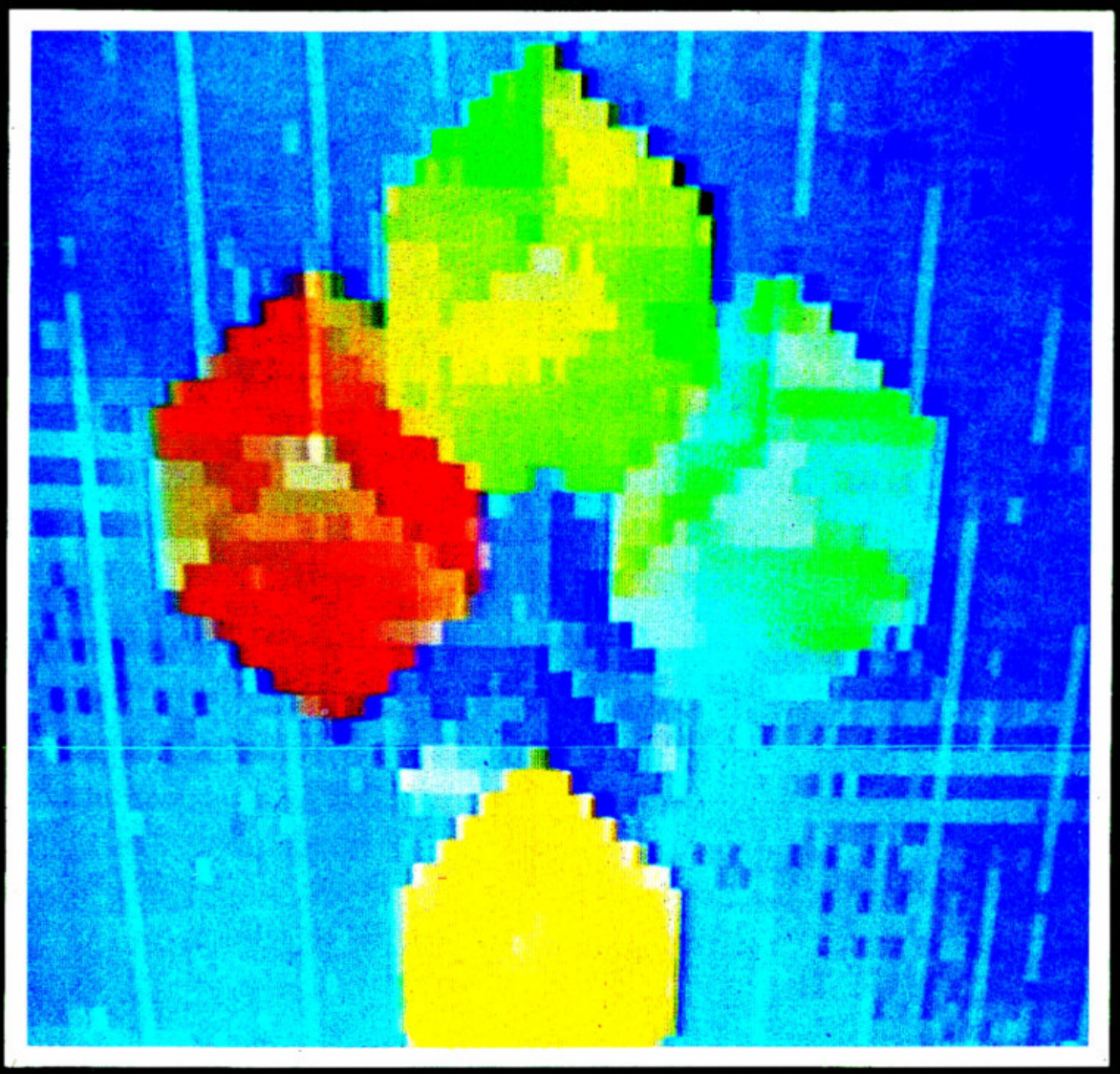
BIBLIOTECA BÁSICA

INFORMÁTICA

INTRODUÇÃO AO PASCAL 8

programação estruturada



SÉCULO FUTURO

BIBLIOTECA BÁSICA

INFORMÁTICA

INTRODUÇÃO AO PASCAL 8

SECULO FUTURO

Diretor editor:
M.A.Nieto
Coordenação e supervisão técnica:
Eng.º Sergio Rocha Paggioli
Tradução:
Ideli Novo
Projeto:
Rainer K.E. Ladewig
Diretor de arte:
Duilio Sarto F.º
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

CAPÍTULO VI

BIBLIOGRAFIA

PREFÁCIO

No mundo dos computadores pessoais as linguagens de programação que disputam o primeiro posto são, praticamente, somente duas: o popularíssimo BASIC e o mais aristocrático Pascal.

Para dizer a verdade, no segmento dos microcomputadores profissionais, aqueles que estão dotados do sistema operacional CP/M (nas máquinas de 8 bits) ou MS-DOS (máquinas de 16 bits) encontramos também outras linguagens, bem conhecidas na informática tradicional, como o FORTRAN ou o COBOL, dedicados o primeiro à programação científica e o segundo à gestão. Inclusive chegou-se a introduzir no PC IBM o já venerável RPG II, linguagem sem dúvida interessante de tipo "não de procedimento", isto é, que o programador define as especificações dos problemas mais que as instruções específicas; pode considerar-se que na lógica de controle fica subentendida pelo menos a grosso modo. Está orientado, por sua própria natureza, ao tratamento de dados por "lotes" (em inglês "batch"), isto é, em grandes massas consecutivas. Pode haver algo mais estranho ao autêntico espírito - interativo, transacional e em tempo real - do computador pessoal? Para quem não conhece estes termos, recordamos que se referem, respectivamente, ao diálogo homem-sistema, ao tratamento esporádico de dados únicos e à imediatividade dos resultados (isto, naturalmente, se é que a lentidão de certos intérpretes BASIC o permite).

Com as ferramentas de programação mencionadas (COBOL,

FORTRAN e RPG) foi realizada a reciclagem do enorme volume de software criado nos anos sessenta e setenta com uma certa facilidade. Este software foi criado para computadores grandes e minicomputadores, especialmente, no campo comercial para contabilidade, faturamento, armazenagem e demais cinzentas mas úteis aplicações. E é uma pena porque este processamento talvez tenha impedido a muitos programadores colocar-se em dia e concentrar seus esforços nas novas aplicações para computadores pessoais: gestão de bases de dados descentralizadas, tabelas eletrônicas ("spreads heet") e, por que não, inteligentes jogos computadorizados.

Não divaguemos mais e voltamos aos computadores pessoais "de verdade". São dois, fundamentalmente, os motivos que fazem do BASIC e do Pascal as linguagens mais adequadas a este novo espírito:

- a interatividade,
- a simplicidade e a rapidez de uso

Para ser exatos, a primeira virtude é mais própria do plebeu BASIC que no nobre Pascal, e está ligada principalmente ao caráter interpretativo do primeiro, ainda que tem que recordar que também existem BASIC compilados. Graça à interpretação, isto é, à tradução direta durante a execução, característica comun também ao LOGO e demais esotéricas linguagens, como o FORTH e o APL, é possível provar rapidamente um programa sem problemas nem complicações. Ao contrário, a Pascal é compilada, o que comporta o uso de quantidades de ferramentas: um editor para redigir os programas, um enlaçador-carregador ("linker-loader") para conjuntá-los e carregá-los na área de trabalho e um compilador para traduzí-los a linguagem máquina.

Em honra à verdade, em certas ímplementações de Pascal que partem do excelente UCSD Pascal, realizado pelo professor Bowles, da Universidade de San Diego, Califórnia (UCSD significa precisamente University of California at San Diego) foi feito um grande trabalho de simplificação para tornar mais fáceis e claras estas aborrecedoras operações. No entanto, fica o fato de que a fases de edição, independente da compilação, continua sendo um mal necessário, com o agravante, além disso, de ter que reeditar o programa desde o princípio cada vez que o compilador dê um erro de sintaxe (também em BASIC "syntax error" é uma mensagem que sai cada vez que é violada alguma regra ortográfica, mas não é necessária nenhuma manobra para mudar de "ambiente", dado que sempre permanece em um mesmo e único entorno).

Além disso, alguns críticos autorizados chegaram a susten-

tar ainda que não compartilhemos plenamente seu extremismo - que"... a reintrodução da compilação no âmbito dos computadores pessoais realizada pelo Pascal ... foi um erro".

Por que falar então de programação estruturada, baseando-se no Pascal, em um livro de divulgação principalmente dirigido ao usuário final? Pois porque o Pascal possui pelo menos uma virtude definitiva que a BASIC nem sequer pode sonhar:

A estruturação

Isto é, um delineamento formal claro e rigoroso inspirado nos mais modernos e sãos princípios de arte da programação.

Os já mencionados críticos autorizados chamam ao BASIC de linguagem "confusão" com os péssimos costumes do antigo FORTRAN (o COBOL, ao contrário, já é mais apresentável, pois tem uma certa pinta de linguagem semiestruturada...). Mas isto é o outro tema, e, se em todo caso, a programação estruturada não tem nenhuma culpa, nem com isto é desprezada sua excelente capacidade pedagógica e formativa.

Efetivamente, o Pascal foi idealizado originariamente com finalidades didáticas. O pai desta linguagem, o suiço Niklaus Wirth, professor do Instituto Politécnico de Zurich (que além disso apresentou recentemente uma nova versão, expressamente modular, denominada Modula 2), sempre declarou que se considera essencialmente um docente. Conseqüentemente, a Pascal, em algumas de suas implementações, não está nem sequer dotada de opções adequadas para a gestão de arquivos sobre memórias de acesso aleatório (disco) enquanto que, no entanto, o "grosseiro" BASIC possui pelo menos aquelas funções necessárias para a vida cotidiana.

