a transforma em cadeia C S que vai buscar na base de dados CV$ (linhas 280-310). Antes de passar ao seguinte ciclo de PROVA (cuidado!) é necessário restabelecer (com a subrotina 500, chamada na linha 330) a matriz originária S em W, já que a matriz de trabalho foi manipulada. Quando por fim sai, com NEXT da linha 360, sabemos que o booleano ENCU tem que estar ativado. A prudência nos aconselha acrescentar a seguinte linha:

```
370 IF NOT ENCU THEN PRINT "ERRO NOS DATAS":STOP
```

De fato, um descuido ou um erro de transcrição, sempre é possível em fase de ampliação do jogo ou de escrita dos DATA. Como alternativa, quem desejar pode deixar, de propósito, os DATA incompletos (seria como deixar parcialmente ignorante, o computador) e inserir na linha 370 um adequado GOSUB, que nos mande a uma rotina na qual o computador escolha a jogada com critérios aleatórios (naturalmente, depois será necessário modificar um pouco o programa para prever que o computador admita a vitória do homem).

Agora, o resto do programa e as distintas subrotinas, já deveriam ser o suficientemente auto-explicativas e seu exame é mais útil que qualquer tentativa de explicação.

Unicamente, e com respeito aos ciclos aninhados, damos um conselho. É uma boa regra evitar, exceto nos casos mais simples, sair fora de um loop diretamente:em muitos BASIC toparíamos como uma interrupção e a sinalização de erro "NEXT WITHOUT FOR" no primeiro encontro com um NEXT de outro loop externo. Será mais seguro forçar o valor final antes de fazer um GOTO na linha que contenha NEXT daquele ciclo. Consulte para isto as linhas 300 e 320.

O gerador de nim-pattern

Redigir um programa que nos ofereça as combinações vencedoras do Marienbad seria um excelente exercício mental. A solução que propomos aparece na Figura 5. Dado que todos os mecanismos já tenham sido suficientemente explicados, os mais adiantados deveriam tentar soluções de própria colheita. Os mais rápidos e voluntariosos poderão, inclusive, fundir os dois programas com algum menú inicial, tendo em conta que ambos são baseados no teste de "invencibilidade".

```
40 REM ****************************************
50 REM * GERADOR DE CONFIGURACOES GANHADORAS *
60 REM *          DO JOGO DE "NIM"            *
70 REM *     (TAMBEM PROGRAMA MARIEBAD)       *
80 REM ****************************************
90 NC=4 : NL=5 : DIM CV$(200),S(NL),W(NL) : VERDADEIRO=-1
100 CNC=1 : S(0)=NC : S(1)=1 : C$=" 1" : REM PRIMEIRA CADEIA GANHADORA
105 HOME : VTAB(10) : HTAB(7) : PRINT "AGUARDE, ESTOU PENSANDO!!!"
110 GOSUB 400 : REM CARREGADO EM CV$
120 REM CRIACAO NOVA CONFIGURACAO
```

```
130 M=2 :REM MARCHA ATE ADIANTE
140 FOR X=M TO NL
150 S(X)=S(X)+1
160 GOSUB 500 : REM PROVA DE INVENCIBILIDADE
170 IF NOT ENCU THEN GOSUB 300 : GOSUB 400 : REM CARREGA CONF. SE NAO REDUZIVEL
180 NEXT X
190 IF S(NL)=NC THEN 240 : REM PRINT FINAL
200 REM MARCHA ATRAS
210 FOR X=NL TO 1 STEP -1
220 IF S(X)=S(X-1) THEN S(X)=0 : NEXT X
230 M=X : GOTO 140
240 REM PRINT FINAL DAS CV$
245 HOME : HTAB(15) : PRINT "SOLUCOES:"
250 FOR K=1 TO CNC :PRINT CV$(K);"  "; : NEXT K
260 END
300 REM TRANSF. MATRIZ-TRAB. EM CADEIA
310 C$=""
320 FOR K=1 TO NL
330 IF W(K)=0 THEN K=NL : GOTO 350
340 C$=C$+" "+STR$(W(K))
350 NEXT K : RETURN
400 REM CARREGA NA MATRIZ CV$
410 CV$(CNC)=C$ : CNC=CNC+1
420 RETURN
500 REM PROVA DE INVENCIBILIDADE
510 GOSUB 700 : REM COPIA S NA MATRIZ-TRAB. W
520 FOR I=1 TO NL : IF S(I)=0 THEN I=NL : GOTO 640
530 FOR PROVA=1 TO S(I)
540 W(I)=W(I)-PROVA
550 GOSUB 720 : GOSUB 300 : REM SORT+TRANSF. NA CADEIA C$
560 ENCU=0 : REM PROCURA DE C$ NA MATRIZ CV$
570 FOR K=1 TO CNC-1
580 IF C$=CV$(K) THEN ENCU=VERDADEIRO : K=CNC-1
590 NEXT K
600 IF ENCU THEN PROVA=S(I) : GOTO 620
610 GOSUB 700 : REM RESTABELECE S EM W
620 NEXT PROVA
630 IF ENCU THEN I=NL
640 NEXT I
650 RETURN
```

```
700 REM COPIA S NA MATRIZ-TRAB. W
710 FOR K=1 TO NL : W(K)=S(K) : NEXT K : RETURN
720 REM BOUBBLE-SORT (ORDEM DECRESC.) DE W
750 FOR K=1 TO NL-1
770 IF W(K)<W(K+1) THEN C=W(K) : W(K)=W(K+1) : W(K+1)=C : K=0
780 NEXT K : RETURN
790 REM FIM DO PROGRAMA
```

Fig. 5. Programa que origina as combinações. As subrotinas comuns com a anterior têm aqui números de linha diferentes.

Com respeito ao programa anterior, o que vamos ver de imediato tem como novidade o mecanismo de geração de configurações (vencedoras ou não). A solução proposta figura nas linhas 120 e 130 (NC e NL os números de coluna e linha escolhidas). Tem que levar em conta que os números da subrotina não são correspondentes com as do programa precedente:

```
120 REM CRIACAO NOVA CONFIGURACAO
130 M=2 :REM MARCHA ATE ADIANTE
140 FOR X=M TO NL
150 S(X)=S(X)+1
160 GOSUB 500 : REM PROVA DE INVENCIBILIDADE
170 IF NOT ENCU THEN GOSUB 300 : GOSUB 400 :
    REM CARREGA CONF. SE NAO REDUZIVEL
180 NEXT X
190 IF S(NL)=NC THEN 240 : REM PRINT FINAL
200 REM MARCHA ATRAS
210 FOR X=NL TO 1 STEP -1
220 IF S(X)=S(X-1) THEN S(X)=0 : NEXT X
230 M=X : GOTO 140
```

O primeiro truque consiste em colocar NC (ou seja, o número máximo de palitos por linha) no elemento de índice O,S(0)(ver na Figura 6). Ao princípio, uma vez carregada a cadeia C$ = "1" em CV$(1) e estabelecido S(1) = 1 - enquanto que de S(2) a S(NL) terão todos zeros - é executada a "marcha adiante", como chamamos à primeira fase. Esta consiste em incrementar em 1 todos os elementos desde M = 2 em diante (linha 140). Assim , ao primeiro passo

teremos: "1"; "111", 111; "11111". Em seguida, M tomará outros valores, assim que de uma configuração inicial como "32111" se passará (com M inicial igual a 3) a: "32211"; "32221"; "32222". O que ocorre na outra fase, chamada "marcha atrás" (linhas 200-230)? É muito simples: partindo desde baixo (X = NL) até em cima, são colocados a zero todos os S(X) que coincidam, parando-se no primeiro que resulte ser de valor maior. Deste modo, por exemplo, a "43222" seguirá "43000" (como cadeia teremos na realidade, "43"), enquanto que a "32222" seguirá "3". E isto, graças ao truque de ter estabelecido S(Ø) = NC. Já não resta mais que fazer (linha 230) M = X e retomar a marcha adiante, até que S(NL) = NC; não se repete a marcha atrás, mas se chega a impressão final. Se quiser pode comprovar que o processo recorre todas as cofigurações (vencedoras e perdedoras). Basta para isso fazer um par de mudanças:

- apagar a linha 160;
- modificar a linha 170 assim:

170 GOSUB 700: GOSUB 300: GOSUB: 400

(depois de ter copiado cada S em W o converte em cadeia e o carrega em CVS). O programa, após o RUN, fará uma lista de todas as

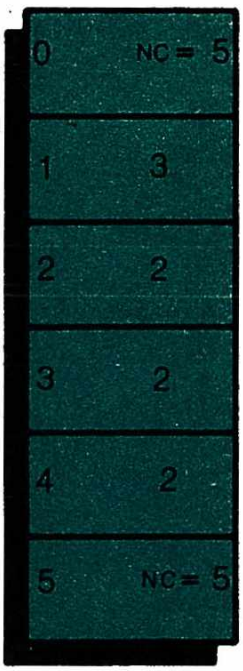


Fig. 6. Matriz de estado S. Na geração das sucessivas configurações se coloca S(0) = 5. Assim, na fase para trás de colocação em zero o teste IF S(X) = S/X-1) proporciona para X = 1, S(1) S(Q) e o 3 da figura não é convertido em um zero (ver texto).

configurações, empregando muito menos tempo que o programa completo. Este último, com todas as suas buscas, provas e ordenações por bolhas, se mostra bastante capengante apenas sobrepassa os valores NC = NL = 5. Com NL = 5 e NC = 6 o Aplle II leva doze minutos para gerar as 59 CVS vencedoras.

Poderiam ser obtidos tempos sensivelmente menores aplicando, para a busca nas tabelas, técnica da bisecção, que consiste em procurar primeiro em uma das metades da matriz, depois na outra metade da metade e assim sucessivamente. Quem a conhecer, nos dirá que esta técnica requer que CV$ esteja ordenado. Pois bem, o está, apesar das aparências que possa dar a estranha geração dos elementos sucessivos. De fato, se trata de cadeias e não de números, e na classificação destas (são dados de tipo alfabético) uma cadeia como "44" é considerada maior que uma, como "4332211", ainda que seja mais longa (o que confirma o acerto do processamento de geração e da escolha do tipo cadeia, não lhe parece?).

Mas a correção que lhe sugerimos é mais simples: é baseada na consideração de que, cada vez que é gerada uma nova CV$, é provável que poderá ser "absorvida" por uma CV$ gerada recentemente (exemplo: "44221" por "4422"); as primeiras CV$ vencedoras serão evocadas cada vez menos.

A experiência confirma esta intuição (racionada), pois os tempos são vistos praticamente reduzidos à metade. A modificação (não esqueçamos) é esta:

```
570 FOR = CNF - 1 TO 1 STEP -1
580 IF C$ = CV$(K) THEN ENCU = VERDADEIRO: K = 1
590 NEXT K
```

Uma vez mais o software é mostrado como uma matéria em contínua evolução.

CAPÍTULO IV

ESTRATÉGIAS INTELIGENTES

Recordando um antigo problema

Veremos agora um problema de estratégia computadorizado simples, como poderá verificar, mas delicadíssimo. O computador deverá encontrar a melhor solução para transportar um lobo, a uma cabra e uma couve-flor, de uma borda do rio até a outra, dispondo de uma barca com capacidade para um só passageiro por viagem (à parte do barqueiro, claro) e evitando que o lobo coma a cabra e a cabra o couve-flor.

A árvore deste jogo é construída sem muitas dificuldades. Também será inetvitável uma varredura exaustiva. Observando a figura 1 notará que a árvore aparece bastante podada mas, paradoxalmente, tem uma amplitude infinita. Para compreender sua evolução, não devemos nos envergonhar se utilizamos três cartões com os rótulos "1", "cb" e "co" - como figuram no grafo para os três companheiros do campones - para que nos guiem em suas posições e movimentos entre as duas bordas. A árvore da figura será melhor entendida se levar em conta que:

- os nós estão assinalados alternativamente com D e E (por borda da Direita e borda da Esquerda: a borda Direita se supõe que é a de partida);
- as ramificações de escolha aparecem assinaladas com os cartões já vistos ("1", "cb", "co") além de "n", significando que não se transporta a ninguém;
- o sinal + indica o êxito do jogo (todos os protagonistas es-

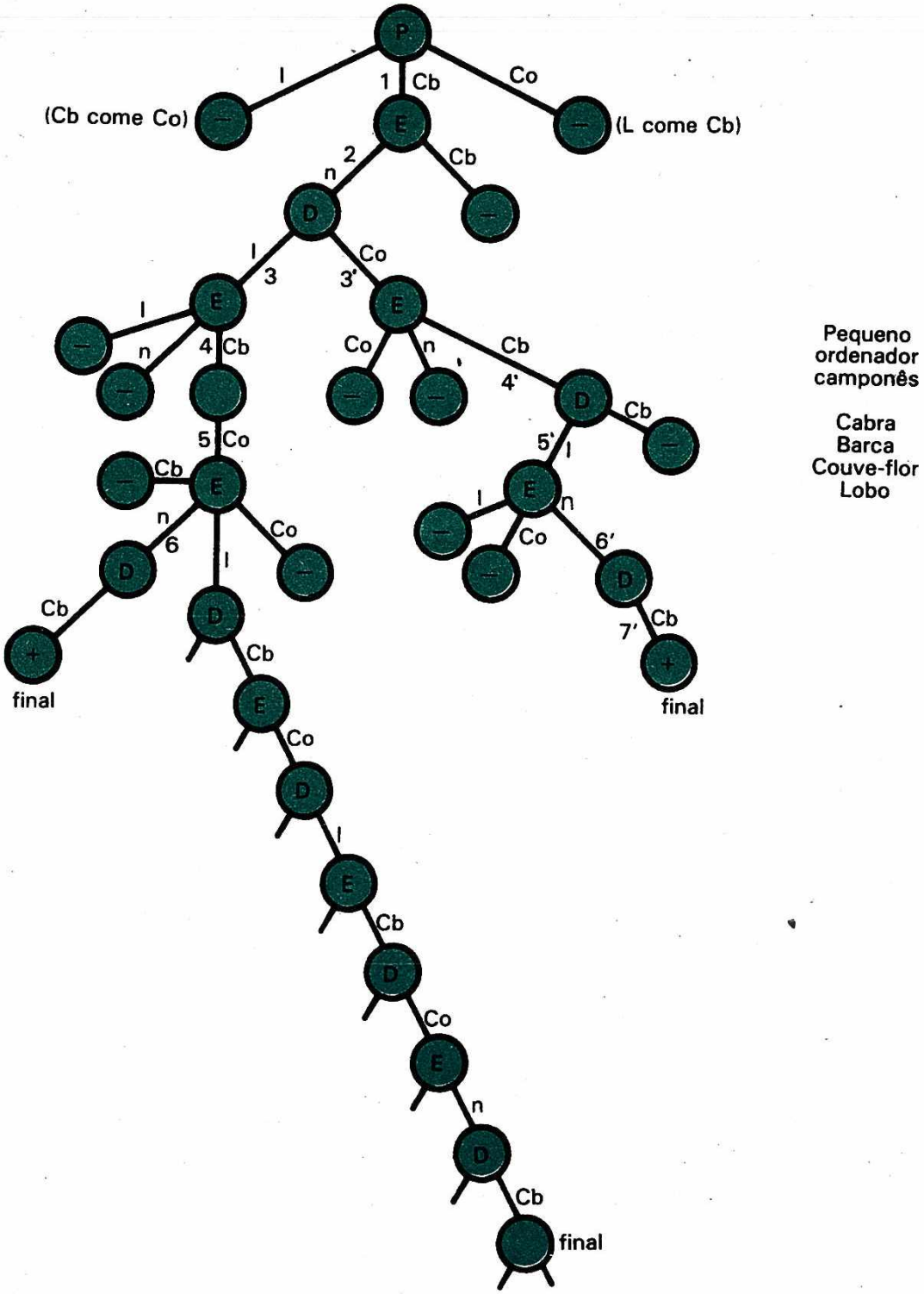


Fig. 1. Árvore simplificada do jogo do lobo, da cabra e do couve-flor. São advertidas duas estratégias abreviadas e infinitas soluções "degeneradas", não otimizadas.

tão na outra borda,
- os nós marcados com "-" supõem jogadas perdedoras.

Com o fim de que a árvore fique podada ao máximo, não somente foram excluídas escolhas que estabeleçam que o lobo coma a cabra ou esta à couve-flor, como também as que na volta transportem o mesmo que foi levado na ida. Na borda esquerda, como será visto, foi feito algo mais específico.