Fazer menção deste e outras carências, como, por exemplo, a ausência, ainda que somente no Pascal standard de Wirth, de variáveis tipo cadeia, quase imprescíndíveis na manipulação de dados alfanuméricos, não significa querer dar pé à polêmica, e ainda menos desanimar-lhes da leitura dos capítulos seguintes. Nosso interesse é que muitos usuários convencidos de BASIC, que perseram no vício de acrescentar desaforadamente linhas aos programas, preocupando-se unicamente de que estes corram (o que acaba correndo depois de passar infinitas penalidades, sempre proporcionar-lhes à pressa com a qual é escrita a primeira versão), aproveitam uma boa ocasião de aprender pelo menos regras de bom estilo. E não somente isto.

De fato, os cânones da programação estruturada podem ser resumidos nos três itens seguintes:

- definir uma série de construções standard sobre as quais articular, por blocos, a totalidade do programa;
- subdividir um problema complexo em subproblemas mais simples. (princípio de "divide e vencerás"). E traduzido no delineamento top-down (de cima para baixo) dos programas;
- determinar uma tipologia e umas estruturas de dados adequadas para o problema considerado.

Aprender, e melhor se é com espírito crítico, estes três princípios lhe servirá para qualquer coisa, porque será acostumado a trabalhar de uma forma ordenada e rigorosa que facilita enormemente a compreensão do que se faz. E isto é um aspecto importantíssimo em especial para aqueles que trabalham profissionalmente na manutenção de software.

Pode ser que ouça dizer que todo o exposto não tem validade para o BASIC, porque esta "grosseira" (ainda que fácil e eficaz) linguagem não possui as estruturas IF/THEN/ELSE, DO/WHILE, REPEAT/UNTIL, etc., características; não faça caso. Este tipo de comentários estão completamente fora de lugar, porque reduzem a programação estruturada somente a seu primeiro aspecto, aquele que, em suma, é o mais externo, formalista e, levado ao limite, bizantino. O próprio Wirth considera que a precisão formal é um meio, não um fim. Torna-se extremamente significativo com respeito ao feliz título de um excelente texto de nosso "guru" suiço: "Algorítmos + estruturas de dados = programas".

A subdivisão em módulos e a estruturação dos dados são relevados como o aspecto mais potente e característico do estruturalismo na programação. E, como tentaremos demonstrar em um próximo volume da B.B.I (programando como é devido), estes dois princípios podem, ao menos em parte, ser rivalizados com os simples meios que o BASIC coloca à nossa disposição.

Será mais fácil para aqueles que se enamorarem do Pascal (os usuários desta elegante linguagem são muitíssimos). Aqueles que permanecem fiéis ao BASIC terão muito que aprender, não podendo ignorar que já existem, e espera-se que aumente seu número, dialetos BASIC propostos cada vez de forma mais parecida ao Pascal.

CAPÍTULO I

TOP-DOWN: PROGRAMANDO DE CIMA PARA BAIXO

As origens

Não devemos nos enganar: a programação estruturada desilude às vezes, ainda que talvez somente em uma mínima parte, a quem se "casa" com ela. Porisso não queremos tampouco mistificá-la demasiadamente (apesar do qual temos que confessar-lhe que nos encanta).

Para fazermos uma primeira idéia começaremos por uma de suas mais célebres peculiaridades: a da programação descendente (chamada em inglês "top-down"), que consiste em começar analizando o problema em suas linhas gerais, para depois, pouco a pouco, elucidar os casos particulares. Todos os passos, naturalmente, devem enlaçar-se de uma forma perfeita, com uma série de hábeis e precisos encaixes, como se fosse um "puzzle".

A vantagem deste delíneamento reside, em primeiro lugar, no fato de que é oferecida uma visão de conjunto, panorâmica, mas fácil de dominar. Como dizia um dos máximos representantes do estruturalismo, "o homem tem a cabeça pequena", e portanto, se cansa se tem que dominar assuntos demasiadamente amplos ou complicados. Se pensarmos bem, isto não é uma novidade. Os desenhistas traçam há muitos séculos em folhas separadas as vistas de conjuntos e os desenhos de detalhes. Os "estruturalistas" tiveram que fazer sua guerra particular por ser a Informática, no fundo, uma disciplina jovem. Durante muito tempo (os malintencionados acrescentaram: inclusive agora, no escuro rincão

dos velhos centros de elaboração de dados...) foram desenvolvidos programas quilométricos escritos como um todo, e que tinham ao final um desenvolvido

```
30000 IF RESP$ = "Y" THEN GOTO 120
```

Pior ainda: foram escritos programas nos quais as condições ao final de execução estavam em um imprevisível ponto intermediário. Nestas condições, seguir o decorrer de um longo programa, especialmente se não havia sido escrito por nós, torna-se algo realmente trágico, e comporta um contínuo enrolar e desenrolar a listagem.

Estruturas de controle e estruturas de dados

Outra crítica dos estruturalistas ao modo tradicional de escrever software é referente ao estilo de programação "espagueti" (programas desordenados). Vejamos um simples exemplo em BASIC:

```
300 IF A > B THEN 330
310 C = B-A
320 PRINT A, B, C: GOTO 340
330 C = A-B: GOTO 320
340 < RESTO DO PROGRAMA >
```

Para ser sinceros, acreditamos que "passamos" um pouco com o exemplo. De todas as formas, em linguagens como o Pascal ou inclusive em alguns dialetos de BASIC dotados da construção IF-THEN-ELSE, a questão é reduzida a:

```
300 IF A > B THEN C = A-B ELSE C = B-A
310 PRINT A, B, C
320 <RESTO DO PROGRAMA>
```

Que para quem não o saiba, é lido de uma forma assim contínua e eloqüente: se (IF) A é maior que B, então (THEN) calcula C como diferença entre A e B; se não (ELSE) calcula como diferença entre B e A. Simples, verdade?

Mas continuaremos com o tratamento descendente. A programação estruturada quer constituir um conjunto orgânico de regras para a realização de "bons programas". Este objetivo, indubitavelmente genérico, será melhor definido, especialmente nos capítulos seguintes, quando dermos exemplos concretos desta mo-

derna proposta. Sob uma visão geral, a programação estruturada apresenta três aspectos:

- sintaxe
- semântica
- pragmática

O primeiro está em relação com as linguagens concretas de programação que são inspiradas explicitamente em princípios parecidos: depois do "progenitor" (Algol) fala-se sobretudo de Pascal, Ada, Modula 2 e "C"; existem alguns observados também em COBOL e em alguns BASIC.

Quanto à semântica, ou seja, ao significado das palavras ou ao enquadramento conceitual das "estruturas", temos que destacar desde agora que, para estruturar não se devem tomar somente em consideração, como acontece comum e superficialmente, as estruturas de controle, isto é, de governo do fluxo das instruções, mas também as ESTRUTURAS DE DADOS intimamente ligadas às primeiras.

A pragmática, por último, se refere às conseqüências práticas com pressa por materializar os "princípios sagrados" na programação de todos os dias. Este aspecto "casa", obviamente, com a exposição da sintaxe do Pascal. Em termos gerais, a pragmática tem que ver com a estruturação da "arquitetura" de um programa, centralizada fundamentalmente no desenvolvimento top-down (descendente) dos programas e das estruturas dadas.

Uma grande polêmica: GOTO sim, GOTO não

O fundamento mais célebre e controvertido da programação estruturada é o anti-GOTO, ou mais polidamente, "GOTO-less programming", junto ao "ego-less programming, isto é, a programação clara para todos e bem documentada. O "GOTO-less programming", entendido radicalmente, significa suprimir para sempre a instrução GOTO, de salto incondicional. Alguém dirá: não são os saltos a "graça" da programação? Além disso, como se pode evitar se está praticamente integrada com as operações de decisão (IF) e, portanto, como os saltos condicionais?

Para que nos entendamos melhor, deve-se assinalar que estamos falando de linguagens de programação de alto nível, ou seja, daqueles "orientados ao programa". Na linguagem máquina os saltos incondicionais são inevitáveis (ainda que exista quem está projetando CPUs de arquitetura inspirada no estruturalismo), mas passando a um nível mais elevado, mais próximo à nossa maneira de

raciocinar e de falar, deveríamos poder abolir os mesmos. Com respeito à maneira de fazê-lo vimos um pequeno exemplo no programa anterior, usando a construção ELSE.

Entre os primeiros que arremeteram contra o ultrajado GOTO está Esdger W. Dijkstra, que em 1968, em uma famosa carta enviada à revista da ACM (Associação de Cálculo Automático USA), proclamou a necessidade de eliminá-lo em todas as linguagens de alto nível que pretendessem ser dignas deste nome.

O título da comunicação era como uma chicotada "GOTO Considered harmful", algo assim como "Esse maldito e pernicioso GOTO" (em tradução livre, claro). Nela vertia duros julgamentos contra o antigo FORTRAN (linguagem para uso científico) e o PL/1 (linguagem de uso geral, mais moderna, mas que teve pouco êxito): ao primeiro, o definia como "enfermidade infantil", e ao segundo desejava "uma morte fatal".

Em correspondência com o "GOTO-less", foram desenvolvidas várias linguagens nas quais a instrução GOTO não existe (por exemplo, o BLIS ou, para citar outros conhecidos também nos computadores pessoais, o LOGO e o FORTH). Existem alguns no entanto, que ainda que sejam estruturados o toleram: por exemplo, a linguagem C aceita frases com saltos incondicionais, e o próprio Pascal inclui a instrução GOTO, apesar de que desaconselha decididamente seu uso. Efetivamente, além de impor que a etiqueta de destino do salto seja declarada explicitamente (se tem o equivalente de um GOTO 150 do Basic, mas com o inconveniente de ter que definir previamente 150 como etiqueta "label") é habitual que nos manuais de Pascal a instrução GOTO seja tratada em último lugar, como dizendo: "se verdadeiramente não sois capazes de prescindir dela, aqui a tens, para casos extremos".

Esta proibição do GOTO parece a muita gente uma mania. De todo modo, como comentamos antes, não esgota os cânones da programação estruturada. Estes, repetimos novamente, são três:

- desenvolvimento descendente (top-down).
- modularidade, ou seja, subdivisão em pequenas partes,
- estruturação dos dados,, além dos programas.

Dijkstra, em "Structured programming" (texto de obrigatória referência, escrito junto com outros "magos" das informática: Dahl e Hoare), faz "pé firme" em que para dominar um tema tão complexo a única esperança consiste na estratégia de divide e domina ("divide and conquer"), que dizem os ingleses). Tudo se re-

mos descrever imediatamente, não sem antes repetir que, depois de tudo, devem integrar-se intimamente com estruturas paralelas de dados.

Um teorema elegante

Três estruturas (ou construções) elementares são os autênticos modelos de pensamento, base de uma programação ordenada:

- seqüência
- seleção, à qual corresponde a construção IF-THEN-ELSE;
- iteração, associada à construção DO-WHILE.

Nenhuma delas requer uma instrução GOTO ou similar. O suporte teórico deste conjunto é o famoso teorema de Jacopini-Böhm, que demonstra que cada programa, por complexo que seja, pode ser suprimido nas três estruturas mencionadas (Fig. 1).

Para explicá-las consideraremos que são conhecidos os diagramas de fluxo (flow-chart), isto é, a representação gráfica - com retângulos, losangos e outros símbolos análogos - de um progra

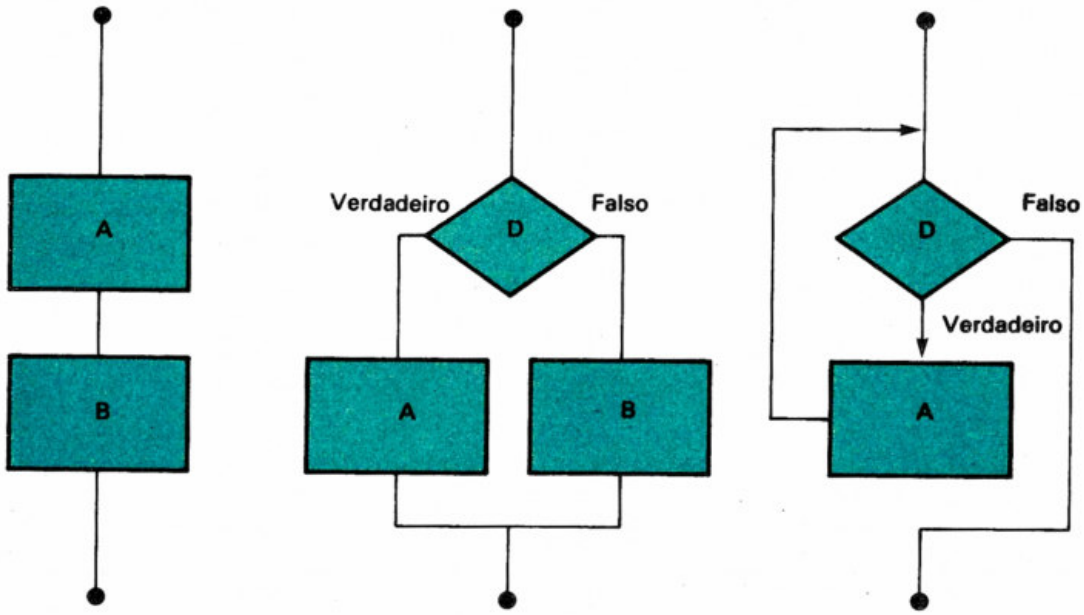


Figura 1 — Uso das três estruturas de controle básicas segundo o Teorema de Jacopini-Böhm (seqüência, seleção, interação).

ma. De todo modo, para benefício dos leigos e dos que não queiram repassar os volumes anteriores da coleção, recordaremos que: um bloco retangular contém uma ou mais instruções, ou então expressa um subprograma, procedimento ou subrotina (neste caso o retângulo costuma ter dois riscos verticais nas laterais), cujo nome será situado em seu interior; um losango representa um bloco de decisão; contém em seu interior uma pergunta ou o teste de determinada condição (comparação ou similar) e tem duas saídas, correspondentes a primeira a "verdadeiro" (ou "SIM") e, a segunda, a "falso" (ou "NÃO").

Um bloco qualquer pode, por sua vez, resumir (é a idéia do top-down) um conjunto de blocos mais ou menos intrincados nos quais está subdividido. Na Figura 1 denominamos A e B às sentenças dos retângulos, e D à decisão genérica ou pergunta dos losangos.

Vamos confiar em Jacopini & Cia e examinar rapidamente as três estruturas propostas por eles, uma após a outra. A primeira é banal; corresponde simplesmente ao fato de que os programas derivam da seqüência de instruções. A estrutura de seleção IF-THEN-ELSE já foi examinada na prática; unida ao desenvolvimento top-down, permite uma visão panorâmica notável. Por exemplo, em um BASIC estruturado será escrito:

```
300 IF < cond > THEN GOSUB 1000 ELSE GOSUB 1500
```

onde < cond > representa uma condição genérica a verificar.

Vamos supor que a subrotina 1000 calcula os descontos para os grandes clientes e a 1500 para os comuns; fica então claro o que o programa, em grandes linhas, realiza. Antecipamos que o Pascal, linguagem de mais alto nível que o BASIC, alinha tudo com uma semântica mais eficaz dos nomes de procedimento. Teremos assim algo parecido:

```
IF < COND > THEN DESCBRUT ELSE DESCORD
```

onde "descbrut" e "descord" são, certamente, mais eloqüentes que os ininteligíveis GOSUB do BASIC.

Finalmente, a terceira estrutura DO-WHILE. Tem que dizer que a iteração não fica ignorada de todo pelo BASIC, pois dispõe do loop FOR-NEXT (herdado do DO do FORTRAN). Ainda que cômodo, não é tão geral como o DO-WHILE, que permite expressar praticamente qualquer tipo de iteração. No DO-WHILE devemos expressar uma condição D: se D não é cumprido saimos do ciclo;

em caso contrário continuamos nele. Em resumo: o ciclo se repete (DO) "enquanto" (WHILE) é verificada a condição.

Em qualquer delineamento "GOTO-less" o programa inteiro está formado por blocos consecutivos que apresentam um só ponto de entrada e um só ponto de saída, o qual permite juntá-los facilmente, como se fossem vagões de um trem.

As maiores vantagens da subdivisão em blocos é obtida quando é possível sua reutilização em outros programas: são convertidos nas chamadas funções ou procedimentos de livraria, que permitem uma grande economia de tempo.

A tendência é, pois, criar um programa principal ("main program") dentro do qual os distintos procedimentos (como são chamados em Pascal, ou subrotinas como são chamados em BASIC) são "chamados" sucessivamente. Ao estar os procedimentos escritos à parte, quando depuramos o programa é possível que somente tenhamos que modificar algum procedimento defeituoso sem ter que tocar nos demais nem no principal, ou então, mais raramente, pode ser revisado o fluxo do principal sem tocar nos procedimentos, especialmente se estes se referem à elaboração das subrotinas (Fig. 2). Este é um dos fundamentos da chamada engenharia do software (software engineering), que se ocupa da cons-

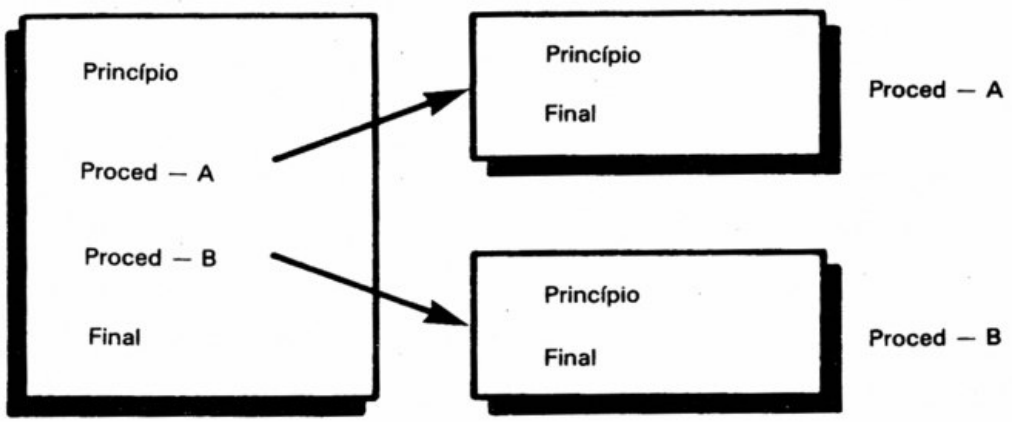


Figura 2 — Outro ponto essencial da programação estruturada é a subdivisão em módulos, sob a forma de "procedimentos", reclamados desde o programa principal. O método certamente não é exclusivo das linguagens estruturadas, mas estes fornecem "ferramentas" mais potentes.

trução correta e eficiente dos programas, além de buscar sua manutenção de forma mais eficaz possível. Se pensarmos detidamente, este princípio também é levado à prática pelos programadores de FORTRAN e BASIC graças a uma cautelosa dosificação de subrotinas. Então...? O que o Pascal (e outras linguagens) oferece a mais são algumas ferramentas de trabalho (tools) mais definidas e potentes.