A análise da figura revela que as estratégias vencedoras otimizadas são duas, e são conseguidas com os seguintes movimentos nos dois sentidos:

- cabra (primeira jogada obrigatória) - ninguem - lobo - cabra (volta) - couve-flor - ninguem - cabra;
- cabra - ninguem - couve-flor - cabra - lobo - ninguem - cabra.

Trata-se de dois grupos de sete viagens assinadas na figura com 1,2,3,4,5,6,7, e com 1,2,3',4',5',6',7'.

Precisemos agora a tarefa que se quer confiar ao programa. Em essência esta consiste em: encontrar a jogada sucessiva simulando o desenvolvimento da máquina de estados variáveis e justificar ao operador as jogadas descartadas.

Segundo os critérios tantas vezes discutidos, o mais importante é, antes de tudo, definir uma estrutura de dados adequada. Um primeiro exemplo, para que o leitor pense nele, aparece na figura 2, em linguagem Pascal, junto a dois dos procedimentos possíveis. Em particular serão apreciados os tipos estrurados seguintes:

```
TYPE passag = [ninguem,lobo,cabra, couve-flor]
     borda = SET OF passag;
```

Recordamos, tanto aos pascalistas como aos que não o são, que um tipo SET está formado por todas as combinações possíveis - desde nenhum a todos - dos elementos do tipo-dado que segue a OF. Neste caso seriam: o conjunto vazio "lobo"; "lobo e cabra" e sucessivamente até chegar a "lobo, cabra e couve-flor". Aplicadas as variáveis "bordir" e "boresq", precisarão com clareza a situação nas bordas e facilitarão muito o delineamento dos procedimentos. Por exemplo, o (hipotético) procedimento "avalcomidir" (por "avaliação da comilona na borda direita) incluiria a frase:

```
PROGRAM cabraco:
TYPE pasag = [ninguém, lobo, cabra, couveflor]
     borda = SET OF pasag;
     VAR bodir, boesq, borda;
     pas, comil, victima: pasag;
     nundir, nimesq: INTERGER;
     primvuel, dir: comil: BOOLEAN;
     pilha: matriz [1..200] OF pasag;
     dirpilha: INTEGER;
PROCEDURE inicialic;
     BEGIN
     bodir: = [lobo, cabra, couveflor];
     boesq: = [ ]
     dir: = FALSE;
     pasag: = ninguém;
     primvol: = TRUE;
( * etc * )
     END;
PROCEDURE valdir ( * valor "comilona" na borda dir * )
     BEGIN
     comilona: = TRUE;
     IF numdir    2 THEN comilona: = false;
      ELSE IF lobo IN bodir AND cabra IN bodir
                THEN comil: = lobo; vítima: = cabra;
                ELSE IF cabra IN bodir AND
                          couveflor IN ordech
                     THEN comil: = cabra; vítima: = couveflor;
     END;
(* resto de programa *)
```

Fig. 2. Possível implementação (a título indicativo meramente) da estrutura de dados do jogo anterior em Pascal, com o esboço dos procedimentos que os utilizam, caracterizados pela auto-explicação típica desta linguagem.

```
IF lobo IN bordir AND cabra IN bordir
THEN comil:lobo; vitima: = cabra
```

que possui uma grande eloquência inclusive para quem não sai-

ba que IN faz o teste de domínio do lobo ou a cabra à variável de tipo SET bordir.

Na mesma é usada a estrutura me pilhas (stack). A pilha servirá também para o fim primário, didático, de entender como esta estrutura, chamada às vezes LIFO (Last In First Out: últio a entrar, primeiro em sair) funciona para a varredura de árvores de jogo e distintos labirintos. Precisamente a pilha serve para:

- amontoar as escolhas feitas uma sobre outra e, isto também é importante, para conservar o estado do sitema anterior à jogada atual;
- recuperar um estado precedente (de nível inferior) no caso de que em um nó de uma árvore TODAS as jogadas estejam esgotadas, depois do qual, voltados a um nível anterior, poderemos iniciar "desde o ponto no qual havíamos ficado" a estudar uma jogada distinta a qual nos levou a uma azinhaga sem saída.

Esta é a essência do mecanismo chamado backtraking (volta sobre nossos próprios passos) que, em linguagem como LISP ou Prolog está disponível em forma transparente para o usuário. Para maior clareza, isto está ligado ao conceito de recursividade, sobre o qual não insistiremos, ainda que lhes recomendamos repassar o capítulo correspondente de "Introdução ao Pascal: programação estruturada".

Quanto à pilha (deve seu nome à possibilidade de ser assimilada a uma pilha de pratos) cresce ou decresce através de operações opostas chamadas PUSH e POP respctivamente. Observe a pilha ilustrada na figura 3 e, seguidamente, a figura 7, que ilustra como será usada a pilha no caso de nossa baldeação estratégica.

Estrutura de dados necessária em BASIC

Está ilustrada na figura 4. Ao lado das variáveis particulares de tipo booleano (lógico, on/off, desviadores ou como as queira chamar) e a pilha de cadeias STKS, das quais falaremos a seu devido tempo, devemos destacar:

- a matriz ANS, que tem quatro elementos (incluindo "ningem" e que usaremos nas mensagens que indiquem o nome do passageiro a bordo da barca;
- as matrizes BD e BE que se referem à situação nas bordas

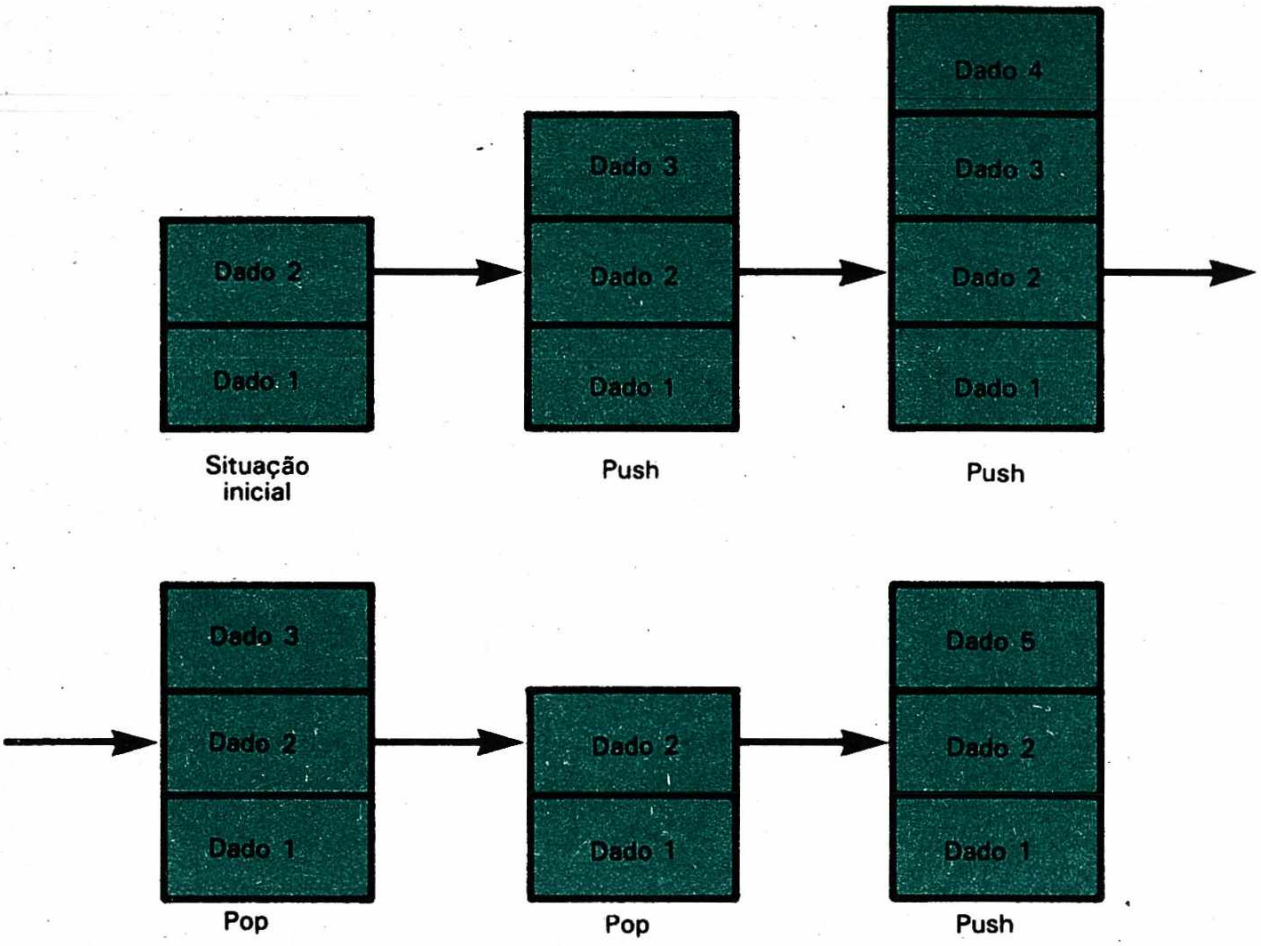


Fig. 3. A pilha (stack) pode ser assimilada com um montão de pratos. A operação PUSH (empurra) coloca um "prato" novo em sua parte de cima (carrega dados), enquanto que a operação contrária, POP, extrai os pratos (dados) em uma ordem inversa ao do empilhamento (o último a ser colocado é o primeiro que é pego). A figura ilustra hipotéticas operações sucessivas de PUSH E POP.

direita e esquerda; o elemento de índice nulo indica o número de personagens ali presentes (excluído o campones), enquanto que os outros elementos podem valer 0 ou 1, denotada a ausência ou presença de seu elemento associado, na seguinte ordem: lobo, cabra, couve-flor. Assim, por exemplo, o estado BD(0) = 2; BD(1) = 1; BD(2) = 0; BD(3) = 1 indica lobo e couve-flor na direita;

- as variáveis PAS, PD e PE, cuja classificação está entre 0 e 3 (0 = "ninguem"; 1 = "lobo", etc.) são respectivamente, o Passageiro atual da barca e os índices usados para escolher ao novo passageiro nas duas bordas;
- DIR é um booleano que indica a Direção atual: de direita à esquerda ou vice-versa.

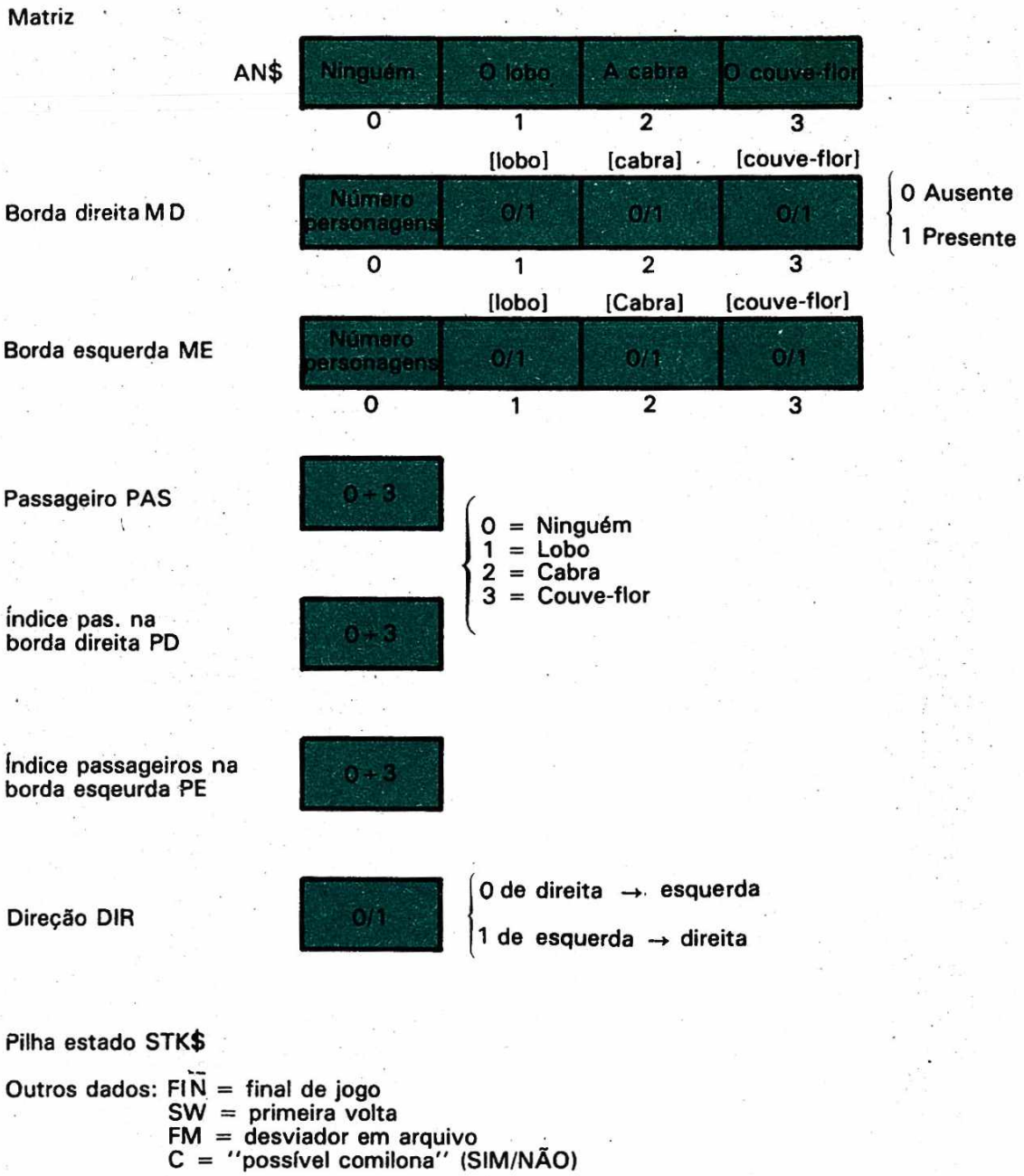
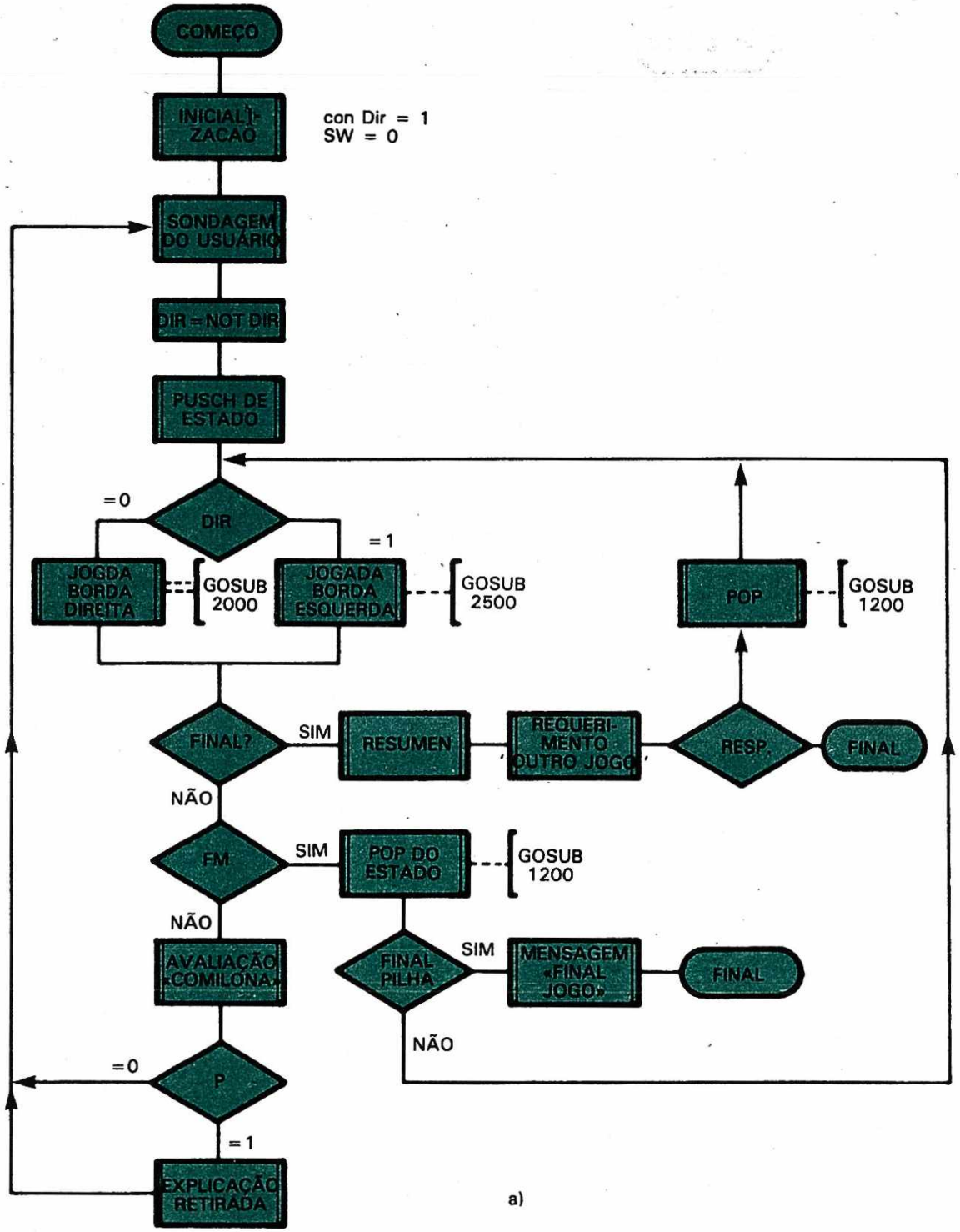


Fig. 4. Estrutura de dados em BASIC. Pode ser criticada uma certa redundância, mas isto favorece a clareza dos delineamentos.