Até aqui a propaganda. Mas, é realmente tudo tão bonito como parece?

Para contestar a esta espinhosa pergunta temos que fazer notar antes de tudo que, depois dos furores iniciais, o extremismo estruturalista cedeu um pouco, seja pela prática dos usuários ou pelas teorias de outros estudiosos menos extremistas. Entre estes últimos devemos citar as autorizadas opiniões de David Knuth (escreveu uma espécie de enciclopédia de algorítmos, ainda inacabada, chamada "The art of programming"). Knuth advertiu o perigo de chegar a escrever programas obscuros e revezados, ainda que respeitando de cheio as regras básicas da programação estruturada. Responsável destes resultados não demasiadamente otimistas foi sobretudo uma interpretação extremista do Teorema de Jacopini-Böhm. Efetivamente, se além de suprimir o GOTO nos limitamos dogmaticamente às três estruturas da Figura 1, podem surgir facilmente inconvenientes como os evidenciados na Figura 3, que, na prática, obrigam a introduzir códigos duplicados e/ou desviadores (switchs em inglês). Em ambos os exemplos entendemos a equivalência como que, entre os pontos "x" e "y", as situações são idênticas no esquema original e no estruturado correspondente.

No caso a), para obter um diagrama de fluxo equivalente, composto pelas três únicas estruturas básicas, se manifesta em seguida a necessidade de duplicar o bloco A; de tal modo nos vemos reconduzidos à seqüência de A seguida de uma estrutura DO-WHILE, com os blocos B e A.

O segundo exemplo, mais complicado, consiste em um loop com possível saída antecipada ao verificar-se duas condições ("p" e "q"). A redução a módulos iterativos e de seleção (IF-THEN-ELSE) é resolvido desta vez recorrendo ao desviador (ou "comutador") SW; este é uma variável booleana, que somente pode valer, portanto, "true" ou "false" (outras nomenclaturas equivalentes são: "1", "0"; "ON", "OF"; "ativado", "desconectado"; "ligado", "desligado"). O desviador não é utilizado para nenhum cálculo; serve unicamente para desviar o fluxo do programa quando ocorrem determinadas condições. Nos diagramas de fluxo a situação on/off dos des-

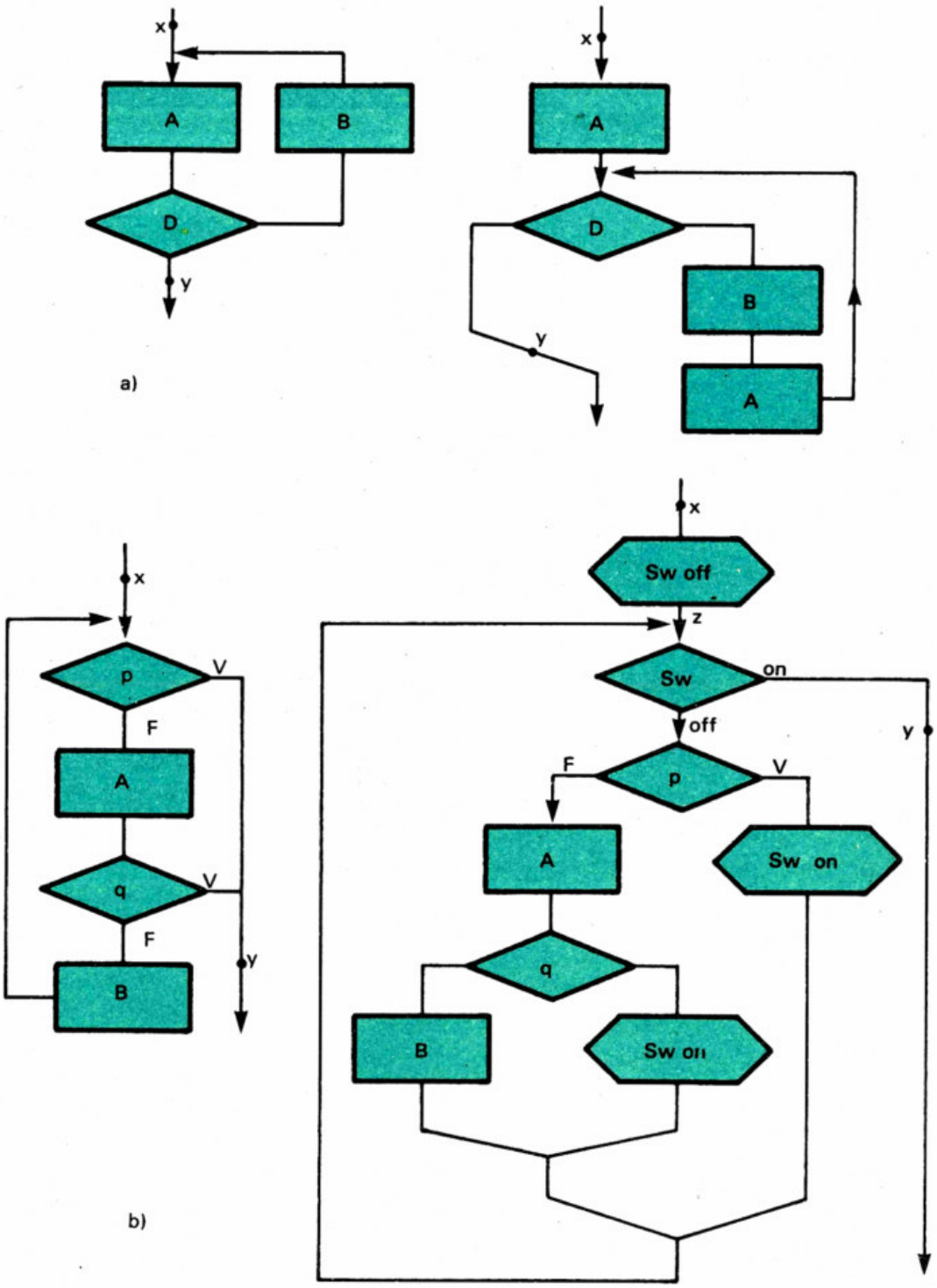


Figura 3 — O uso exclusivo das três estruturas mencionadas no teorema Jacopini-Bhöm pode comportar redundâncias: em a) foi necessário duplicar um bloco. em b) teve que recorrer a desviadores (switchs).

viadores geralmente está encerrada dentro dos hexágonos irregulares. Como se vê na Figura 3, SW está inicialmente desconectado, para ativar-se depois ao produzir-se uma das condições ("p", "q"). Desta maneira, ao início do novo ciclo, no ponto "z", SW conserva rastros do ocorrido em "p" ou em "q", interrompendo-se a iteração (ponto "y"), no caso de que alguma foi cumprida. Para compreender a função de um desviador se pode imaginar um trem que, ao chegar a uma troca de linha pode virar à direita ou esquerda segundo um evento precedente (por exemplo, que houvesse passado outro trem).

Voltando à Figura 3 temos que reconhecer que conseguiu-se decompor tudo em seqüências encerradas em uma única estrutura DO-WHILE (inicia com o losango SW acima), mas a um preço que supõe quase contradizer um dos pressupostos do estruturalismo: a clareza (e poderíamos propor exemplos nos quais a situação seria trágica). A culpa destas situações não é, evidentemente, de Jacopini, nem muito menos do agradável Knuth; eles limitaram-se a demonstrar e comprovar os efeitos de um simples e honrado Teorema.

Agora então, é certo que não devemos limitar a estas três únicas estruturas? Certamente, não. A programação estruturada nos permite dispor de estruturas de controle de fluxo que, ao menos sobre o papel, formam um repertório para quase todas as situações. A Figura 4 nos mostra as mais típicas e difundidas. A sintaxe utilizada é genérica e não tem uma correspondência exata com nenhuma linguagem em concreto.

No DO-WHILE (faz enquanto...) é abandonado o loop quando a condição é deixada de cumprir (é falsa), no entanto no DO-UNTIL (faz até que...) ocorre ao contrário, saindo da iteração ao ser cumprida a condição.

As construções REPEAT WHILE e REPEAT UNTIL funcionam como as duas anteriores, mas a verificação da condição é feita ao final do loop, o qual indica que, com o mínimo, este será recorrido uma vez

A passagem de uma construção WHILE a uma UNTIL, ou viceversa, é imediato, pois basta inverter a condição (A $<$ B em lugar de A $>$ B, por exemplo) ou usar o NOT. O Pascal normalmente é limitado às estruturas DO-WHILE e REPEAT UNTIL.