Sobre a possível redundância dos dados pode ser muito discutido, mas nós reivindicamos a virtude da clareza e do "mais vale sobrar..." tanto aqui como em outros aspectos da vida.

Na figura 5 aparece o diagrama de fluxo do conjunto. Se aprecia que a fase de inicialização dos dados segue uma fase de "polling de usuário" (polling equivale, mais ou menos, a investigação)



a)

na qual o computador precisa, a requerimento nosso, o que está fazendo. A instrução elementar DIR = NOT DIR inverte a cada ciclo - 0 a 1 e vice-versa - o indicador de rota. Desta forma, um pouco mais adiante, se tem a possibilidade de escolher ou alternar, nos giros pares e impares, a jogada referente à borda direita ou esquerda. Como veremos em breve, não foi possível unificar em um só proce-

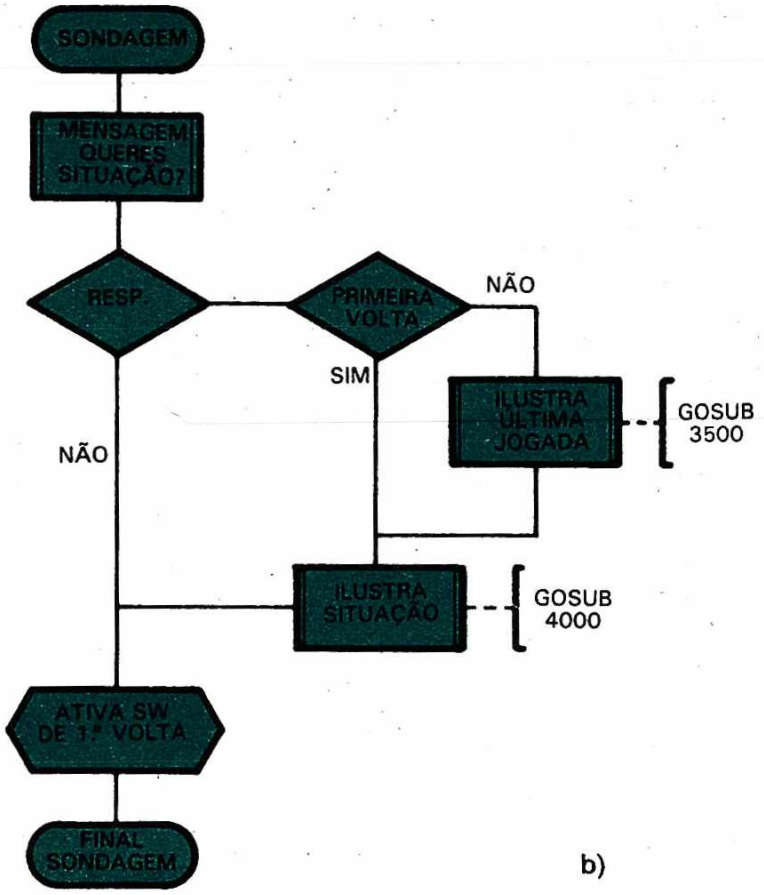


Fig. 5. a) Esquema geral do programa. b) Detalhe do subprograma.

dimento os dois subprogramas, ainda que são bastante parecidos.

Antes de tais avaliações se dispõe da subrotina PUSH, para guardar o estado atual na pilha (ou então, antecipando coisas, as matrizes ANS, BD, BE e as variáveis PAS, PD, PE, etc.). Vamos supor agora que FIN e FM são dois booleanos que representam, o primeiro, o baldeamento completo do trio e o segundo o Final dos Movimentos do nível. No caso de uma resposta afirmativa à pergunta "FINAL?" iremos ao epílogo e conclusão do jogo (o veremos), enquanto que com FM desligado se volta a iniciar com o loop principal. Ao contrário, o que passa quando FM está "on"? Está claro: esgotadas as possibilidades de jogada nesse nível, é executado POP para subir de nível. Para esclarecer este assunto recordem a àrvore da figura 1; imaginem se são encontradas no nó tipo D seguinte a jogada número 6 e que (jipótese absurda) todas as jogadas que foram tentadas deram um resultado negativo: o POP deverá levá-los ao nó E anterior.

Seguindo com o fluxo, depois é examinada a pilha: se estivesse completamente vazia significaria que todas as possíveis ruas tinham sido exploradas e o algorítmo acaba com uma mensagem similar "FINAL JOGO". Em caso contrário nos baldeamos ao ponto X debaixo do PUSH, para entrar na jogada seguinte (o baldeamento do lobo na hipótese que fizemos). Quando, ao contrário, é alcançada a condição FINAL, se faz o RESUMEN seguido do requerimento "OUTRA VOLTA?". Se o usuário contesta que sim, é executado um POP posterior, a via mais direta e lógica para explorar uma solução alternativa à recém encontrada.

A listagem e as explicações

Resumindo, o programa está delineado de forma que o computador, com as regras do jogo em sua memória, seja capaz de varrer de modo exaustivo a árvore inteira, assinalando todas as possibilidades e terminando automaticamente quando tiver percorrido todo o caminho e encontrado todas as soluções. É importante destacar que, a priori, não podemos excluir que NÃO tenha nenhuma solução; em tal caso responderá com "SIM" à pergunta "FINAL DE JOGO?" sem que a condição FINAL e o sucessivo RESUMEN tenham tido lugar. Em suma, o programa resolve "mecanicamente" o problema (deverão "esquecer-se" de que já conhecem - pelo menos - duas soluções).

A listagem do programa, em BASIC, está representada na figura 6 e reflete o diagrama de fluxo do qual já falamos, exceto pela instrução DIR = NOT DIR, que está colocada exatamente como no diagrama. Tem os que precisar que o programa foi escrito para um computador pessoal Apple IIe, motivo pelo qual farão falta pequenas modificações do tipo de CLS, em de HOME e, sobretudo, revisar alguns tratamentos booleanos nos dialetos nos quais verdadeiro" esteja codificado com "1" ao invés de "1" como no BASIC Applesoft.

```
5 REM %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
10 REM %%%                                    %%%
20 REM %%%   SALVAR CABRA E COUVE-FLOR        %%%
30 REM %%%     JOGO DE MICRO - I.A.           %%%
40 REM %%%                                    %%%
50 REM %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Fig. 6. Listagem do programa.

```
60 HOME
70 REM  INICIALIZACOES
80 AN$(0)="NADA"
90 AN$(1)="O LOBO"
100 AN$(2)="A CABRA"
110 AN$(3)="A COUVE-FLOR"
120 REM  PREPARACAO ESTADO BORDA
130 BD(0)=3
140 FOR I=1 TO 3 : BD(I)=1 : NEXT I
150 BE(0)=0
160 FOR I=1 TO 3 : BE(I)=0 : NEXT I
170 DIR=0 : PAS=0 : SW=0 : FIN=0
180 PD=0 : PE=0
190 REM
200 REM  PSEUDO PILHA DE ESTADO
210 DIM STK$(200) : REM  EX ABUNDANTIAN!!
220 IS=0 : REM  INDICE PILHA
230 REM
240 REM  SONDAGEM USUARIO
250 PRINT "QUERES A SITUACAO?"
260 PRINT "(<S>=SIM,<OUTRA>=N)"
270 PRINT : INPUT R$ : IF LEFT$(R$,1)<>"S" THEN 310
290 GOSUB 4000 : REM  ILUSTRA SITUACAO
300 REM
310 SW=1 : GOSUB 1000 : REM  PUSH ESTADO
320 REM  EXECUXAO JOGADA
330 IF DIR=0 THEN GOSUB 2000
340 REM  JOGADA NA BORDA DIREITA
350 IF DIR=1 THEN GOSUB 2500
360 REM  JOGADA NA BORDA ESQUERDA
370 IF FIN THEN 610 : REM  RESUMO E FINAL
375 IF FM THEN 580
380 GOSUB 3010 :  REM  AVALIACAO COMILONA
390 IF NOT P THEN DIR=NOT DIR : GOTO 250
400 REM  C=COMILONA SE NAO -C INVERTE ROTA
410 PRINT "MUMBLE-MUMBLE...ISTO NAO PODE SER FEITO !!!" : PRINT
420 REM  SE C=1 EXPLICA A RETIRADA
```

Fig. 6. Continuação listagem do programa.

```
430 IF DIR THEN 510 : REM  EXPLICACAO PARA BORDA ESQUERDA
440 REM  EXPLICACAO PARA BORDA DIREITA
450 PRINT "SE CARREGO NA BARCA ";AN$(PAS)
460 PRINT AN$(C);" COME ";AN$(V)
470 PRINT "NA BORDA DIREITA..."
480 FM=NOT FM:GOTO 560
490 REM
500 REM  DIR=1, OU SEJA, BORDA ESQUERDA
510 PRINT "SE DEIXO NA BORDA ESQUERDA"
520 IF PAS=0 THEN 540
530 PRINT "COM ";AN$(PAS);" A BORDO"
540 PRINT AN$(C);" COME ";AN$(V)
550 PRINT "NA BORDA ESQUERDA"
560 PRINT "PORTANTO TROQUE A JOGADA":PRINT:PRINT
570 FOR I=1 TO 1000 : NEXT I : REM  PAUSA
580 IF FM THEN GOSUB 1200 : REM  POP
585 IF IS<0 THEN PRINT "FINAL DE JOGO!" : END
590 GOTO 330 : REM  NOVA JOGADA
600 REM  RESUMO E CONCLUSAO
610 PRINT : PRINT : PRINT "TRAJETO CONCLUIDO (DAR <RET> PARA O
    RESUMO)" : GET R$
620 HOME : HTAB 14 : PRINT "RESUMINDO:" : PRINT
630 PRINT "JOGADA"; : HTAB 11 : PRINT "SE"; : HTAB 23
640 PRINT "DA"; : HTAB 32 : PRINT "PARA A"
650 PRINT "NUMERO TRANSPORTA";
660 HTAB 22 : PRINT "BORDA"; : HTAB 32 : PRINT "BORDA"
670 PRINT : SPEED=100
680 FOR I=1 TO IS : REM CONSULTA A PILHA
690 DIR=VAL(MID$(STK$(I),9,1))
700 PAS=VAL(MID$(STK$(I),10,1))
710 HTAB 3 : PRINT I; : HTAB 9 : PRINT AN$(PAS);
720 HTAB 22 : IF DIR THEN PRINT "DIREITA - ESQUERDA":GOTO 740
730 PRINT "ESQUERDA - DIREITA"
740 NEXT I
750 PRINT : PRINT : PRINT : FOR I=1 TO 2000 : NEXT I
760 PRINT "....E VIVERAM FELIZES E CONTENTES"
770 PRINT "DE FATO, NEM SEMPRE E FACIL"
```

Fig. 6. Continuação listagem do programa.

```
780 INVERSE : PRINT : HTAB 8 : PRINT "SALVAR CABRA E COUVE-FLOR" : NORMAL
790 HTAB 10 : PRINT "********###########" : SPEED=255
800 PRINT "OUTRA VOLTA?" : INPUT R$
810 IF LEFT$(R$,1)<>"S" THEN END
820 FIN=0 : GOSUB 1200 : REM  POP
830 GOTO 330
999 REM PUSH
1000 IS=IS+1
1010 REM  DADOS DE ESTADO EM CADEIA
1020 STK$(IS)=""
1030 FOR K=0 TO 3
1040 STK$(IS)=STK$(IS)+STR$(BD(K))
1050 NEXT K
1060 FOR K=0 TO 3
1070 STK$(IS)=STK$(IS)+STR$(BE(K))
1080 NEXT K
1090 STK$(IS)=STK$(IS)+STR$(DIR)
1100 STK$(IS)=STK$(IS)+STR$(PAS)
1110 STK$(IS)=STK$(IS)+STR$(PD)
1120 STK$(IS)=STK$(IS)+STR$(PE)
1130 P=0
1140 RETURN
1150 REM
1200 REM  POP
1210 FOR K=1 TO 4
1220 BD(K-1)=VAL(MID$(STK$(IS),K,1))
1230 NEXT K
1240 FOR K=5 TO 8
1250 BE(K-5)=VAL(MID$(STK$(IS),K,1))
1260 NEXT K
1270 DIR=VAL(MID$(STK$(IS),9,1))
1280 PAS=VAL(MID$(STK$(IS),10,1))
1290 PD=VAL(MID$(STK$(IS),11,1))
1300 PE=VAL(MID$(STK$(IS),12,1))
1310  REM  VOLTA SOBRE TEUS PASSOS
1340 IS=IS-1 : REM  DESINFLA PILHA
1350 P=1
```

Fig. 6. Continuação listagem do programa.

```
1360 RETURN
1990 REM  JOGADA NA BORDA DIREITA
2000 IF PAS=0 THEN 2030
2010 BD(PAS)=1 : REM  DESCARREGA PASSAGEIRO
2020 BD(0)=BD(0)+1
2030 IF P=0 THEN PD=0 ELSE PD=1
2040 PD=PD+1 : REM  NOVO PASSAGEIRO
2050 FM=(PD=4) : IF FM THEN RETURN
2060 IF PD=PE OR BD(PD)=0 THEN 2040
2070 PAS=PD : BD(PD)=0 : BD(0)=BD(0)-1
2080 RETURN
2090 REM
2490 REM  JOGADA NA BORDA ESQUERDA
2500 IF PAS=0   THEN 2530
2510 BE(PAS)=1 : BE(0)=BE(0)+1 : REM  DESCARREGA PASSAGEIRO
2520 IF BE(0)=3 THEN FIN=1 : RETURN
2530 IF P=0 THEN PE=0 : GOTO 2590
2540 REM  PROVA A VOLTAR VAZIO
2550 PE=PE+1 : REM  P=1, VOLTA A LEVAR ALGUEM
2560 FM=(PE=4) : IF FM THEN RETURN
2570 IF PE=PD OR BE(PE)=0 THEN 2550
2580 BE(PE)=0 : BE(0)=BE(0)-1
2590 PAS=PE
2600 RETURN
3010 P=0 : REM  HIPOTESE OTIMISTA
3020 IF DIR THEN 3100
3030 REM  ANALISE BORDA DIREITA
3040 IF BD(0)<>2 THEN RETURN
3050 IF BD(1) AND BD(3) THEN RETURN
3060 P=1 : REM  COMILONA SEGURA
3070 IF BD(1) AND BD(2) THEN C=1 : V=2
3080 IF BD(2) AND BD(3) THEN C=2 : V=3
3090 RETURN
3100 REM  ANALISE BORDA ESQUERDA
3110 IF BE(0)<>2 THEN RETURN
3120 IF BE(1) AND BE(3) THEN RETURN
3130 P=1 : REM  COMILONA SUPER SEGURA
```

Fig. 6. Continuação listagem do programa.