A construção 5) é referente às saídas intermediárias, no entanto 6) é a clássica DO do FORTRAN e a conhecidíssima FOR do BASIC, pelo que não merece mais comentários; somente observar que no equivalente em Pascal standard a variação do índice somente pode ser em uma unidade e não é permitido modificar seu

```
1) DO WHILE <condição>
     .
     .
     .
   REPEAT

2) DO UNTIL <condição>
     .
     .
     .
   REPEAT

3) DO
     .
     .
     .
   REPEAT UNTIL <condição>

4) DO
     .
     .
     .
   REPEAT UNTIL <condição>

5) DO
     .
     .
     .
   EXITIF <condição>
   REPEAT

6) DO VARYING ... FROM ... TO ... BY ...
     .
     .
     .
REPEAT
```

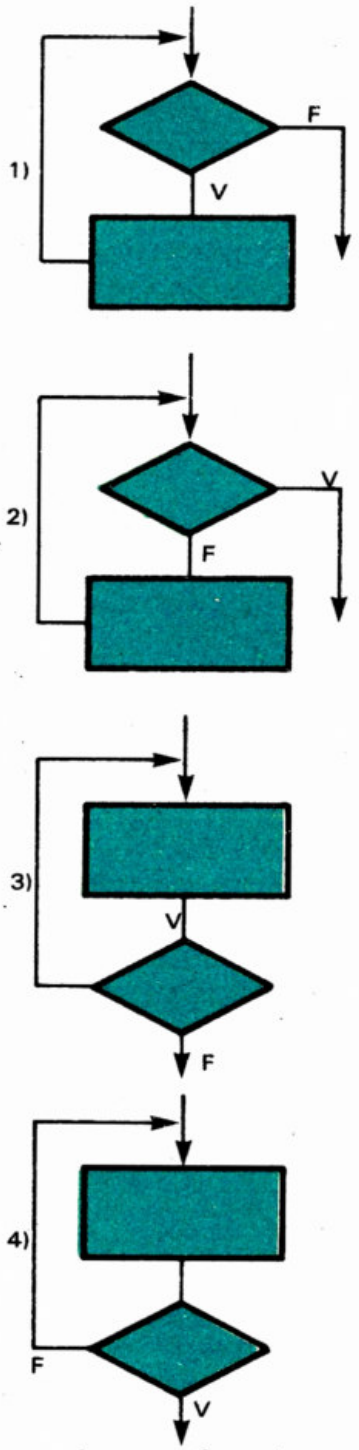


Figura 4 — Com mais variedade de estruturas poderemos fazer frente aos problemas surgidos com a limitação às três essenciais. Nos primeiros pares a diferença está na condição de saída (V ou F). O Pascal somente adota as construções DO-WHILE e REPEAT UNTIL. É útil a possibilidade de saída antecipada (EXITIF. chamada às vezes "break").

valor no loop.

Finalmente, duas palavras sobre a penúltima estrutura, ausente na versão standard do Pascal, mas introduzida em algumas versões comerciais e aceita também pelo mesmo Wirth em sua nova linguagem (Modula 2). A cláusula EXITIF (na linguagem C é chamada "break", denotando uma ruptura antecipada do ciclo ao cumprir-se alguma condição intermediária) corresponde a uma situação que, em BASIC, seria expressa como segue:

```
IF (condic. - 1) THEN
                 BEGIN
                   instrução - 1
                   DO VARYING I FROM 1 TO 60
                        IF (condic. - 2) THEN instrução - 2
                                         ELSE instrução - 3
                     ENDIF
                   REPEAT
                 END

ENDIF
```

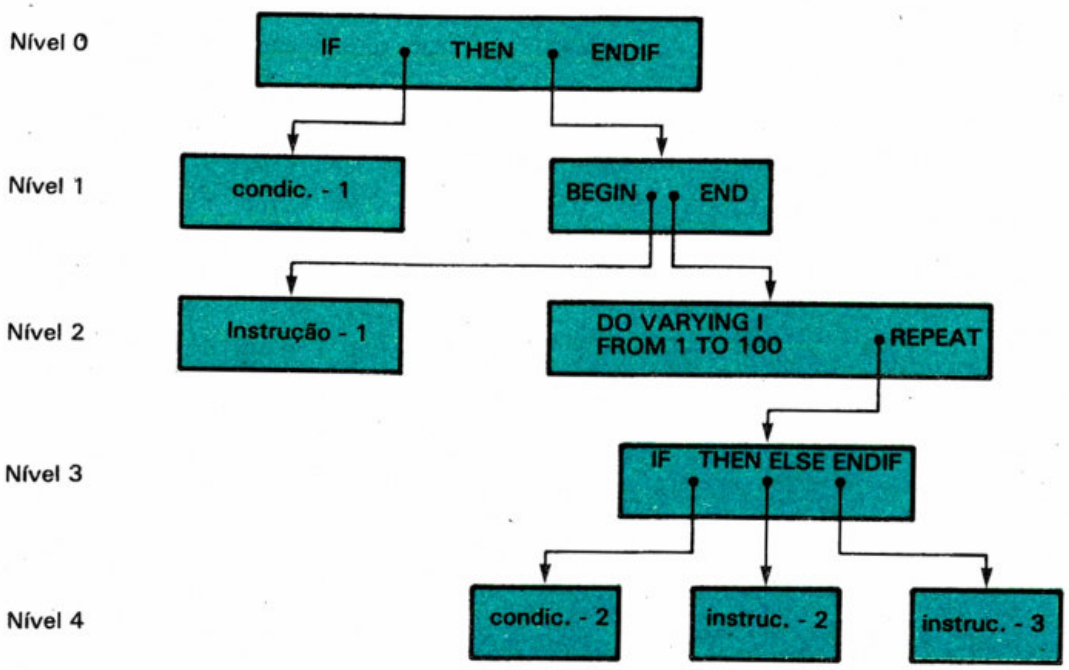


Figura 5 — O aninhamento de estruturas com distintos níveis hierárquicos é típica das linguagens estruturadas. Costuma ser representada mediante "escalonamentos" usando margens diversas.

```
3150 IF <CONDIÇÃO> THEN RETURN
     * * * * * * * * *
3200 RETURN: REM FINAL SUBROTINA
```

Uma estrutura como esta (tolerada pelos puristas apesar de que o EXITIF equivale a um GOTO graças a que salta ao final do procedimento, salvando assim o princípio de que somente exista um único ponto de saída) resolve problemas como os do exemplo b) da Figura 3, permitindo a interrupção de um ciclo ao tornar-se verdadeiro, algum teste interno a ele; desta maneira é subdividido, de fato, em distintas ramificações.

Para completar o quadro, devemos mencionar a estrutura CASE. Atua aproximadamente como a ON < condição > GOSUB xx, yy, zz do BASIC. CASE realiza um teste com soluçoes múltiplas (superior a dois), para cada uma das quais tem definidas as ações que tem que cumprir e/ou os procedimentos aos quais deve recorrer. Em essência é uma ampliação da IF-THEN-ELSE, com distintos aninhamentos a níveis hierárquicos inferiores.

Em termos gerais seria expressa como: IF < condição > THEN < expressão > ELSE IF... Isto coencide com a idéia do desenvolvimento top-down, que, além do aspecto mostrado na Figura 2, poderia ser apresentado como se observa na Figura 5. Este simples exemplo deve esclarecer a idéia de aninhamento de pequenos blocos uns dentro dos outros. Nele foram representados os diferentes níveis de aninhamento graficamente; no mais baixo encontramos o esqueleto da construção IF/ENDIF; no nível 1 encontra-se a condição-1 (que em certos casos poderia implicar operações de mais baixo nível) colocada na "caixa" BEGIN... END, que, por sua vez, encerra a instrução-1 e uma construção DO-VARYING, que, por sua vez...

No decorrer deste livro nos acostumaremos a este mecanismo e ao particular "escalonamento" com o qual se trata de evidenciar, mediante diferentes margens do texto, os distintos níveis (Fig. 5)

Será tedioso ou divertido? Depende dos gostos. Útil, desde já.

CAPÍTULO II

MAIS SOBRE O TOP-DOWN: ESTRUTURAS DE DADOS

Ser abstratos para tornar-se concretos

Freqüentemente nos esquecemos que a elaboração de dados não é limitada somente ao cálculo numérico, ainda que os computadores tenham nascido com este fim (é tanto assim que muita gente continua chamando-os de calculadores). O cálculo matemático se inicia seriamente com o filósofo e matemático Blaise Pascal (não é uma coincidência: Writh chamou Pascal à sua linguagem em hora a Blaise); inventou a "Pascaline", uma máquina de somar mecânica a base de rodas, precisamente porque apiedou-se do trabalho contábil tão desumano que seu pai, arrecadador real, realizava. A história dos computadores se refere muitas vezes ao "number crunching", isto é, a necessidade voraz e incessante de números: na preparação de tabelas de tiro (ciência balística), nos complicados cálculos de projetos científicos (como a caça das particulas subatômicas do prêmio Nobel Rubbia), técnicos (CAD/CAM) ou, mais banalmente, nos cálculos contábeis.

Talvez foram os trabalhos de gestão os que abriram caminho à idéia de que o computador podia fazer algo mais que uma série de operações aritméticas: podia manipular habialmente muitos outros Tipos de Dados. Quando se tem que trabalhar com listagens (clientes, fornecedores, etc.) tem que arquivar, ler e manejar quantidades enormes de dados, numéricos e alfanuméricos. Aqui foi iniciada a elaboração não-numérica, que posteriormente deu ori-

gem à mesma denominação de Ciência da Informação. Como ocorre sempre, tanto na história do homem como da ciência, primeiro são encotradas soluções parciais, e em seguida, com os primeiros problemas, surgem momentos de crises que dão lugar a uma mais profunda reflexão teórica.

Este processo é complicado pela caótica evolução dos suportes informáticos, favorecida pela ausência de normas. Enumeramos em seguida alguns suportes que foram sucedendo no campo da informática: cartões perfurados, fitas perfuradas, fitas magnéticas, cartões magnéticos, tambores magnéticos, discos magnéticos rígidos, flexíveis (disquetes ou floppy disk). Nós paramos por aqui, ainda que tendo em conta que por um lado, existem outros a ponto de aparecer (discos ópticos) e que, por outro, cada um dos suportes mencionados não somente são baseado sem tecnologias diferentes, como também requerem distintos "formatos" e codificações, e, pior ainda, uma organização e estrutra dos dados gravados incompativelmente, quase sempre. Para colocar um exemplo clássico, no cartão Hollerit, adotado pela IBM, seus 80 caracteres, tantos como colunas furadas constituiam o registro (ou record): uma verdadeira camisa de força. Além disso, o cartão de 80 caracteres era "o registro" por antonomásia. Outro limite vinha dado neste como em outros sistemas de armazenamentos externos (na gíria informática fala-se de "memórias de massa", para evidenciar sua grande capacidade em relação à da memória interna) pelo fato de que o acesso era seqüencial, isto é, para alcançar um dado (ou gravação) intermediário tem que ler antes por força, todos os anteriores ainda que não nos interessem. Afortunadamente, hoje existem memórias de acesso direto ou aleatório (random, em inglês) nas quais o dado procurado pode ser obtido de forma direta, como ocorre nas memórias RAM (siglas que querem dizer, Random Access Memory, memória de acesso aleatório).

Todos estes problemas iniciais fizeram sentir uma necessidade de definir standards também nos dados, isto é, fixar tipos válidos universalmente. A coisa surgiu muito espontaneamente com o desenvolvimento das linguagens de programação "problem-oriented", dirigidos aos problemas das pessoas e não às exigências e particularidades da máquina. Enquanto nas linguagens "machine-oriented" fala-se de bits, bytes, campos de cartões, etc., nos "problem-oriented" referem-se aos tipos "lógicos para distinguí-los dos físicos; assim existem, por exemplo, o registro lógico CLIENTE por um lado, e por outro, a posição física na qual são gravados os dados. A gravação física pode diferir do registro lógico em sua organização: por exemplo, em uma fita magnética

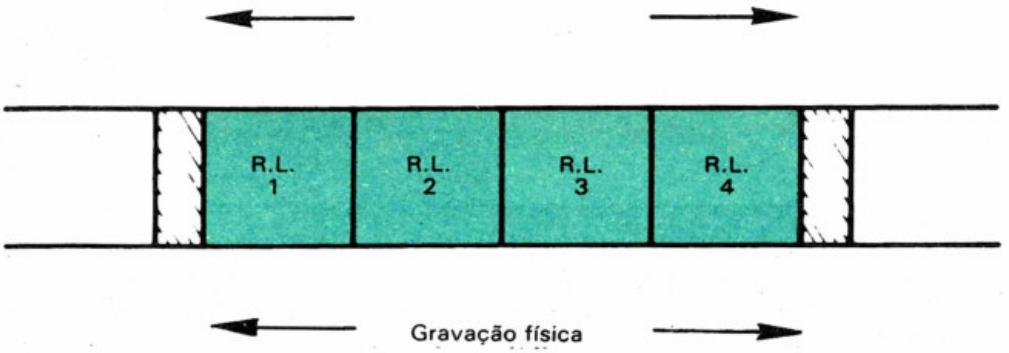


Figura 1 — O registro lógico pode não coincidir com o de gravação física. Por exemplo, em uma fita ou disco magnético podem estar contidos em um mesmo bloco ou setor vários registros lógicos colocados um após o outro.

pode ocorrer que a gravação física contenha em uma mesma zona os dados de vários clientes, por motivos de economia de espaço; também podem ser encontradas codificações que estabelecem a desagregação dos dados em diferentes localizações físicas (Fig. 1). O usuário, certamente, não quer saber nada destes assuntos, nem muito menos enlouquecer passando de um sistema de gravação física a outro. Muito rápida foi produzida a evolução dos dados; em uma linguagem de alto nível não é operacional estar obrigado a trabalhar somente com bits e bytes. Assim, o antigo FORTRAN, desde suas primeiras versões oferecia variáveis inteiras, reais, em simples ou dupla precisão, alfabéticas, cadeias, e outras mais complicadas, como tabelas, vetores e matrizes.

Também neste caso é reafirmado o princípio de independência da máquina, conduz à definição lógica ou, como hoje se diz, *abstrata* dos dados. Assim, pois, haveremos de ser abstratos nos dados para que os resultados possam ser concretos.

Tipos de dados

A programação estruturada - já dissemos mais de uma vez, mas insistiremos ainda à custa de ganhar o Oscar do aborrecimento - significa, entre outras coisas, estruturação dos dados. Como foi comentado no item precedente,trata-se, em primeiro lugar, de uma operação de abstração, sentido de que prescinde ao máximo das modalidades de implementação de um certo conjunto de dados. Em outros termos, com os *Dados Abstratos*, a ênfanse é colocada em sua significação, em seu aspecto lógico. Isto é feito assim para assegurar a máxima transportabilidade do software de um

sistema ao outro. O que se entende por "transportabilidade"? Explicaremos em duas palavras: um software se diz transportável quando, tendo sido escrito para um determinado computador, funciona perfeitamente também em outros.

Este ideal, nascido com as primeiras linguagens de alto nível, culmina hoje com a linguagem Ada (potentíssima e superestruturada, ainda que não adequada para computadores pessoais), que, desenvolvido para o potente Departamento de Defesa USA, procura ser um verdadeiro esperanto informático e permitir que o D.D USA possa fazer funcionar um determinado programa em uma máquina de qualquer uma das inumeráveis e variadas marcas das que está dotado o Ministério de Defesa, com todas as suas divisões e departamentos.

Antes de continuar e de ... concretizar o discurso sobre a abstratação (finalmente!) tem que ser apontado, pelo menos, o fato de que esta tipologia abstrata ideal está contudo longe de ser perfeita. Em nosso tratamento de Pascal, que tem fins exclusivamente didáticos e formativos, nos ocuparemos quase exclusivamente dos tipos de dados que residem, também fisicamente, na memória interna.

O conceito de tipo abstrato de dados significa duas coisas:

- um conjunto de valores ou campo de existência de algumas características dadas,
- um conjunto de operações que podem ser feitas sobre esses valores.

Um exemplo banal seria o tipo "inteiro"(INTEGER em Pascal) que obviamente se refere aos números inteiros algébricos, e cujas operações são aritméticas; mas podemos acrescentar outros inventados por nós. Vejamos um par deles, prescidindo em ambos os casos de qualquer linguagem de programação.

Primeiro exemplo: CASAS COLORIDAS.

Definição: o conjunto de casas que tem que pintar.

Operações: pintar de branco, pintar de vermelho, pintar de azul... (muito freqüentemente trata-se de enumerações exaustivas: serão evitadas, espero, casas pintadas de horrendas cores, como roxo e preto!).

Segundo exemplo: CARTAS DE BISCA

Definição: conjunto de cartas dos quatro naipes (copas, espadas, ouros e paus) do ás ao rei.

Operações: baralhar, distribuir aos jogadores...

Com o segundo exemplo já se compreende em seguida uma particularidade: um tipo, por exemplo, CARTAS DE BISCA, pode subdividir-se em subtipos (os naipes). Para voltar a assuntos que

tem que ver mais diretamente com a informática podemos recordar que um subtipo elementar do tipo "integer" é o CARDINAL, constituido somente pelos inteiros positivos (distinção que alguém poderia achar um pouco prolixo, mas que às vezes é útil). Imaginemos agora que definimos um novo tipo (existe realmente e é muito importante) chamando-o PILHA (no sentido de montão, stack em inglês). Uma pilha estará formada por dados empilhados uns sobre outros, segundo a ordem de chegada e que podem ser extraídos um a um da parte superior (top do stack), evidentemente, em ordem inversa com relação à da acumulação (se tentássemos tirar outro situado debaixo, os que estão por cima "cairiam"). Porisso o stack é chamado também memória LIFO (Last In First Out, último a entrar, primeiro a sair), tradução livre do evangelho "os últimos serão os primeiros" (Fig.2).

As operações do tipo PILHA que definiremos são, principalmente:

- PUSH (empurra). Consiste em acumular um dado novo na parte superior.
- POP (tira). Operação inversa. Recupera um dado de cima, com a conseqüente redução dos elementos da pilha.
- TOP. Copia o dado de cima, mas sem modificar o conteúdo da pilha.

Como é obvio, uma boa linguagem de programação deve prover ao usuário uma quantidade adequada de tipos pré-definidos.

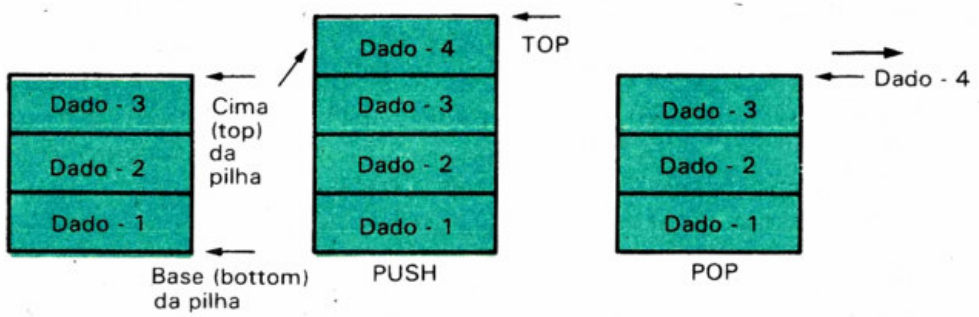


Figura 2 — Esquema geral de uma pilha (stack). A operação PUSH carrega um dado novo em cima da pilha enquanto que POP extrai o dado situado "acima" modificando a pilha. Pode ser feita também uma operação TOP que colhe este dado e o copia sem modificar a pilha.