```
3140 IF BE(1) AND BE(2) THEN C=1 : V=2
3150 IF BE(2) AND BE(3) THEN C=2 : V=3
3160 RETURN
3170 REM
3180 REM
3990 REM  ILUSTRA A SITUACAO COMPLETA
4000 PRINT "AGORA AS COISAS ESTAO ASSIM:"
4010 PRINT "BORDA DIREITA:"; : IF BD(0)=0 THEN PRINT AN$(0) : GOTO 4070
4020 FOR I=1 TO 3
4030 IF BD(I) THEN PRINT AN$(I);" ";
4040 NEXT I
4050 PRINT : IF DIR OR NOT PAS THEN 4070
4060 PRINT "MAIS, AGORA, ";AN$(PAS) : PRINT
4070 PRINT "BORDA ESQUERDA:"; : IF BE(0)=0 THEN PRINT AN$(0) : GOTO 4110
4080 FOR I=1 TO 3
4090 IF BE(I) THEN PRINT AN$(I);" ";
4100 NEXT I
4110 PRINT : IF DIR OR NOT PAS THEN 4130
4120 PRINT "MAIS, AGORA, ";AN$(PAS) : PRINT
4130 RETURN
4140 REM
4150 REM %%%%%%%%%%%%%%%%%%%%%%%%%%%%
4160 REM %%%  FIM DO PROGRAMA  %%%
4170 REM %%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Fig. 6. Continuação listagem do programa.

Depois das linhas de inicialização 70-230, que damos por óbvias, especialmente se consultarmos a figura 4, aparece o POLLING DE USUARIO (linhas 240-300), também fácil de entender com a ajuda do pequeno fluxo homônio da figura da figura 5. Em ambos é apreciado o papel do indicador do primeiro giro SW: serve para saltar ao princípio da ilustração de uma (ainda inexistente) jogada precedente. É também um exrcício útili e simples seguir as mensagens das subrotinas 2500 e 4000 e os de justificações da "retirada", desdobradas para as duas bordas, nas linhas 430 a 570. O delineamento estrutral dos dados fica particularmente claro.

Por exemplo:

```
410 PRINT "MUMBLE,MUMBLE, ISTO NÃO SE PODE FAZER"
420 REM SI P = 1 EXPLICA RETIRADA
430 IF DIR THEN 510:REM EXPLICACAO PARA BORDA ESQ.
440 REM EXPLICACAO PARA BORDA DIR.
450 PRINT "SE CARREGO NA BARCA"; ANS (PAS)
460 PRINT ANS(M); "COME A";ANS(V)
470 PRINT "NA BORDA DIREITA"
480 GOTO 560
.....
560 PRINT "PORTANTO, MUDA DE JOGADA":PRINT:PRINT
```

OS parâmetros P (0 OU 1), J e V são gerados na subrotina (desdobrada, como de costume) AVALIAÇÃO COMILONA (GOSUB 3010). Nela o RETURN é imediato se não for deixado na borda dois sujeitos ou então se é tratado do caso inócuo lobo + couve-flor; em outra situação em C e V serão colocados os números correspondentes ao indivíduo "comedor" e a "vítima". Exemplo:

```
3070 IF BD(1) AND B(2) THEN J = 1:V = 2
```

Isto é,a presença simultânea do lobo e da cabra atribui aos animais 1 e 2 os papéis J e V respctivamente.

O miolo do problema

Os pares de subrotina 2000, 2500 (de gestão da jogada nas duas bordas) e 1200 (os conhecidos PUSH e POP são o verdadeiro miolo. Os primeiros dois são pareceidos em muitas coisas, por exemplo, que ambos inicialmente prevêm que, na condição NOT PAS, seja evitado um desembarque inútil mas que seja simulado incrementando em uma unidade BD(0) ou BE(0), que contarão os que permanecem em terra, situado em "1" em BD(PAS) ou BE(PAS). Depois será tentado com um novo passageiro mediante os índices PD ou PE. Por exemplo, na borda direita:

```
2040 PD = PD + 1:REM NOVO PASSAGEIRO
2050 FJ = (PD = 4):IF FJ THEN RETURN
2060 IF PD = PE OR BD(PD) = 0 THEN 2040
2070 PAS = PD:BD(PD) = 0:BD(0) = BD(0)-1
2080 RETURN
```

O pequeno loop das linhas 2040-2060 acaba por força, então

com a condição FJ = "on" (provocada, por sua vez, pela chegada a PD = 4, ou seja, por esgotar todas as jogadas teoricamente possíveis, desde "ninguem" até "couve-flor"; é o mesmo FM = Final de Movimento, que FJ = Final de Jogada) então com o descobrimento de um passageiro PD comível, que (linha 2070) será embarcado em PAS, anulando BD(PD) e diminuindo o conta-passageiros BD(0). Na linha 2060 aparece toda a casuítica do "passageiro impossível", isto é, porque é excluído o voltar transportar a quem acaba de desembarcar (dê conta que depois do primeiro miniloop é mais limpo usar PE em lugar de PAS) ou simplesmente, porque o passageiro em questão NÃO está. No que se refere à borda esquerda, a imperfeita especularidade (ou simetria do código se faz evidente desta forma:

```
2500 IF NOT PAS THEN 2530
.....
2530 IF P = 0 THEN PE = 0:GOTO 2590
2540 REM TENTA VOLTAR VAZIO
.....
2590 PAS = PE:REM EMBARQUE
2600 RETURN
```

Neste ponto deixaremos que o paciente leitor encontre no programa principal o caminho pelo qual se chega à já citada avaliação "comilona" que, por exemplo, poderia supor a renúncia à tentativa de voltar vazio porque a cabra comeria o couve-flor.

Chegou o momento de ver o PUSH e o POP da matriz STKS que reinicia a pilha (subrotinas 1000 e 1200). O mecanismo escolhido, não demasiadamente dificíl de rastrear na listagem, consiste em introduzir com o PUSH e recuperar mediante o POP (com a função MIDS) uma cadeia de estado STKS. Esta será resultado da concatenação sucessiva de: BD(0),...,BD(3), BE(0),... BE(3), DIR, PAS, PD e PE. O índice ES é incrementado ao princípio da subrotina de PUSH, enquanto que na de POP a instrução ES = ES-1 está colocada ao final. Em outros termos: com PUSH, antes que nada, é criado um novo espaço na matriz STKS; com o POP, ao contrário, são extraídos os dados ao princípio. Observamos, além disso, que a condição ES = 0 corresponde à pilha vazia.

Quanto ao porquê dos ajustes (semi-empíricos) DIR = NOT DIR e P = 1 das linha 1310 e 1350 se trata de um pequeno quebra-cabeças que deixamos ao leitor resolver, um pouco por tédio e um pouco por sadismo. Os mais adiantados saberão que a pilha completada mediante uma matriz (ou vetor) deve ser considerada co-

mo uma pseudopilha (opera com um índice em vez de com um ponteiro). A escolha era obrigatória em BASIC, pois esta linguagem não possui semelhantes estrutura. De qualquer modo, em nosso caso, sua relativa "impureza" nos permitiu uma pequena vantagem. De fato, uma memória LIFO não poderia ser recorrido deiretamente, a menos que fosse descarregada antes... em um uma atriz. Ao chegar à condição FIN "ligada" na linha 2520 (obviamente equivalente a BE(0) = 3) é inserida facilmente a fase de resumo (linha 600). A visualização de dados, como se pode deduzir e comprovar é do tipo seguinte:

```
RESUMINDO:
JOGADA          SE                DA          A
NÚMERO          TRANSPORTA A BORDA            BORDA
1               A CABRA           DIREITA     ESQUERDA
2 NINGUEM       ESQUERDA          DIREITA
3               O LOBO            DIREITA     ESQUERDA
.....
7               A CABRA           DIREITA     ESQUERDA
```

É muito simples dar-se conta de que varrendo a matriz STKS desde 2 até ES, se consegue reconstrutir as sete jogadas - mediante MIDS(STKS(I),10,1) - que estão guardadas e, inclusive os diferentes DIR (para escrever alternativamente "DIREITA" e "ESQUERDA".

Inclusive com uma reconstrução "em frio" (ainda que sempre será melhor experimentando com o programa, ou suas variantes, em um computador) fica mais simples seguir as mensagens e reflexões "em voz alta" que examinamos.

Na figura 7, segundo o que já adiantamos, estão representados os movimentos da pilha; por simplicidade foi representado, para cada nível, somente o dado PAS (expresso por seu nome: CABRA, COUVE-FLOR, NINGUEM, etc. para maior evidência). Notar-se-á que o primeiro passo comporta somente operações de push.

Um pequeno (ou grande?) paradoxo

Nos resta uma curiosidade: consegue nosso programa varrer inteiramente a árvore e encontrar o caminho das sete jogadas? Como?

Bem, a resposta à primeira pergunta é negativa. Nós que escrevemos temos que admitir nossa supresa quando, depois das não

poucas penas da investigação e colocação em ponto, nos encontramos, DEPOIS DA PRIMEIRA BUSCA, com êxito (como já foi visto) frente à desconcertante visualização de alguns dados que definem como via alternativa uma série de treze jogadas. Estas, à terceira busca, se convertiam em dezenove e iam aumentando a cada posterior resposta positiva à pergunta "OUTRA VOLTA?.

Temos que confessar que, até então, não havíamos traçamos nenhuma árvore do aparentemente jogo banal. Alem de rápida, esta escolha nos pareceu, de alguma forma "honrada". De fato, queríamos um programa que nos "ajudasse" a desfazer-nos de uma árvore como se nos fosse desconhecido a ambos. Se, com um certo sentido do "depois", nos remetemos ao exposto na figura 1 e com a ajuda da figura 7 (que evidência a sequência de push e pop desde o caso anterior) a explicação fica evidente rapidamente: o computador depois de ter encontrado a estratégia de sete jogadas "cb,n,l,cb,co,n,cb" na segunda prova terá chegado com dois pop ao penúltimo nó "E" e, como alternativa da escolha anterior "n", provará agora com "1", depois do qual seguirá sem dificuldade até o nó " + " situado ao final da figura. As jogadas se sucederam agora na seguinte ordem: cb,n,1,cb,co,1 em vez de n), cb,co,1,cb,co,n,cb, para um total de treze viagens.

O problema, por assim dizer, é duplo:

- com o programa visto serão encontradas soluções cada vez mais longas;
- o mecanismo, no entanto, é correto, pois a aparentemente esquálida árvore, na realidade é infinita.

Mas então, sem as oportunas modificações é impossível encontrar outra estratégia em sete jogadas? Bom, antes de tudo, existem duas soluções (um pouco grasseiras, isso sim). A primeira consiste em modificar a ordem dos passageiros (couve-flor, lobo, cabra, por exemplo) e, coerentemente, algumas das linhas referentes às jogadas e à avaliação da comilona (senão, o programa, impertubável, nos dirá: "mumble, mumble, a cabra come o lobo" ou outras frases parecidas). Este artifício permite, que nos seja evidente a outra solução: consiste em acrescentar outro pop à linha 820 (prove para acreditar):

```
820 FINAL = 0:GOSUB 1200:GOSUB 1200:REM DUPLO POP!
```

A partir daárvore da figura 1, pode dar-se conta que este brusco truque faz com que seja saltado o nó E já comentado e, dado

Fig. 7. Movimentos na pilha. Por simplicidade é representado somente o dado associado ao passageiro carregdo em cada caso. Na primeira viagem, de 7 passos, são realizadas somente operações de PUSH. No segundo giro (d) somente são assinalados, para maior clareza, as 5 primeiras e as 2 últimas condições. Depois de dois POP nós encontramos um nó onde, em lugar da jogada feita na primeira volta (NENHUM), é escolhida a segunda alternativa (LOBO). O resultado são 6 viagens a mais com relação ao primeiro caso. A árvore deste simples jogo é revelado ao infinito e não é portanto factível representá-lo de uma forma exaustiva.

que os outros dois de cima estão "saturados", fará com eles um POP automático, com o qual se poderá chegar ao nó D no qual o computador escolherá, em um segundo momento, couve-flor ao invés de lobo. À terceira volta, inclusive poderá subir a raiz e declarar o final do jogo por falta de outras vias otimizadas.

Estas soluções, por desgraça, não conseguem satisfazer nossos escrúpulos lógicos.

Existe uma solução correta; consiste em descartar vias cujo coprimento seja maior que a mínima encontrada anteriormente. Como de costume, deixaremos como exercício para os mais adiantados.

Terminamos agora com algumas considerações sobre as grandes dificuldades que são encontradas no novo e fascinante mundo da chamada Inteligência Artificial.

Inclusive deste pobre exemplo pode nascer a atroz dúvida de que seja certamente terrível, mas "organicamente" impossível, tentar remediar as contradições de fundo entre os mecanismos geraisde dedução e as exigências de problemas específicos. Fazemos notar aos mais perceptivos que, inclusive nas pequenas escolhas que fizemos anteriormente (como a de tentar voltar de vazio da borda esquerda), não se consegue entender como poderiam ser expressos, ou "sair por elas mesmas" com a magia de linguagens tipo LISP e Prolog. Em suma, um Resolutor Generalizado que seja ao menos um pouco eficiente e que evite (sem necessidade de intervenções penosas do programador) armadilhas como a dos percursos infinitos, está hoje em dia ainda nos limites da utopia.

Este tema, no entanto, sai de nossos muito reduzidos limites. Destacando que os investigadores no campo da Inteligência Artifiscal souberam escolher artifícios e macanismos que utilizam para evitar ou, pelo menos,limitar perigos semelhantes,devemos em todos os casos admitir que os desafios nestas categorias de problemas, são fascinantes e estimulan idéias posteriores Inclusive para os que se tem que apanhar com o BASIC.

CAPÍTULO V

ALGUNS CONSELHOS SOBRE QUESTÕES PRÁTICAS

O complicado tema de programar corretamente foi analisado até agora seguindo uma série de princípios. Estamos seguros de que alguém os classificará como "teóricos", mas não queremos iniciar uma polêmica entre os partidários da Teoria e os da Prática; portanto, nos limitamos a observar que, para ser teóricos, os exemplos que oferecemos foram ricos, concretos e abundantes. Se referiam a temas e problemas de delineamento com os quais não poucos dos habilíssimos seguidores da informática que encontramos hoje em dia, lutam penosamente sem resultado e sem saber, muitas vezes, nem sequer por onde começar.

Porisso insistimos em que preocupar-se obsessivamente com os algorítmos é um exercício penoso e frustrante muitas vezes, ainda que possa dar satisfações muito maiores que o exercício da PEEK-POKE-mania. Com esta etiqueta nos permitimos ironizar (perdão) sobre certos fanáticos do computador pessoal que abusam das úteis instruções PEEK e POKE do BASIC para ter acesso à linguagem máquina e às subrotinas do sistema operacional. Terão notado que nós as temos evitado com o maior cuidado, com o fim de obter a máxima "transportabilidade", isto é, a maior equivalência possível entre programas para um computador e outro. Devemos notar que enquanto tocamos o terreno pragmático nos damos de cara com as contradições mais atrozes: de fato, a transportabilidade é uma exigência prática, mas também o é aproveitar um sistema ao máximo de suas potencialidades e, no entan-

to, ambas são opostas...

É necessário reconhecer que, do ponto de vista operacional, foram deixadas à margem muitas coisas pertencentes à realidade das máquinas e dos problemas de todos os dias. Mas também foram ditas várias coisas, em modo indireto ou implícito, não somente neste livro como também no da programação estrutura (B.B.I nº 8); para não falar dos volumes sobre a linguagem BASIC (B.B.I nºs 5, 6, e 7) os quais nos remetemos para minucias tais como o uso do GET, como alternativa à INPUT (teram notado, que, por negligência, a instrução GET não foi utilizada aqui...). Mas existem outros problemas práticos que deveriam ser afrontados...em linha teórica (NÃO é uma finura, mas uma contradição real nos termos!). Aludimos, pelo menos, a duas grandes categorias:

- como organizar uma adequada interface com o usuário, ou seja, um "menú" apropriado para definir as opções a quem vai utilizá-lo, que fique claro, simples e, como dizem os americanos, "fool proof" (à prova de tontos);
- como tornar mais rápido e eficiente um programa, especialmente em relação ao microsistema que se possui e a linguagem utilizada.