A potência de uma linguagem é medida pela quantidade (e qualidade) dos tipos de dados que pode manejar. O pobre BASIC o é de verdade, sob este ponto de vista, dado que geralmente oferece somente os tipos inteiro, real (em simples e às vezes em dupla precisão), cadeia e matriz.

O aspecto de maior relevância da programação estruturada, presente em todas as linguagens que se inspiram nela, reside em que tem a possibilidade de criar novos tipos, definidos pelo usuário em base suas necessidades particulares. Os novos tipos são construídos a partir dos pré-definidos, oferecidos pela linguagem. Uma vez mais a idéia é a de realizar uma construção com as peças do Leigo, ou se preferir, as de um "puzzle".

Deste modo podem ser projetados os tipos mais adequados a cada problema determinando ao mesmo tempo uma limpa e clara separação entre tipos heterogêneos. Dito familiarmente: não se devem confundir maçãs com peras, tal e como nos ensinavam na escola. As maiores causas à criação destes conceitos vieram do habitual trio de gurus (Hoare, Dahl e Dijkstra). O grande mérito de Niklaus Wirth foi implantá-los em suas linguagens (Pascal e Modula 2). Coloquemos agora um pouco de ordem, subdividindo os tipos de dados em duas categorias:

- simples,
- compostas.

Tipos simples

Poderiam chamar-se também atômicos, não por referência à energia atômica, mas aludindo ao fato de que se trata de valores elementares, sem possíveis subdivisões ulteriores. Os compostos, ao contrário, são tipos construídos a partir dos simples, por adição.

Os simples não são limitados aos definidos "a priori" como parte da linguagem (tipicamente, os conjuntos de números inteiros, reais, caracteres alfanuméricos e valores booleanos). O Pascal e outras linguagens estruturadas permitem definir novos tipos simples, geralmente mediante duas técnicas: enumeração e intervalo (range).

O exemplo que segue esclarece estes conceitos: está escrito seguindo uma sintaxe que se assemelha muito (e freqüentemente coincide) à da Pascal, sem limitar-se com exatidão às regras. Em suma, uma antecipação que, já colocados, nos familiariza com a Pascal. As palvras reservadas, ou seja, típicas da linguagem estão

escritas todas com maiúsculas.

```
TYPE diasem = (segunda-feira, terça-feira, quarta-feira,
              quinta-feira, sexta-feira, sábado, domingo);
     diatrab = segunda...sexta-feira

VAR x:diasem;y:diatrab;
```

Trata-se da parte de seção de um programa na qual são definidos os tipos e as variáveis. Os que conhecem somente o BASIC notarão em seguida que a Pascal e suas afins, são mais sofisticadas. Por desgraça, é verdade. Como compensação, são extremamente mais claras de ler (clareza e síntese juntas? impossível tê-las em uma mesma linguagem). As linhas que acabamos de ver nos dizem que o programador quiz definir um TYPE (tipo) diasem, enumerando as sete expressões alfabéticas entre parêntesis. Depois de ter utilizado a técnica da enumeração recorreu ao da classe; criando o tipo diatrab, constituído pelos dias que vão desde a segunda até a sexta-feira (os quatro pontinhos na sintaxe da linguagem estabelecem todo o intermediário entre os dois valores assinalados). Posteriormente são definidas as variáveis "x" e "y" com a palavra reservada VAR como pertencentes, respectivamente, aos novos tipos de diasem e diatrab; a associação é estabelecida, como podem ver, mediante os dois pontos (:).

Estamos seguros de que, chegados a este ponto, alguém poderia pensar ser banal todo o anterior. E efetivamente é!, mas são precisamente as idéias banais (em aparência) as interessantes, desde o ovo de Colombo à maçã de Newton.

O mérito de toda esta gente de sobrenome estranho (Wirth, Dijkstra... como são pronunciados?; não sabemos) é duplo:

- ter traçado um caminho, esclarecendo publicamente coisas que talvez alguns programadores já faziam por instinto;
- colocar à nossa disposição instrumentos práticos para obter uma implementação cômoda dos tipos mais complexos, que nos permitem fazer frente às mais variadas necessidades.

É inútil dizer que na maioria das outras linguagens tradicionais de alto nível não poderíamos fazer nada pelo estilo, e teríamos que apanhá-la com tipos pré-definidos (números ou como muitas, tabelas). Com os novos meios se ganha sobretudo em clareza; além disso, a linguagem compiladora está agora em condições de assinalar, no momento da tradução do programa, erros de

forma que poderiam corresponder a erros de essência. Coloquemos o caso de que, por algum estranho motivo, seja atribuído a uma variável "y", definida do tipo diatrab, um valor como sábado ou domingo: o compilador, antes que o programa seja executado, se dá conta em seguida de que sábado e domingo não podem ser atribuídos a uma variável que não inclua as festas, e nos indicaria.

Outra pequena observação em relação com o fragmento de linguagem estruturada que acabamos de ver: quando é definido um novo tipo por enumeração, como no caso de diasem, está sendo dada de vez uma determinada disposição que é a mesma da enumeraçao. Assim, em nosso exemplo, a segunda-feira está antes que a terça, que está antes que a quarta, etc. Isto é o que nos permite definir depois outros tipos por classe, como vimos para diatrab.

Tipos compostos ou estruturados

Os tipos compostos (também chamados estruturados) podem ser construídos geralmente - é o momento de dizê-lo - por meio das seguintes operações

- produto cartesiano,
- união discriminada,
- array ou matriz,
- conjunto potência,
- seqüência.

Temos que destacar que foi apontada uma situação ideal, pois nenhuma linguagem é hoje em dia tão potente como para complementar todas estas possibilidades, oferecendo-as ao programador para que sejam fabricados seus própios tipo. Façamos a chamada rapidamente, dando simples exemplos ilustrativos.

O produto cartesiano parte de dois conjuntos genéricos, A e B, para obter o conjunto-produto C = A x B pegando todos os possíveis pares de elementos, ordenados tomando o primeiro de A e o segundo de B. Aqui um exemplo esclarecedor:

```
TYPE naipe = (copas, espadas, ouros, paus);
     valor = 1..10
     carta = (p:naipe; v:valor)
```

Sem aprofundar-nos em questões sintáticas (recordemos so-

mente que os dois pontos, como já vimos, precedem a indicação do tipo) deveria estar claro que o tipo estruturado *carta* está formado por pares de elementos ordenados, o primeiro do tipo naipe, o segundo de tipo valor. Assim, um par como (espadas;1) identifica o ás de espadas. Definindo por enumeração o tipo simples valor, poderíamos ter, obviamente, elementos estruturados como (espadas; dama) (paus; rei), etc.

Outros exemplos parecidos poderiam ser:

```
TYPE cliente = (no:nome; end.:endereço; cp:código postal;
               cid: cidade); artig = ( cod:código; des:descri-
               ção; dp:depósito);
```

Esta estrutura corresponde em Pascal, em essência, ao tipo RECORD que é dado como standard. Aqui um exemplo simples:

```
TYPE fecha = RECORD
             d:dia;
             m:mês;
             a:ano;
             END;
```

Geralmente RECORD e END são palavras reservadas que também fazem papel limitadores, isto é, estão no lugar dos parêntesis dos exemplos precedentes. É supérfluo dizer que é subentendido qual nome, endereço, dia, etc., haviam sido definidos anteriormente.

Quanto à união discriminada, parte de dois ou mais conjuntos para constituir um terceiro, no qual estão presentes tantos os elementos do primeiro, como os do segundo. Serve para misturar os tipos, fundindo-os em um de caracter mais geral no qual, apesar de tudo, cada subtipo conserva sua própria estrutura, ou seja, sua indentidade. Esta é discriminada ou distinguida por meio de um oportuno campo comum, cujo conteúdo é nitidamente diferente nos dois casos.

Explicaremos com um exemplo, também ilustrado na Figura.3:

```
TYPE autonac = (constr:fabricante;matricula:numatr;
               propriet:pessoa;primereg:ficha);
      autoest = (constr:fabricante;matricula:numatr;
               nacorig:naçao);
       auto = nacional:autonac;estrangeiro:autoest);
```

Aqui, os tipos *autonac* e *autoest* (referentes a automóveis nacionais e estrangeiros), já estruturados por sua conta, têm em comun o campo constr(nome do construtor), enquanto que no resto

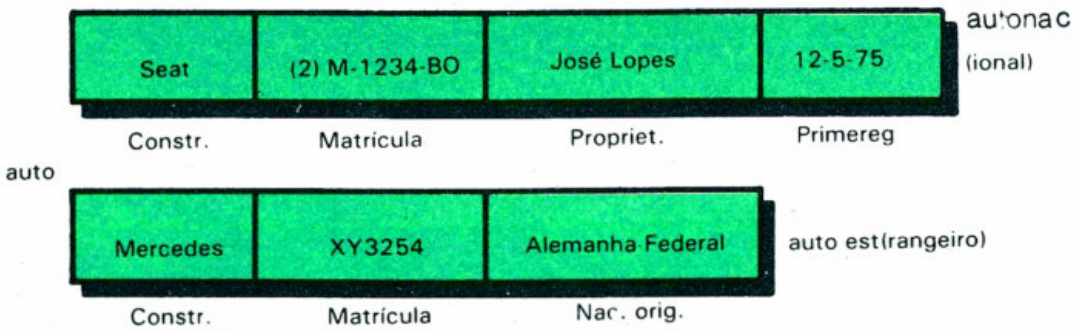


Figura 3 — Com a operação de união (discriminada) é possível unificar sob uma mesma categoria os tipos "com variantes", sobre a base do conteúdo de um campo discriminante comum.

têm diferenças apreciáveis (o campo nacorig - nacionalidade de origem - que não tem sentido com carros de tipo autonac). O novo tipo *auto* definido por união (também se diz função OR) permite distinguir de uma porção aparentemente uniforme os carros nacionais dos estrangeiros, em base ao campo comum constr. Aqui deixamos ao leitor que imagine como se pode pensar em listas separadas de construtores locais ou, mais drasticamente, em uma subdivisão do campo constr em dois subcampos, o primeiro contendo um N ou um E, segundo os casos.