Ambos os problemas são agravados no caso de programas muito longos, que procuram a seriedade multiplicando e complicando os dilemas. Estes nos remetem a uma questão delicada que, a grandes passadas, é expresso como segue:

Até que ponto é conveniente a adoção de "truques" que, com um uso astuto da máquina e de suas peculiaridades específicas a aproveitem da melhor maneira?

Uma contra-indicação - a da transportabilidade - já a vimos. Podem ser dadas sugestões e conselhos úteis com a finalidade de chegar a um compromisso razoável e válido em quase todos os ambientes sistemas.

Por outro lado, a maior dificuldade reside precisamente em proporcionar exemplos concretos.

Antes de passar a esmiuçar a segunda categoria de problemas, vamos nos permitir dar um par de apontamentos referentes a primeira categoria (interface com o usuário), apontamentos que cada um deverá acoplar a sua particular experiência pessoal através da prática da "prova e volta a provar", que nenhum tratado ou manual podem substituir.

No caso de um menú de escolhas múltiplas pode ficar mais claro para o interlocutor do programa que, em lugar de encontrar-se diante de habituais numerinhos:

1. Editar
2. Imprimir
3. Transferir
4. Carregar
5. Terminar
6. .....

encontre, ao contrário, como proposta as letras iniciais das diferentes opções, ou seja, algo assim como:

E = Editar
I = Imprimir
T = Transferir (ao disco)
C = Carregar (desde o disco)
A = Abandonar
X = ...

(dêem-se conta dos ajustes de sinônimos necessários para diferenciar as iniciais das opções).

A vantagem das ordens mnemônicas é que permitem que o usuário recorde facilmente o código que corresponde a cada opção sem necessidade de recorrer a mensagem do menú. São complementadas muito bem em linguagens que, como o Pascal, oferecem e estrutura CASE OF. E no BASIC? Aqui, a estrutura quase equivalente ON...GOSUB é um pouco menos potente, dado que trabalha comodamente com número, mas não aceita letras. Pode ser remediada mediante um pequeno vetor que contenha as diferentes iniciais. Vamos supor que o chamamos INIC$ e que R$ contém a inicial escolhida pelo usuário; a resposta do computador poderia funcionar mediante as linhas BASIC que segue:

```
300 ENCU = FALSO
310 FOR I = 1 TO NUMOPC
320 IF R$ = INIC$(I) THEN ENCU = VERDADEIRO:K = 1:I = NUMOPC
330 NEXT I
340 IF NOT ENCU THEN GOTO X
350 ON K GOSUB R,S,T,.....
```

É óbvio que, r,s,t, etc., serão os números de linhas das diferentes subrotinas de resposta, enquanto que x será o da linha que contém INPUT. Assim temos satisfeito (naturalmente é um exemplo muito simples) o princípio de reduzir as possibilidades de erro por parte do usuário. Em programas mais longos e/ou importantes será oportuno prever, também, barreiras defensivas contra as mais estranhas faltas de atenção (apertar teclas críticas, fazer manobras que façam voltar ao sistema operacional, etc.), sempre possíveis no transcurso de sessões longas e fatigantes. Afortunadamente, em todos os dialetos BASIC existe a cômoda ins-

trução ON ERROR GOSUB (ou GOTO) que põe remédio a muitas de tais desgraças (para aprender suas diferentes modalidades de uso os remetemos aos manuais dos computadores e aos volumes dedicados a esta linguagem, onde são encontrados descritos adequadamente).

A propósito de teclas e menús devemos fazer-lhes outra recomendação referente ao caso de programas nos quais são encontrados vários menús e submenús, ou seja, aqueles organizados "por árvore": tente adotar sempre a mesma semântica em todas as circunstâncias, evitando que em um menú Imprimir seja expresso com "I" enquanto que em outro com letra "P" (de Print) ou com o número "2". Outro conselho: utilize as teclas combinadas com a de Control e, para as condições de "quit" (abandono de programas), usem sistematicamente a tecla de Escape.

Especialmente em programas de cálculo científico e técnico será uma boa norma introduzir controles de congruência de variáveis e ordem de magnitudes, mas tenha muito cuidado com as chamadas "modalidades de restabelecimento". Vejamos um exemplo tonto, que é mais "caricaturesco":

```
1500 IF DADO  INF OU DATO  SUB THEN END
```

Aqui, se o desafortunado usuário se equivoca introduzindo um DADO que está fora dos limites definidos por INF e SUP, este é penalizado gravemente com a paralização do programa e, o que é mais grave, não somente com a perda dos dados gerais pelo programa, mas também com a de todos os introduzidos até o momento. Se trata de sadismo? Talvez, mas estamos nos referindo a um caso real (naturalmente a condição de paralização surge de uma forma mais indireta e enfadonha): um senhor comprou um lote de programas de cálculo de engenharia e, dando-se conta de que havia algo raro, recusou efetuar o pagamento até que o problema não fosse resolvido; acabaram no Tribunal. Não sabemos como foi resolvido, mas se estivéssemos no lugar do juiz teríamos dado toda a razão ao comprador.

Acabamos recomendando-lhes que não tomem demasiadamente em conta os ouros falsos cenográficos. Ainda que às vezes não é demais um mínimo de videografia e algum efeito sonoro agradável, sobretudo para chamar a atenção no momento adequado, ao que se deve de prestar a máxima atenção é aos possíveis defeitos. Somente a experiência ensina a corrigí-los e, possivelmente, a prevení-los.

Nos próximos dois capítulos trataremos bastante amplamente

dos problemas de velocidade e eficiência que delinea o BASIC (a linguagem-caracol). Como é inevitável, uma vez que se trata de um tema muito concreto, faremos referência a um computador pessoal em carne e osso (perdão, em plástico e silício). Portanto como ilustração e como um ato de imparcialidade para com os nossos fabricantes, resolvemos escolher o Commodore 64, também chamado C-64 de grande difusão em outros países, porém não fabricado no Brasil. A "grosso modo" o tratamento é válido para a grande maioria dos computadores do mercado. Por exemplo, os problemas da "informação inservível" nas cadeias podem ser encontrados, praticamente sem mudanças, em qualquer computador. Passando as referências ao mapa de memória de seu próprio microcomputador será fácil para todos reciclarem todos os conceitos (geralmente basta adotar as direções dos PEEK e POKE àquelas em vigor em nosso sistema).

CAPÍTULO VI

O BASIC, UMA LINGUAGEM-CARACOL

Interpretação: uma vantagem que sai cara

A enorme popularidade da linguagem BASIC no mundo dos pequenos sistemas (computadores domésticos e pessoais) está ligada sobretudo ao fato de ser, quase sempre, uma linguagem interpretada. Este adjetivo, como vocês bem sabem, quer dizer que cada instrução - IF, GOTO, sentenças aritméticas, etc. - é introduzida diretamente à linguagem máquina (de agora em diante escreveremos l.m.) no momento da execução ("run time") cada vez que esta é levada a cabo, sem necessidade de compilações ou outros processos preliminares.

Aparentemente o computador se comporta como uma máquina BASIC: compreende e executa de maneira imediata ordens em liguagem evoluída, mas na realidade, cada vez que o faz realiza antes a tradução, ainda que sem deixar rastro ao final. Do ponto de vista da interação homem-máquina se trata, sem dúvida, de uma vantagem notável, porque o sistema pode ser manejado diretamente em um código simbólico (de alto nível, próximo à linguagem humana e não surgem os inconvenientes (e a perda de tempo) que traz consigo a intervenção de um programa compilador e o fato de ter duas versões de um mesmo programa (do ponto de vista lógico o programa "fonte"(source program, o que podemos editar e listar), e o programa "objeto" (object program, o que está em l.m. traduzido pelo compilador ao código máquina do microprocessador que constitui a CPU do sistema).

Com o BASIC interpretado são executados mais facilmente, entre outras coisas: controles, acréscimos, modificações (em qualquer momento) e a busca e eliminação de erros fica mais simples e rápida.

No entanto esta vantagem se vê fortemente penalizada, porque se paga em termos de diminuição da velocidade de execução. Pode acontecer, de fato, que o tempo de interpretação (que, repetimos, é invertido todas as vezes que é executada cada instrução e não de uma vez por todas como acontece com as linguagens compiladas) fique muito maior que o correspondente a sua execução efetiva. O inconveniente existe, inclusive, nas instruções mais simples.

Mas a popularidade do BASIC também reside, no tratamento das variáveis e no manejo da memória interna: o programador não tem que se preocupar com nada, já que se encontra inteiramente sob a supervisão do intérprete. Uma vez mais, é obtida uma prestação muito cômoda mas que retarda o processamento. Em geral, não estamos demasiadamente dispostos a renunciar a estas vantagens, pelo menos dentro do âmbito das autênticas aplicações dos computadores domésticos.

Afortunadamente, esta lentidão, derivada do processamento interpretativo, pode ser evitada, pelo menos em parte, com uma série de pequenos recursos, cuja descrição é nosso próximo objetivo. Um muito habitual é a inserção de pequenas rotinas de l.m. dentro dos mesmos programas BASIC; os códigos correspondentes à posição de memória inicial são introduzidos mediante POKE em uma determinada situação de memória, para ser recuperados no momento oportuno com instruções específicas, como a USR e a SYS do C64, a CALL de outros microsistemas, etc., que provocam saltos à subrotina em l.m. Esta técnica suspende momentaneamente a interpretação e passa à execução direta em l.m., permitindo assim aumentar a velocidade em certas passagens críticas de um programa. Não vamos nos estender sobre elas por dois motivos:

- está abundantemente descrita em todos os manuais e livros;
- anulam a transportabilidade de um programa: entre máquinas que adotam o mesmo dialeto BASIC mas que são baseadas em CPU diferentes é impossível a transferibilidade de programas com SYS ou CALL, enquanto que com CPU iguais mas com computadores desenhados de forma diferente, pode ser conseguido somente com modificações radicais.

Nosso objetivo é descobrir, sobretudo de forma experimental, tudo o que se pode fazer permanecendo dentro do BASIC vulgar e (quase) standard: as vantagens em termos de velocidade e, de alguma maneira, de eficiência serão pequenas se as tomamos de uma em uma, mas quando estão em jogo programas bastante longos e/ou loops com inúmeras iterações, o tempo que é economizado pode chegar a ser importante.

Tanto os inconvenientes como os remédios sugeridos, serão ilustrados quando entrarmos em detalhes sobre o mecanismo da interpretação e sobre os critérios de memorização das variáveis na memória RAM disponível. Quanto às provas que os convidamos a realizar diretamente, serão executados por nós em um sistema Commodore 64, mas todos poderão fazê-las, com fáceis modificações, no computador que tiver em casa.

Em relação com os resultados das provas que levamos a cabo, em linhas gerais podem ser consideradas válidas (naturalmente para igual freqüência do relógio interno) para todas as máquinas que utilizam como CPU o microprocessador 6502 e derivados (o 6510 no caso do C64).

Para fazer comparações entre a velocidade de distintos programas não existe nada melhor que usar o relógio interno do C64; isto vale para outras máquinas que possuam também relógio interno, em outros casos terá que ajustar-se com um cronômetro externo. No Commodore 64 o valor do tempo pode ser encontrado utilizando duas variáveis especiais (reservadas): TI e TI$, a primeira numérica e a segunda de tipo cadeia (alfanumérica).

Dado que TI$ tem uma resolução bastante escassa (um segundo) não se adapta aos nossos fins. Ao contrário TI pode ser considerada suficientemente precisa, sendo capaz de discernir para 1/60 de segundo, ou seja, a 16,6 milésimos de segundo. A verdade é que os tempos de execução das instruções BASIC são inferiores, mas isto não tem uma importância excessiva, pois o que precisamos não é tanto o medir com exatidão a duração, mas comparar os resultados obtidos com distintas técnicas de programação. Antes de tudo trataremos que o número de instruções executáveis seja suficientemente elevado, o que contribuirá para acentuar as diferenças, tornando-as mais apreciáveis.

Recordamos que a variável reservada "TI" não pode ser inicializada desde o software. Mas não importa, será suficiente copiar ao princípio seu valor em uma variável auxiliar (ou depósito, como quiser dizer); ao final, o valor do tempo transcorrido será obtido com a diferença entre o valor final do TI e o guardado inicialmente.

Memória necessária para o programa BASIC

Um aspecto muito importante que deve ser levado em consideração quando se quer utilizar da melhor forma possível um computador e, sem dúvida, a quantidade de memória necessária para que um programa possa trabalhar. Utilizando o BASIC interpretado deverá ter em conta tanto o espaço de RAM ocupado pelo código (o verdadeiro e próprio programa) como o utilizado pelas variáveis em jogo. Aparentemente, um programa BASIC ocupa tanto espaço como número de caracteres que compõem suas linhas; em outros termos, precisa tantos bytes quantas teclas tenha sido utilizadas na redação do programa. Na realidade as coisas caminham de forma um pouco distinta.

Vejamos como e por que. Para maior comidade do leitor, ilustramos na Figura 1 o mapa de memória do C64, com particular atenção à utilização que faz o BASIC das primeiras 256 posições de memória (a chamada página zero).

Nos daremos conta muito rápido de que é fácil verificar quanta memória é utilizada e como. Uma vez ligado o computador escreva:

```
PRINT PEEK (44)*256 + PEEK(45)
```

A resposta será 2049, isto é, a direção na qual a primeira instrução do programa BASIC é introduzido. Em outros computadores pessoais as localizações da página zero que apontam a este lugar de entrada, não serão 44 e 45, mas o processamento será o mesmo; somente necessitamos saber que são dois (44 e 45 aqui) os bytes necessários para conter o valor da direção, em um (44) é guardada a parte mais significativa (256 vezes mais) e no outro (45) a menor.

Depois, sem fazer mais nada, podemos pedir a direção de final do programa fazendo:

```
PRINT PEEK(46)*256 + PEEK(47)
```

(Aos bytes 46 e 47 são aplicados os mesmos comentários realizados antes com 44 e 45).

O lógico seria esperar o mesmo número de antes, já que não introduzimos nenhuma linha BASIC. Ao contrário obtivemos o valor 2051. Não se assustem, não acontece nada estranho; de fato, dois bytes estão comprometidos desde o princípio pelo BASIC para guardar o valor binário zero, indicador do final do programa (00

POSIÇÃO		
DEC	HEX	
0	0000	Registro direção. dados do 6510
1	0001	Registro 8 bits de entrada/saida do 6510
2	0002	Não usado
40-41	0028-0029	38÷42 Area usada para la multiplicação real
42	002A	
43 44	002B 002C	Ponteiro principio do texto BASIC
45 46	002D 002E	Ponteiro final programa e principio variaveis
47 48	002F 0030	Ponteiro principio vetores (matrizes)
49 50	0031 0032	Ponteiro final vetores (+1)
51 52	0033 0034	Ponteiro principio variaveis cadeia
53 54	0035 0036	Ponteiro final variaveis cadeia
55 56	0037 0038	Ponteiro no limite da memoria para programas BASIC
136÷140	0088÷008C	Aqui da correspondentes localizacões son 139÷143
197	00C5	Valor de la tecla pulsada actualmente
198	00C6	Número de caracteres no buffer do teclado (cauda)
631÷640	0277÷0280	Cauda do buffer do teclado (F1F0)
650	028A	Indicador programavel de repetição para as teclas (0 = cursor, 128 = todas)
653	028D	Indicador tecla SHIFT/CTRL/
828÷1019	033C÷03FB	Buffer de E/S da fita
1020÷1023	03FC÷03FF	Livres
1024÷2023	0400÷07E7	Area de memoria de tela (1024 bytes) matriz tela (25 linhas × 40 colunas)
2040÷2047	07FB÷07FF	Ponteiros aos dados de animação

POSIÇÃO DEC	POSIÇÃO HEX	
2048 ÷ 40959	0800 ÷ 9FFF	Area RAM normal dos programas BASIC, com ponteiros é possível "inserir" na área (32768-40959) cartuchos ROM (8 k)
40960 ÷ 49151	A000 ÷ BFFF	8 k ROM do intérprete BASIC (é possível usar RAM aqui para utilizá-la com linguagem máquina)
53248 ÷ 54271	D000 ÷ D3FF	Controlador interface tela (VIC II) MOS 6566
54272 ÷ 55295	D400 ÷ D7FF	Interface de som (SID) MOS 6581 3 osciladores independentes e programáveis com 4 classes, de ondas, 9 oitavas, etc.

Fig. 1. Mapa de memória do Commodore 64. O texto faz referência às posições da página zero. Pelo que respeita a organização dos tipos de dados (inteiros, reais, de cadeia) e sua distribuição na memória, o sistema ilustrado, referido ao C64, é parecido, com poucas mudanças, aos que usam a maioria dos computadores pessoais.

00 = final de programa).

Se agora introduzimos uma simples linha podemos comprovar imediatamente quanta memória ocupa:

```
1 REM ABCDEFGHIJK
```

A resposta obtida com o PRINT anterior é 2067, que corresponde a 18 bytes (2067-2049) ocupados desde o princípio, e não a 15 como se poderia supor. Tentemos então visualizar o conteúdo de todas estas posições ocupadas:

```
FOR K = 2049 TO 2067: PRINT CHR$(PEEK(K));:NEXT K
```

A resposta é:

```
ABCDEFGHIJK
```

isto é, em aparência somente existe onze caracteres. Como é isto? Evidentemente, os outros 7 contém valores binários não visualizáveis na tela. Fazendo a prova com outros tipos de instruções

a situação não muda muito. Em particular, os nomes das instruções (como REM) e os números de linha parecem ter desaparecidos ou ter sido substituídos por estranhos caracteres gráficos. A explicação deste aparente mistério reside na forma na qual o intérprete memoriza o programa

Cada uma das linhas do programa é representada na memória com a estrutura que segue (ver Figura 2):

- 2 bytes que contém a direção da seguinte instrução;
- 2 bytes que contém o número de linha;
- um byte par cada instrução BASIC, representada segundo um código compacto que alguns chamam "token";
- N bytes tal e como tenham sido escritos desde o teclado;
- 2 bytes contendo zeros para assinalar o final do programa, depois da última instrução.

São explicados assim os setes bytes aparentemente desaparecidos que vimos anteriormente. Pode ser concluído, pois, que em princípio a memória que ocupa qualquer programa está dada pelo número de caracteres teclados (contabilizando, no entanto, um só byte para os códigos operacionais tipo REM, INPUT, DA-

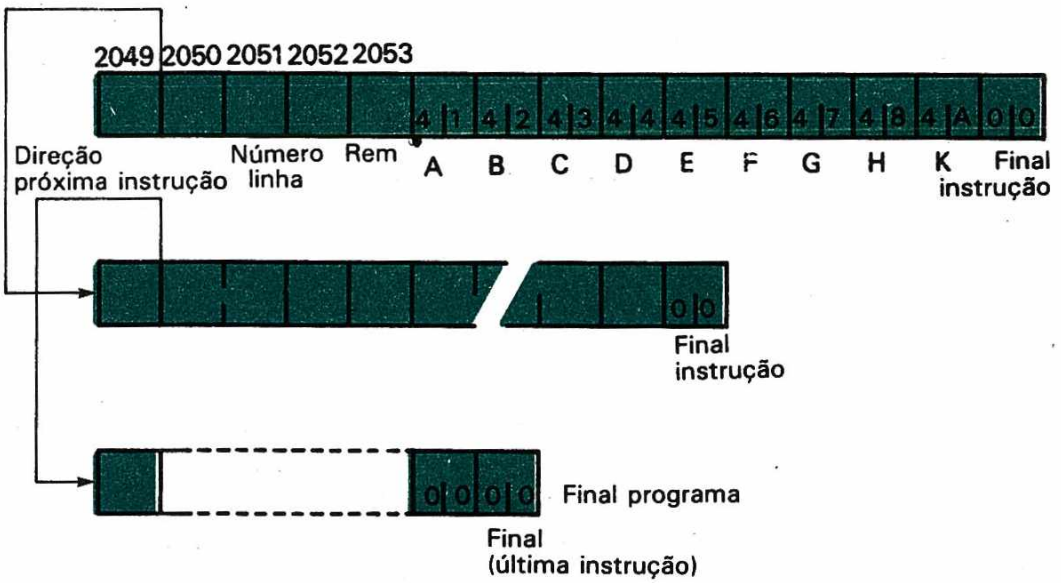


Fig. 2. Estrutura e sucessão das instruções BASIC. Para maior comodidade foram suprimidos os conteúdos das distintas posições; por exemplo, o "token" ASCII do código REM.

TA, +, —, *, etc.), acrescentando 4 bytes para cada linha e contando os dois finais de clausura

Consequentemente, se há problemas de espaço, pode tentar:

- compactar várias linhas em uma só, ganhando assim cada vez 3 bytes (um é sempre necessário para os dois pontos de separação);
- eliminar todos os espaços de separação inúteis

Deve ficar bem claro que esta técnica, indispensável com programas muito longos destinados a computadores domésticos com não muita memória, não é compatível com exigências de clareza. Existe além disso alguns dialetos que limitam sua aplicação. Por exemplo, no BASIC Microsoft os espaços de seperação entre comando e operadores são obrigatórios (escrevendo PRINTAS é obtido da máquina um "syntax error").

Espaço para as variáveis, cadeias e matrizes.

Uma escolha inteligente da natureza e número das variáveis a utilizar proporciona também resultados surpreendentes. Sempre com o computador recém ligado, examinemos o princípio e o fim da área destinada às variàveis, situada logo depois do final do programa. Repetindo:

```
PRINT PEEK(46)*256 + PEEK(45)
```

a resposta obtida é 2051, dado que neste momento não está presente nenhum programa, e com:

```
PRINT PEEK(48)*256 + PEEK(47)
```

a resposta continua sendo 2051, já que não foi definido ainda nenhuma variável.

Introduzamos agora:

```
AA = 1

PRINT PEEK(48)*256 + PEEK(47)
```

Esta vez será obtido o valor 2058. Se introduzimos com X = 12345678901 teremos que o ponteiro de final da área de variáveis alcança o valor 2065; a imediata conclusão é que uma va-

riável numérica, qualquer que seja o valor que lhe é atribuído, ocupa sempre 7 bytes de RAM.

O motivo disto reside no fato de que o BASIC, para este tipo de dados, adota a notação de ponto flutuante (floating point), com nove cifras significativas e exponente; porisso são necessários 5 bytes para conter essas 9 cifras mais sinal e exponente, aos quais deverão ser acrescentados dois bytes para o nome simbólico (em outros dialetos BASIC nos quais o nome das variáveis pode estar formado por mais de dois caracteres ASCII esta regra, por outro lado muito comum nos computadores mais usuais não vale). Se, como no último caso, este está formado por uma só letra o intérprete acrescenta, por sua conta e risco, um espaço (blank). Consequentemente a única economia que é conseguida usando nomes de um caracter reside no comprimento do programa.

Com a ordem CLR (CLEAR, em outros dialetos) são anuladas todas as variáveis definidas, o que é facilmente comprovável: o ponteiro de final de variáveis é levado até 2051. Um conselho prático: levar para diante artificialmente este ponteiro de final de variáveis (mediante uma ordem POKE) pode ser um método astuto e simples para recuperar os dados perdidos inadvertidamente.

O fato de que a zona reservada às variáveis inicie imediatamente depois do final do programa explica porque, acrescentando instruções, os valores das primeiras variáveis são destruídos irremediavelmente e como, com a técnica de "overlay" (segmentação de programas que não podem estar contidos inteiramente na RAM disponível) podem ser carregados somente programas mais curtos que o primeiro da cadeia.

A versão BASIC adotada pelo C64 não reserva uma área específica para as variáveis inteiras (do tipo "integer") e, portanto, também estas ocupam sete bytes. É uma verdadeira pena, porque seu limite de amplitude (não superior ao valor 65535, máximo alcançável com 16 bits na notação binária adotada) permitiria a utilização de somente quatro bytes: dois para o nome e dois para o valor. Provavelmente, a boa disponibilidade de memória induziu aos projetistas a não cuidar deste aspecto, presente em outras máquinas do Commodore.

Quanto às variáveis alfanuméricas (cadeias) também estas são memorizadas como as anteriores, e assim ocupam sete bytes em lugar de cinco, que seriam suficientes: além do nome (dois bytes) é memorizado o comprimento (um byte) e a direção de começo de cadeia (outros dois bytes). De fato, o comprimento das cadeias é variável e imprevisível a priori, porque não seria possível memorizá-las diretamente ao lado do nome, somente com contínuos des-

locamentos das variáveis subseqüentes em caso de mudar o conteúdo da cadeia. A área na qual são memorizados os valores das cadeias começa ao final da memória disponível e se dirige para a já ocupada. Este tipo de organização dá lugar a uma gestão eficiente e veloz, mas tampouco é imune a todos os problemas, como veremos mais adiante.

Agora, alguém estará fazendo esta inquietante pergunta: como distingue o BASIC os diferentes tipos de variáveis, dados que estas são memorizadas todas juntas, colocadas simplesmente na ordem temporal de definição? Elementar: o nome das variáveis numéricas é memorizada com o correspondente código ASCII, enquanto que no caso das variáveis inteiras é acrescentado 128 à primeira letra e, no das cadeias, é acrescentado 128 à segunda letra. Dado que o código ASCII somente representa caracteres de código 0-127 (0-01111111 em binário) o acrescentar 128 a qualquer caracter ASCII equivale a colocar em "1" seu bit mais significativo (o situado mais a esquerda). Por exemplo, o nome PE (0101 0000 0100 0101 em binário) ficará memorizado como:

— em ponto flutuante:
  0101 0000 0100 0101
— inteira:
  1101 0000 0100 0101
— cadeia:
  0101 0000 1100 0101

O método é simples mas pouco rápido, já que a procura de cada uma é entorpecida pela presença de todas as demais e não somente pelas do mesmo tipo. Na Figura 3 está ilustrada a organização das variáveis simples dos três tipos (ponto flutuante, inteira e alfanumérica).

Imediatamente depois do final das variáveis simples começa a zona dos vetores (arrays). Para comprovar experimentalmente sua colocação (e inclusive antes de ver a Figura 4, que ilustra a estrutura de sua organização) convém esta vez atuar de maneira muito diferente de como temos feito até agora, valendo-nos de um simples programa, útil para cada tipo de vetor. Este é:

```
1 CRL:REM LIMPEZA DA MEMÓRIA DE VARIÁVEIS
10 IA = 0:FA = 0:REM PREDEFINIÇÃO VARIÁVEIS DE TRABALHO
20 DIM RR(300):REM ARRAY DEFINIDO
30 IA = PEEK(48) * 256 + PEEK(47):REM PRINCIPIO AREA ARRAY
40 FA = PEEK(50) * 256 + PEEK(49):REM FINAL AREA ARRAY
50 PRINT FA,IA,FA-IA
```

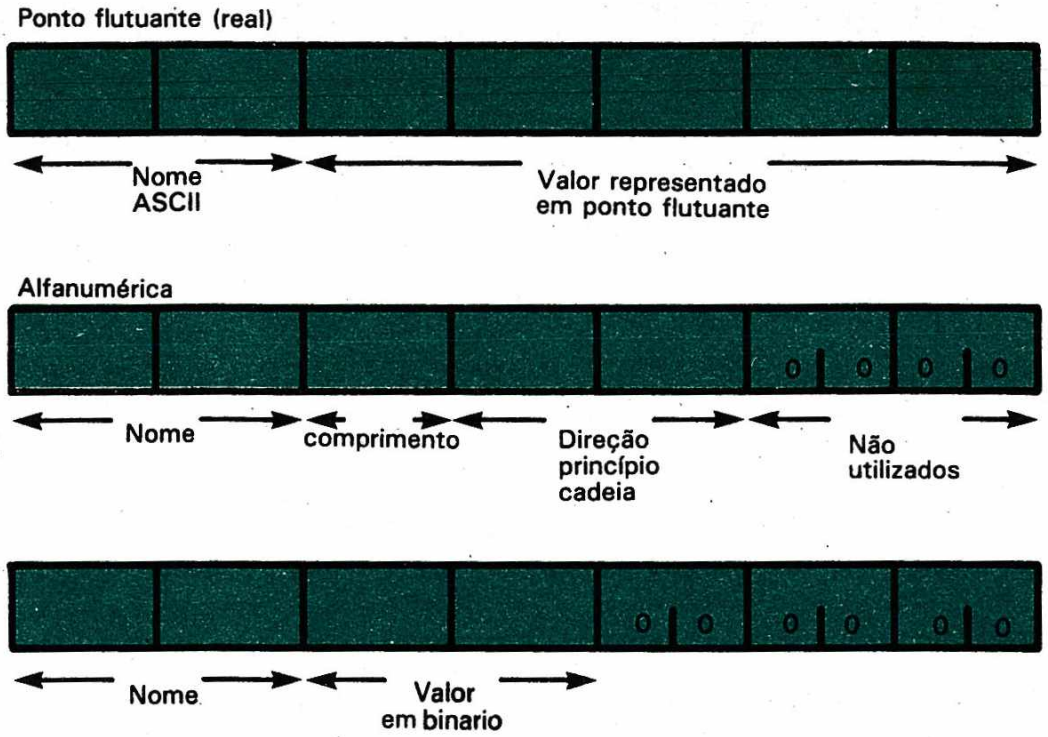


Fig. 3. Organização das variáveis (ao final do programa BASIC). No caso das cadeias, ao nome seguem o comprimento e um ponteiro que indica a direção na qual a cadeia tem seu princípio. No C64 os três tipos ocupam 7 bytes (em outros dialetos BASIC nem sempre acontece isto).

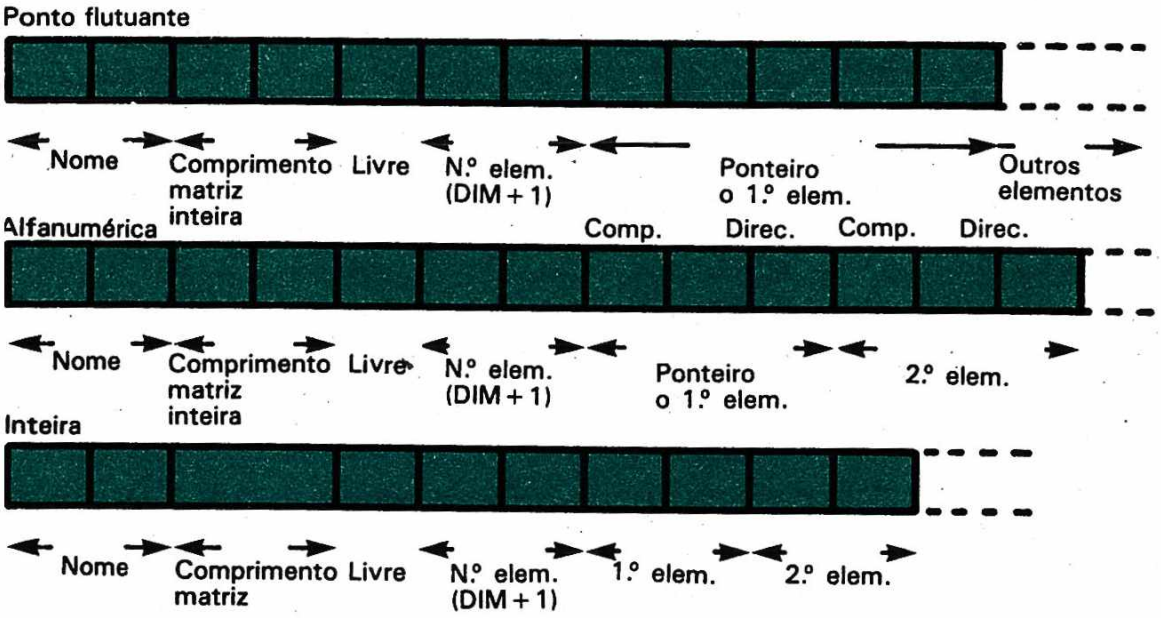


Fig. 4. Organização dos distintos tipos de matrizes.

Uma vez executado, o programa escreve como terceiro dado a memória ocupada pela matriz, igual neste caso a 1512 bytes. Sendo os elementos em jogo 301 (recordem que também existe o elemento de índice zero) se deduz, sem muita dificuldade, que cada um deles emprega cinco bytes, como é necessário para uma variável em ponto flutuante, mais sete bytes de cabeceira que contém o nome, o número de elementos e outras informações para uso do intérprete.

Mudando agora na linha 20 "RR" por "RR%", que indica matrizes de números inteiros, o resultado será 609. Isto é: sem mudar o número de elementos (301) cada um destes ocupa agora somente dois bytes, enquanto restam por acrescentar ao total os sete bytes do encabeçamento comum. Em outros termos, isto significa que a utilização de variáveis de tipo inteiro acarreta uma economia considerável de memória.

Definindo uma matriz alfanumérica, colocando na linha 20 RR$ (300), o resultado visualizado é 910, o que significa que cada elemento deste tipo de matriz ocupa somente três bytes. O resultado, sobretudo tendo em conta o fato de que uma cadeia pode ter um comprimento de até 256 caracteres, pode parecer paradóxico, mas somente às pessoas pouco atentas. Inclusive estas não terão dificuldades em compreender que, também no caso das matrizes, as cadeias são encontradas na parte alta da memória, enquanto que no espaço de vetor cada elemento indica tão somente o comprimento (um byte, pelo qual o comprimento máximo é de 256 caracteres) e o ponteiro que indica começo da cadeia efetiva. Desta forma, com as cadeias, a ocupação de RAM cresce proporcionalmente segundo a cada elemento - vazio no momento do DIM inicial - no qual vai associando uma cadeia real (não vazia).

Concluindo, se puder trabalhar com números inteiros inferiores a 65533 convém, sem dúvida, adotar matrizes de tipo inteiro, podendo assim quase triplicar a memória disponível para dados. Ao contrário - pelo menos no caso de C64 como já foi visto - não existe vantagem alguma, em termos de espaço ganho, ao usar variáveis inteiras em lugar das de ponto flutuante (em outras versões BASIC, no entanto, existem).

Memória e velocidade

Para comprovar experimentalmente como BASIC aglutina as cadeias e, em particular, como influi seu comprimento nos tempos de execução, propomos usar o programa seguinte com cronômetro incorporado. Nele intercambiaremos três mil vezes o con-

teúdo de duas cadeias entre si, recorrendo a uma cadeia intermediária:

```
1 CLR
10 A$ = "PRIMEIRA CADEIA":B$ = "SEGUNDA CADEIA":C$ = "  "
20 TD = TI
30 FOR K = 0 TO 3000
40 C$ = BS
50 B$ = AS
60 A$ = CS
70 NEXT K
80 PRINT TI-TD
```

É obtido um valor, correspondente a múltiplos de 1/60 segundos, igual a 965. Provemos a reduzir o comprimento das variáveis, fazendo, na linha 10, A$ = "PRIMEIRA" e B$ = "SEGUNDA".

Antes de proceder a modificação e à nova medida, respondam: mudará a duração do loop? A reposta é negativa, contrariamente ao que poderia ser imaginado, mas a explicação deste paradoxo não é muito difícil. De fato, bastará refletir novamente sobre a maneira na qual o intérprete BASIC situa as variáveis alfanuméricas: na área reservada a tais variáveis não está colocado o conteúdo da cadeia, mas somente seu nome, comprimento e direção a partir da qual está escrita a cadeia. Mudar os dados alfanuméricos, portanto, simplesmente significa mudar tais comprimentos e direções (ponteiros), no total seis bytes, sem necessidade de nenhuma outra operação. É evidente que para este procedimento o tempo totalmente independente do comprimento das duas cadeias, como nos mostram claramente os experimentos práticos.

Por outro lado, a técnica que emprega o intérprete, ainda que muito eficaz, pode produzir sérios incovenientes quando se tem uma utilização intensiva de dados alfanuméricos em programas de uma certa complexidade. Procedendo desta maneira, de fato, o BASIC provoca um enorme desperdício de memória, com a pretensão de obter uma velocidade muito superior, mas por desgraça nos encontramos com um autêntico rendimento de contas: o tempo que foi economizado até esse momento deve ser devolvido de uma vez e com interesses.

Mas procedamos com ordem, tentando compreender mais de perto como atua exatamente o BASIC com as variáveis alfanuméricas. Com referência à figura 1 do tópico anterior, ou seja, ao mapa da página zero do Commodore 64, teclemos o seguinte programa:

```
10 CLR
20 REM FINAL DA MEMÓRIA DISPONÍVEL
30 TM = PEEK(55) + PEEK(56) * 256
40 REM FINAL DA ÁREA DE VARIÁVEIS
50 REM (CRESCE PARA CIMA)
60 FA = PEEK(49) + PEEK(50) * 256
70 REM FINAL DA ÁREA DE CADEIAS
80 REM (CRESCE PARA BAIXO)
90 BS = PEEK(51) + PEEK(52) * 256
100 PRINT FA,BS,TM
110 PRINT "MEMÓRIA OCUPADA PELAS CADEIAS";
120 PRINT TM-BS
130 "MEMÓRIA TODAVIA DISPONÍVEL";
140 PRINT BS-FA "BYTES"
```

Colocando em andamento o programa obteremos:

```
2400   40960   40960
MEMÓRIA OCUPADA PELAS CADEIAS  0 BYTES
MEMÓRIA TODAVIA DISPONÍVEL 38560 BYTES
```

É obvio que a área das cadeias não contém nada, desde o momento em que nenhuma variável alfanumérica tenha sido definida.

Vamos inserir agora a linha seguinte, que introduz três variáveis de cadeia, mas todas iguais à cadeia vazia:

```
15 A$ = "   ":B$ = "   ":C$ = "   "
```

Desta vez, o programa assim modificado, nós dá:

```
2447   40960   40960
MEMÓRIA OCUPADA PELAS CADEIAS 0 BYTES
MEMÓRIA TODAVIA DISPONÍVEL 38517 BYTES
```

A diminuição da RAM disponível tem que ser imputada, em parte, ao alargamento do programa, devido à inserção da nova linha 15 e, em parte, ao espaço requerido para memorização das variáveis.

Agora procedamos a atribuir um valor diferente de "vazio" às três cadeias, modificando como segue a linha 15:

```
15 A$ = "PRIMEIRA": B$ = "SEGUNDA":C$ = "TERCEIRA"
```

O programa assinalará agora o primeiro valor igual a 2460,

enquanto que a memória disponível agora é igual a 38445; o par de valores 40960 e 40960, estranhamente, permanece invariável e continua sendo nula a memória ocupada pelas cadeias. Aparentemente, os três valores de cadeia introduzidos não aparecem por nenhuma parte. A chave do enigma é descoberta logo; uma vez mais, o BASIC se mostra muito astuto; aproveita que os conteúdos das variáveis alfanuméricas estão contidos no mesmo programa BASIC (que, recordamos aos esquecidos, reside também em memória). Assim, os ponteiros das três variáveis A$, B$ e C$, são limitados a apontar a estas situações do programa - aquelas nas quais estão materialmete escritos os valores "PRIMEIRA", "SEGUNDA" e "TERCEIRA", de nossa linha 15 - e não é requerido nenhum espaço extra. Mas, o que ocorre se, manipulamos cadeias para criar outras novas cujo conteúdo não foi definido a priori? A resposta é obvia: o intérprete não pode evitar reservar certo espaço de memória para este exercício. Para convencer-nos experimentalmente acrescentamos esta outra linha:

```
25 C$ = A$ + B$ + C$
```

e a resposta será:

```
2476   40931   40960

MEMÓRIA OCUPADA PELAS CADEIAS 29 BYTES
MEMÓRIA TODAVIA DISPONÍVEL 38455 BYTES
```

Esta vez, o intérprete teve que utilizar, o depósito das cadeias para colocar "PRIMEIRASEGUNDATERCEIRA" , que constitui o novo conteúdo de C$. Por outro lado, o valor 29 assinalado parece exagerado com relação aos 23 bytes que constituem o novo comprimento de C$. Também este mistério tem explicação. A concatenação (soma) de três cadeias não é feita de uma só vez mas em duas fases: na primeira são somadas as duas cadeias A$ e B$ obtendo uma cadeia intermediária "PRIMEIRASEGUNDA"; na segunda fase é concatenada aquela com a C$ para obter seu novo valor. O inconveniente reside no fato de que as cadeias auxiliares, utilizadas nas concatenações intermediárias, não são eliminadas (para evitar complicações e ganhar tempo). Estas cadeias são como defuntos que estorvam, sem nenhum fim, na zona das cadeias, sem estar "apontadas", e além disso, por nenhuma variável alfanumérica. Com nosso programa, de momento, certamente não tem problemas, ao dispor de 38455 bytes ainda livres, mas não se

necessita muito para entender que, prosseguindo tão alegremente, até a vasta memória do C64 fica curta ao encher-se do "lixo" (em inglês "garbage") deixado pelas manipulações de dados alfanuméricos.

Um amontoamento catastrófico dos resultados intermediários de inúmeras operações não é tão raro como pode ser acreditado: pense, por exemplo, na construção de uma cadeia, caracter após caracter, mediante sucessivas instruções GET e compreenderá a seriedade do problema, inclusive em situações à primeira vista simples. Uma cadeia de 250 caracteres, construída segundo o sistema que acabamos de mencionar, criaria 249 cadeias intermediárias, de um comprimento variável entre 1 e 249 bytes. Fazendo contas se trata de uma montueira imensa: (249 x 248)/2, quase igual a 30 kilobytes. Podemos divertir-nos um pouco imaginando uma situação parecida no seguinte programa, que trata de avaliar as efeitos (perversos) do fenômeno em relação com a velocidade:

```
1 CLR
10 V$ = "  ":  J = 0:K = 0:N = 0:TD = 0;DIM A$(100)
20 FOR J = 1 TO 100
30 V$ = "  ":TD = TI
40 FOR K = 0 TO 250
50 V$ = V$ = "A"
60 NEXT K
70 PRINT TI-TD,J
80 A$(J) = V$
90 NEXT J
```

Respiremos profundamente e deixemos para o próximo capítulo este e outros problemas.

CAPÍTULO VII

O que fazer para que o CARACOL CORRA?

Cronômetros rápidos!

Vamos nos lançar sem temor para o primeiro problema: dado que tanto as variáveis simples como as matrizes são memorizadas justo depois de acabar um programa, o que sucederia se, em um dado momento da execução, definíssemos uma nova variável? Está claro (pelo menos se pode intuir): o intérprete BASIC não tem mais remédio que deslocar todas as matrizes (cada uma com sete bytes); esta fatidiosa operação supõe uma perda de tempo que pode ser notável se existem muitas matrizes.

Com o programa seguinte é fácil verificar, medindo experimentalmente a retardação com o cronômetro interno do C64:

```
 1 CLR:REM RESET PONTEIROS
10 TD = 0:REM PREDEFINE VARIAVEL DE TRABALHO
20 DIM AA (1000):REM DEFINIÇÃO DA MATRIZ
30 TD = TI:REM DA A SAIDA AO CRONOMETRO
40 A = 0:B = 0:C = 0:REM DEFINE OUTRAS TRES VARIAVEIS
50 PRINT TI-TD:REM TEMPO TRANSCORRIDO
```

Resultado: 98 x 1/60 segundos (mais de um segundo e meio) para deslocar todos os 3001 elementos de sete bytes para cima, indispensável operação repetida três vezes, tantas quantas variáveis (A, B e C) são introduzidas. Em outras palavras, com a inocente

instrução intermediária da linha 40, foi perdido um tempo precioso sem nenhum sentido prático. Seguramente será instrutivo realizar a medida efetuada no programa anterior para distintos tipos e dimensões de matrizes. Nos daremos assim conta que quando se acusa ao BASIC de ser um caracol, existe bastante razão. Porisso, para que não seja algo temível sua lentidão, devemos tomar medidas para, pelo menos, as situações mais graves de perda improdutiva de tempo.

O caso que estamos examinando nos ensina claramente o oportuno que é definir *ao princípio* em um programa todas as variáveis em jogo, evitando cuidadosamente "golpes de engenho" durante as instruções seguintes. O preço que é pago por não seguir esta regra de ouro é o de temíveis retardações, em uma linguagem já por si só não muito brilhante neste aspecto. Além disso, seguindo a simples regra que acabamos de dar, se matam os pássaros com um tiro, já que é obtida também uma maior clareza na documentação. Por exemplo, em um programa de uma certa complexidade se recomenda que nas linhas inicias esteja contida uma série de "Informações sobre os Dados" que resumam a organização. Podemos nos ajudar eficazmente com a REM, como no exemplo (indicativo) seguinte:

```
10 REM * * * RESUMO DE DADOS * * *
20 REM * * * 1-VALORES CONSTANTES * * *
30 REM VERDADEIRO = -1:FALSO = 0:RAIZ 2 = 1.4142135
.....
60 REM * * * 2-DADOS DE ENTRADA * * *
70 QT = 0:PREC = 0:NOM$ = .....
.....
100 REM * * * 3-AREAS DE TRABALHO * * *
110 IMP = 0:SW = 0:DEP = 0:DEP$ = .....
120 DIM VET (100), TAB (200).....
.....
```

(em nenhum dos exemplos dos capítulos 1 a 5 foi adotado este critério, dada a simplicidade dos casos examinados; isto não é obstáculo para que em alguns destes casos elementares valha a pena introduzí-lo, para evitar os comportamentos lentíssimos). Mas sigamos com a cronometragem neste outro programa:

```
 1 CLR
10 K = 0:TD = TI:REM DA SAIDA AO CRONOMETRO
20 FOR K = 0 TO 3000
```

```
30 NEXT K
40 PRINT TI-TD:REM TEMPO TRANSCORRIDO
```

O tempo de execução resultante é de 243 "jiffys" (este estranho nome corresponde a 1/60 de segundo), isto é, uns 4 segundos. Tente agora substituir na linha 30 NEXT K por um simples NEXT: terá uma redução a 200 jiffys, 43 menos. Esta não desdenhável redução se deve ao fato de que se especificar a variável K do ciclo FOR, o intérprete está obrigado a comprovar, cada vez, sua existência na área específica. Resultado: uma inútil e prejudicial falta de tempo, bastante imprevisível. Portanto convém não especificar no NEXT a variável do loop exceto quando se corra o risco de dar lugar a alguma ambiguidade, como nos loops aninhados. Escrevendo o loop todo inteiro em uma mesma linha o benefício de velocidade obtido é, ao contrário, mais modesto do que se poderia pensar: exatamente 193 jiffys contra os 200 e 243 dos dois casos que vimos.

Provemos agora com o seguinte:

```
1 CLR
10 N = 0:K = 0:TD = TI
20 FOR K = 0 T 3000
30 N = N + 1
40 PRINT TI-TD
```

O tempo desta vez, é de 731/60 de segundo. Se tentamos mudar o tipo de operação teremos:

```
30 N = N - 1      tempo = 737 jiffys
30 N = N * 1      tempo = 681 jiffys
30 N = N / 1      tempo = 688 jiffys
```

Como vemos as diferenças são muito exíguas ainda que, curiosamente, a multiplicação torna a operação mais rápida. Evidentemente, estes são os mistérios das operações com ponto flutuante. É conveniente assim mesmo fazer as mesmas provas com um número distinto à unidade. Substituindo nos casos que acabamos de ver o 1 por 1234, os tempos ficam iguais a:

1368 para a soma;
1372 para a diminuição;
1315 para a multiplicação;
1322 para a divisão.

Isto é, os tempos aumentam consideravelmente, quase são duplicados, quando se sai dos casos particulares como 1 ou 0. Chegados aqui, façamos uma modificação aparentemente inocente:

```
 1 CLR
10 N = 0:V = 1234:TD = TI
20 FOR K = 0 TO 3000
30 N = N + V
40 PRINT TI = TD
```

Os tempos, para as quatro operações aritméticas, ficaram agora pela mesma ordem de antes: 662, 666, 609 e 616 jiffys, isto é, a metade que antes, para fazer o mesmo. A explicação deste estranho fato reside em que o intérprete somente pode trabalhar com variáveis do mesmo formato (ponto flutuante com ponto flutuante, inteiros com inteiros) porque está obrigado a fazer a tradução de um dos dois operandos ao formato do outro, trabalho bastante custoso em termos de tempo (as diferenças podem tornar-se trágicas quando as instruções que concernem a tipos mistos forem encontradas dentro de um loop formado por muitíssimas iterações).

Naturalmente, poderia ser obtido o mesmo resultado substituindo na linha 30 o valor por 1234.0, mas é mais prático e elegante definir desde o princípio o valor V da constante. Fazendo-o assim, somente será perdido o tempo necessário para a atribuição ao princípio e de uma vez por todas. A segunda recomendação, que em essência vai emparelhada à dada anteriormente, poderia ser enunciada assim: é necessário evitar, até onde seja possível, os valores imediatos, substituindo-os pelas variáveis adequadas, inicializadas ao começo do programa de uma vez por todas, para chamá-las quando sejam necessárias.

A estas variáveis que, em essência, correspondem a constantes deverá ser dado, se possível, nomes mnemônicos adequados, tipo R2, PI (por "raiz de 2", número "pi") e similares. Agora complicaremos um pouco mais a vida ao intérprete BASIC, acrescentando um loop em um programa mais longo no qual exista muitas variáveis ativas no momento da execução. Acrescentamos ao programa estas duas linhas:

```
2 A = 33:B$ = "ABCD":C% = 456
3 D = 0:E = 98765:F = 0:G = 0:H = 0:J = 75
```

Voltamos a executar o programa, em aparência inalterado em

sua parte central, da qual é medida a duração. No entanto, estranhamente, o tempo de execução é de 848/60 de segundo, um excesso de mais de um terço. Este paradoxo é explicado segundo o que vimos anteriormente sobre a técnica usada pelo BASIC para memorizar as variáveis, isto é, como as coloca uma após outra seguindo a ordem temporal de definição. Desta vez "N" e "V" serão colocadas, respectivamente, no décimo e décimo primeiro lugar da área destinada às variáveis não indexadas. O intérprete, todas as vezes que tem que aceder a elas, terá que procurar entre os nomes, partindo sempre desde o primeiro, até encontrar o que deseja. A retardação da execução pode chegar a ser dramática em programas de certa complexidade, nos quais seja notável a quantidade de dados em jogo. Aqui, outra recomendação prática: **é aconselhável cuidar a ordem de definição das variáveis de forma que**, até onde seja possível, **sejam colocados nos primeiros postos as de uso mais comum ou as que necessitem de um acesso mais rápido**

E o que ocorre com os comentários?

Algum de vocês poderia chegar a temer, segundo o visto, que os inconvenientes da retardação cercam cada passo que dêem. Mas as surpresas não somente surgem a princípio. Tentem acrescentar a seguite linha:

```
25 REM COMENTARIO INUTIL
```

no interior do loop que estamos examinando: sua duração passará a 961 jiffys. Como é possível que uma instrução que não comporta nenhum tipo de elaboração faça perder tanto tempo (113 jiffys a mais)? É provável que os mais adiantados saibam encontrar uma explicação exata a esta pergunta. Será suficiente que pensem no dito sobre o comportamento do intérprete, poderíamos dizer inclusive sobre sua própria natureza.

Damos um pouco de tempo para vocês pensarem sobre isto, enquanto isso, modificaremos a linha 25, alongando um pouco o texto do comentário (por exemplo: COMENTARIO QUE TALVEZ PODERIA SER ELIMINADO); agora o tempo chegará a 1052 jiffys, o que nos leva à conclusão de que **a retardação originada por um REM é também proporcional ao seu comprimento**

Encontraram a explicação que pedimos? Provavelmente será parecida à que vamos dar: **a natureza seqüencial da interpretação**, isto é, sua execução linha após linha, **supõe também uma no-**

tória perda de tempo no exame das instruções que, como os comentários, não dão lugar a um processo de tradução à linguagem máquina nem muito menos, à execução de um código. De todas as formas, o intérprete não pode evitar fazer tal exame cada vez que, por assim dizer, tropeça novamente com uma REM. A reflexão que acabamos de fazer nos dá pé para dar outro conselho: evite usar comentários no interior dos loops e limite-os em todos os casos ao comprimento mínimo indispensável..

Seguramente algum de vocês dirá que se trata de um princípio arquiconhecido; melhor assim. Somente queremos acrescentar que, quando não se quer renunciar a uma utilização abundante de comentários, o melhor que se pode fazer é ter na memória de massa duas versões de programa: a primeira, sem REM, servirá para a execução, enquanto a outra, rica em comentários inclusive quilométricos, será utilizada como cópia de arquivo para ser utilizada cada vez que se quer trabalhar sobre as distintas instruções para revisá-las ou colocá-las em dia.

Agora, vamos divertir-nos com este programa:

```
 1 CLR
10 K = 0:TD = TI
20 FOR K = 0 TO 3000
30 GOSUB 1000
40 NEXT K
50 PRINT TD-TI
60 END
1000 RETURN
```

Tempo: 504 jiffys. Se substituir o número 1000 da subrotina por 100 é obtida uma pequena vantagem (484 jiffys) devido a que ao converter em binário, de uso interno, um número a outro menor, requer para seu manejo menos tempo. Mas isto não é tudo. Acrescentamos instruções que, à primeira vista, não influenciam na duração do loop que estamos dissecionando, ou seja:

```
1CLR:GOTO 10
2 REM
3 REM
4 REM
5 REM
6 REM
7 REM
8 REM
```

```
 9 REM
70 REM
71 REM
72 REM
73 REM
74 REM
75 REM
76 REM
77 REM
78 REM
79 REM
```

Desta vez a duração passou a 611 jiffys, mais de 25% de aumento. Como é possível? Para entendê-lo melhor, realizemos uma nova modificação:

```
 2 RETURN
30 GOSUB 2
```

obteremos que a duração se reduz a 405 jiffys. O mistério parece fazer-se mais denso, mas logo será esclarecido se pensar em como atua o intérprete na busca de um número de linha. Dado que as instruções estão memorizadas uma após a outra, em ordem ascendente de linha, mas sem limitações sobre o passo da numeração, a única coisa que pode fazer o BASIC é explorar todas, recorrendo-as a partir da primeira, até encontrar a que tenha o número indicado no GOTO ou GOSUB. Isto explica facilmente como, para alcançar a linha 1000, é utilizado mais tempo que para encontrar a linha 2. Conclusão: contrariamente ao que costuma ser feito é preferível, para conseguir uma maior velocidade, colocar as subrotinas à cabeça em lugar de na cauda do programa, como é feito em Pascal.

No caso de programas muito longos a vantagem conseguida com tal proceder chega a ser enorme e capaz de reduzir drasticamente os tempos de resposta. Evidentemente, a consideração anterior vale também, para os GOTO e os THEN, ainda que com eles possa ficar difícil uma intervenção sem arriscar-se a alterar o fluxo lógico correto das instruções. Em qualquer caso, tem que tentar colocar ao princípio as instruções que são executadas mais freqüentemente.

Resumindo, as regras práticas para aumentar a velocidade de um programa BASIC interpretado que vimos até agora são as se-

guintes:

- escrever NEXT em lugar de NEXT e a variável do loop;
- definir todas as variáveis e os vetores nas linhas iniciais, incluídas as constantes, às quais será atribuído um nome simbólico;
- antepor na ordem de definição os dados mais utilizados e/ou aqueles para os quais seja necessário um acesso o mais rápido possível;
- evitar, ou pelo menos limitar, o número e comprimento dos REM;
- situar as subrotinas à cabeça, e não à cauda, do programa.

Seria um útil exercício, neste ponto, repassar os exemplos dos primeiros capítulos, associando-os com estes conselhos práticos.

Novamente as cadeias

Seguramente as cadeias despertarão também o interesse dos leitores, seja pelo curioso de seu comportamento, ou pela importância das retardações que pode provocar, das quais é indispensável conhecer suas causas e possíveis remédios.

O tempo investido na construção de uma cadeia é de 91 jiffys a primeira vez; depois vai crescendo, mas não de um modo uniforme, alcançando o valor de 102 no passo 15 e baixando a 91 no 16, para continuar crescendo continuamente até um valor de 156 na cadeia 43. Provando com DIM A$(500) todos os tempos ficam alongados notavelmente, alcançando no passo 49 um valor de 297/60, mais do triplo do que foi empregado ao princípio. Evidentemente, quando se dá conta o intérprete de que não tem mais espaço à sua disposição se vê obrigado a eliminar as cadeias inúteis, reordenando as ativas para poder continuar a execução de maneira apropriada, o que comporta uma enorme quantidade de trabalho, já que tem que compactar quase 40 kilobytes de memória. Mas, como se explica que com um número de variáveis mais elevado, 500 em lugar de 100, o tempo fique mais longo? O número de cadeias criadas não teria nada que ver com a extensão da matriz. Isto é explicado porque quanto maior o número de variáveis definidas mais custosa resulta a primeira parte (construção) em termos de tempo.

O processo descrito anteriormente é chamado em inglês "garbage collection" (detecção e eliminação da informação inútil) e é uma expressão que explica eloqüentemente o objeto da reordenção da área das cadeias. A operação supõe grandes problemas na

execução de alguns programas complexos, seja porque pode dar lugar a retardação de muitos décimos de segundo, ou porque a ação de limpeza pode iniciar em momentos bastante imprevisíveis que não se tem sorte, poderiam coincidir com os mais críticos e menos indicados. Durante a recolhida da informação inútil o computador aparece bloqueado e não responde a nenhuma ordem, inclusive STOP, para voltar a funcionar ao final da operação como se nada tivesse acontecido; um comportamento enganoso mas plenamente justificado pela exigência de colocar remédio a uma situação insustentável. Portanto, não existe tempo para gestionar outros recursos de E/S.

Talvez o único remédio praticável é o de prever a recolhida de informação inútil, forçando esta operação em um momento escolhido cautelosamente pelo programador. Isto pode ser feito recorrendo à instrução FRE(0). De fato, a pergunta referente à quantidade de memória ainda disponível obriga ao BASIC a ordenar o espaço das cadeias para dar uma resposta inexata sobre as disponibilidades reais de espaço, o qual pode aproveitar-se adequadamente. No caso do programa que estamos examinando podemos acrescentar a linha:

```
22 K = FRE(0)
```

A duração do processo de construção da cadeia permanecerá constante (91) até o ciclo 23, pois a operação partirá sempre com a memória reordenada e com todo o espaço restante à sua disposição. Desgraçadamente, a presença de variáveis ativas reduz sucessivamente esta disponibilidade e, a partir do passo 24, se faz necessário um recolhimento de informação inútil inclusive durante a criação da simples cadeia, o que alonga o tempo de execução. Por outro lado a duração da reordenação cresce proporcionalmente ao número de variáveis que devam ser examinadas e neste ponto o BASIC entra em crise.

Dois acréscimos posteriores permitem analisar com precisão o funcionamento dos tempos de trabalho e de reordenação:

```
21 TD = TI
23 PRINT TI-TD
```

Paradoxalmente é indispensável ter uma grande capacidade de memória para conseguir uma certa velocidade no tratamento das cadeias. Não é tanto uma questão de espaço que pode ser ocupado, como a conseqüência indireta da quantidade de vezes que

tem que realizar-se necessariamente o recolhimento de informação inútil: quanto maior for a capacidade da área das cadeias menos vezes terá que intervir o computador para fazer uma reordenação.

É uma boa regra limitar ao mínimo o número das variáveis, isto é, abreviar a primeira fase de reordenação; lamentavelmente não pode ser feito para superar este grave e insuspeito limite, típico das linguagens interpretadas em geral.

Acrescentando-o a nosso programa as linhas seguintes:

```
51 D = PEEK (51) + PEEK(52)*256 + (PEEK(49) + PEEK(50)*256)
52 PRINT D,K
```

será possível fazer uma idéia sobre ocupação da memória durante a complicada construção de uma longa cadeia.

Nos referimos, por último, a outra conseqüência da "Recolhida de Informação Inútil". A presença no C64 de uma porta RS232 é um instrumento simples e econômico para a comunicação, também a grande distância, entre computadores. Quem o provou terá dado conta que as coisas em BASIC nem sempre andam bem. Ainda que não se tenha problemas para a saída de dados, inclusive em grandes velocidades (2.400 bytes por segundo, é muito mais difícil tentar adquirir dados, especialmente com a instrução GET. De fato, muito rápido, ao recorrer à reordenação de memória (que como vimos, inibem as operações de E/S) não é permitido ao BASIC descarregar o armazenamento intermediário do canal série ao mesmo rítmo com o qual é carregado o Sistema Operacional, porque o computador entrará em crise obstinadamente. Diminuir a velocidade do canal RS 232 somente serve para retardar o bloqueio do sistema, dado que a duração do recolhimento de informação inútil tende a aumentar com o crescimento das variáveis ativas e a tornar-se mais freqüente conforme a memória disponível é reduzida. A única forma de superar as limitações impostas pela linguagem interpretada vem dada pela adoção de rotinas de manejo escritas em linguagem máquina.

Compiladores de BASIC, o ponto final

É este um tema freqüentemente motivo de acesas discussões entre os usuários de pequenos sistemas. Se trata de programas tradutores capazes de "pré-interpretar" o programa fonte traduzindo-o à linguagem evoluída e ganhando assim tempo no momento da

execução do programa objeto (ou seja, o programa fonte traduzido a linguagem máquina).

Existem aqueles que pensam que estes aparelhos transformam um programa fonte em BASIC a um programa em linguagem máquina idêntico (ou quase) àquele que tinha sido escrito por um programador mediano em linguagem Assembler. Contrariamente ao que assegura a publicidade isto está muito longe de ser certo. Os compiladores mais comuns são limitados, quase sempre, a traduzir unicamente as instruções de BASIC aos correspondentes saltos a suas respectivas subrotinas de manejo. Evidentemente, é obtido um certo aumento da velocidade ligado ao fato de que é eliminado o tempo de interpretação, mas por outro lado a maioria destes compiladores (falamos dos que são utilizados para computadores não profissionais) fazem muito pouco mais, limitando-se a simples modificações, nem tanto para otimizar o código mas fazer pequenos retoques como a eliminação de todos os comentários.

Segundo nossa experiência pessoal temos que estar previnidos frente a certos compiladores: as vantagens que podem não ser obtidas (medíocres as vezes) podem não compensar os gastos e desgostos com os quais podemos nos encontrar na compilação. Em qualquer caso, somente um programa BASIC que esteja bem otimizado poderá beneficiar-se das vantagens prometidas pela compilação, e sempre que o tempo de interpretação das instruções tenha chegado a uma porcentagem não desprezível com respeito à da execução.

Concluindo: para dar um bom empurrão ao caracol BASIC é uma boa regra seguir os conselhos práticos que recomendamos sem esquecer naturalmente, o delineamento geral: fazer um programa corretamente é difícil, mas possível.

BIBLIOGRAFIA

1. Inteligência Artificial em BASIC
   James
2. Introdução às estruturas de informação
   Lucena
3. Dicionário de Informática - Inglês - Espanhol - Francês
   Mansa
4. Enciclopédia de linguagem BASIC
   Pereira
5. BASIC prático - Conceitos fundamentais e avançados
   Botelho
6. Inteligência artificial no Spectrum - Faça seu microcomputador pensar
   Brain
7. Aventuras com o Spectrum - Um guia para jogar e fazer programas de aventuras
   Bridge
8. BASIC avançado para o TK-90X
   Carvalho
9. Gráficos no Apple e TK-2000 - Conceitos e programas
   Cavanha
10. Como programar seu PC
    Hartnell

11. 6502 tabela de referência
Hernandez

12. MSX - Guia do usuário
Hoffman

13. Inteligência artificial em BASIC
James

14. 57 rotinas em BASIC para o Spectrum
Johnson

15. Padrões em programação - métodos e procedimentos
Longworth

16. Computação gráfica
Magalhães

17. MSX - Jogos de ação
Monsaut

18. BASIC com estilo
Nagin

19. MSX - Jogos em Assembler
Ravis

20. Explorando o TK-90X
Silveira

21. Your IBM-PC - A guide to the IBM PC (DOS 2.0) and XT
Graham

22. Apple II-c - User's guide
Poole

23. Software engineering
Shooman

24. Apple fun and games - for Apple II, II + , IIc, IIe
Wilcox

25. Learn and use Assembly language on the ZX Spectrum
Woods

NOTAS

Se desejamos que nossos programas em BASIC sejam flexíveis, rápidos e eficientes não basta um mero conhecimento das instruções à nossa disposição e da função de cada uma delas. Devemos ir um pouco mais longe.

Os algorítmos, por exemplo, um ponto chave. Tanto se se reduzem simplesmente a problemas numéricos como se exigem uma linha de reflexão (uma estratégia) devemos ter claro o modo de enfrentarmos a sua programação. Ainda assim é de grande importância conhecer a forma na qual o intérprete BASIC maneja a memória, atribuindo, movendo e apagando variáveis, constantes e matrizes, e a maneira de conseguir que uma linguagem tão lenta não o seja mais, e que, ao contrário, ganhe em rapidez.

Todos estes temas, firmados com grande quantidade de exemplos, serão o núcleo deste livro.

Cz$49,50