Esta estrutura de dados está presente no Pascal com o nome de RECORD com variantes.

Vejamos agora os tipos compostos formados como arrays, já familiares aos conhecedores do FORTRAN ou do BASIC. O tipo array (que quer dizer "matriz") é um conjunto *matr* obtido mediante um conjunto *ind*, dos índices, e outro *val* dos valores. Cada um dos índices pode ser associado a um dos valores. À diferença dos vetores ou tabelas das linguagens tradicionais, o array, segundo a concepção de Hoare, permite também a utilização de índices não numéricos.

Em Pascal a forma geral é a seguinte:

```
TYPE matr = ARRAY[ind]OF VAL;
```

por exemplo:

```
TYPE diames = ARRAY [MES] OF 28..31;
```

o índice *mês* é um não tipo número definido anteriormente por enumeração dos meses (de janeiro a dezembro), enquanto que o

campo dos valores está definido por intervalos (dá somente os valores possíveis, mas não os associa com o índice; aqui; por exemplo, não diz que o primeiro elemento do *mês* tenha 28 dias). Uma possível aplicação deste tipo estruturado recém definido poderia ser:

```
VAR dias: diames
```

e no curso do programa poderia ser utilizada a atribuição:

```
dias[fevereiro]: = 28;
```

Para informações dos que não o sabem, antecipamos que, em Pascal, o sinal = serve para definir tipos ou assinalar a semelhança entre termos comparados, enquanto que para a atribuição se faz preceder o = por dois pontos (: =), assim é eliminado o equívoco de atribuições como N = N + 1, (escrevendo N: = N + 1 é mais evidente que N é "convertido" em N + 1...).

Outro exemplo, altamente eloqüente (isso esperamos) é:

```
TYPE naipepoker = (corações, diamantes, trevos, lanças);
     valpoker = 1...3;
     cartapok = (naipe: naipe poker; valor: valpoker);
     baralpok = ARRAY [1..52] OFcartapok;
```

Vejamos agora o conjunto potência. Se parte de um conjunto de dados e é formado o de todos os subsistemas que o compõe, incluído o conjunto vazio e o total. Imaginemos um rebanho de quatro ovelhas: a, b, c, d. O rebanho-potência estaria formado por: o" rebanho vazio"(nenhuma ovelha), os rebanhos de um ovelha (a...d), os de duas ovelhas (ab, ac, ad, bc,...cd), os de três ovelhas (abc, abd....bcd) e o abcd. Não podemos estranhar que exista um "rebanho vazio": pense em um pastor a quem roubaram todas as ovelhas.

Em Pascal, a palavra reservada que serve para definir um conjunto potência é SET. Aqui um simples exemplo:

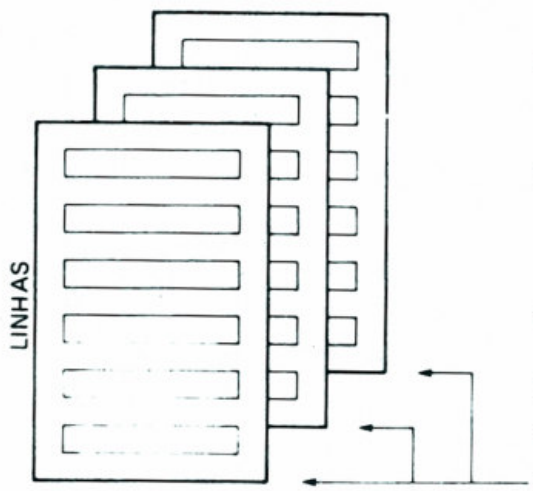
```
TYPE colorbase = (vermelho,amarelho,azul);
     color SET OF colorbase;
```

depois do qual é possível, uma vez definida uma certa

```
VAR cor:color
```

fazer uma atribuição como cor: = (amarelo, azul). Como informação diremos que esta estrutura tão cômoda em alguns casos é criticada sobretudo pela dificuldade de complemetações eficientes. De fato, para identificar a cada elemento do conjunto potência são necessários ao menos tantos bits quantos elementos tenha o conjunto do qual se parte. A identificação mais simples que se possa imaginar consiste em colocar um bit a 1 ou a 0, segundo que o elemento esteja ou não presente (uma complementação que utilizasse uma gravação independente do resultado exigiria espaços proibitivos). Por exemplo, no caso da tricomia acima exposta são necessárias 5 bits e a atribuição cor: = (amarelo, azul), corresponderia a 011. Imagine o que aconteceria com conjuntos base inclusive de poucas centenas de elementos.

Ainda mais pesado do ponto de vista da realização é o modo seqüência, que é fabricado com um conjunto X de "n" elemen-




```
TYPE direção =          RECORD
                        Rua: STRING;
                        Núm: INTEGER;
                        Cp: INTEGER;
                        Cid: STRING;
                        END;

TYPE nome prop. =       RECORD
                        Nome: STRING;
                        Sobren: STRING;
                        END;

TYPE linha =            RECORD
                        nome: nome prop;
                        end: endereço
                        número: INTEGER;
                        END;

TYPE página =           ARRAY (1.. núm.
                          linhas) OF linha;

TYPE guíatel =          FILE OF página;

VAR guía; guíatel:

PAGINAS
```

Figura 4 — Com os tipos estruturados é fácil construir, com refinamentos sucessivos, as mais complexas construções de dados compostos. Na figura temos o exemplo de um guia de telefones, descritas como um arquivo por array-página, por sua vez composto por record-linha, contendo cada um uma direção, subdividida em... Para facilitar sua compreenção, a descrição formal da figura é um bottom-up. De todas as formas é evidente o uso do mecanismo nidificado, típico da programação estruturada.

tos desde a sequência vazia àquelas formadas por 1, 2..., "n" elementos.

```
TYPE cadeia = SEQUENCE caracter
     identificador = (primcar:letra;resto:SEQUENCE
     (1:letra;d: digito)).
```

tos tomando todas as possíveis seqüências (ordenadas) de elementos, O primeiro caso define uma cadeia como um conjunto qualquer de caracteres ordenados. No segundo se trata de um identificador, entendido como a sucessão de letras ou cifras, nas quais o primeiro caracter é sempre uma letra. Se alguém não vê muito claro tudo isto, deve ter presente que não estamos aprofundando no campo estruturado, com o qual estes não são mais que simples exemplos (inclusive algumas coisas não aprofundaremos nem sequer posteriormente). Em Pascal, o tipo que mais é parecido à seqüência é o tipo "file", que constitui uma complementação reduzida, dado que se refere somente à memória externa. Depois de todo o falado sobre os tipos "abstratos" pode parecer uma traição, mas o fato é que também nos terrenos mais avançados se deve distinguir entre sonho e realidade, ideais e limites objetivos.

Na Figura 4 encontramos um exemplo resumido que deveria esclarecer o procedimento top-down com suas fases de refinamento sucessivos, passo a passo ("step refinements") na definição de dados estruturados.

Conclusões provisionais

Na Figura 5, para comodidade do leitor é representado o quadro sinóptico das estruturas presentes em Pascal, antecipando o que explicaremos melhor mais adiante (nos referimos então, também, aos "pointers" ou ponteiros, que nada têm que ver com os cachorros de casa). Para concretizar a panorâmica dos préstimos que podem ser conseguidos com a ajuda de Hoare e companhia, tratarentos aqui um caso de aproveitamento da recursividade ("recursivity" em inglês). Todo processo recursivo tem um carater estranho: chama a si mesmo. Teremos ocasião de voltar a ouvir falar de recursividade, o propósito dos programas. Este conceito moderno é aplicado aqui à definição de dados estruturados e consiste no fato de que a expressão chama a si mesma desde o interior de sua definição:

```
TYPE expressao = SEQUENCE termino;
```

```
termino = (addop:operador;f:SEQUENCE fator);
fator = (muitop:operador;p:primario);
primario = (CONST:(val:real),VAR:(id:identificador),
            entreparent:(e:expressao));
operador = ( + ,-,*,/);
```

(Para ser sinceros, no Pascal corrente esta estrutura 'top-down" seria rechaçada pelo compilador...)

Não tem que desesperar se não consegue fazer outra coisa que intuir para que serve a complexa construção precedente (mas deveria ficar claro que se trata de uma expressão algébrica...). De todas as formas , tem que ser explicado que, geralmente, na notação de Hoare a vírgula expressa alternativa OR e o ponto e vírgula AND ou sucessão. Deve ser suficiente observar que "expressão" figura dentro de sua definição.

Para finalizar é necessário fazer notar novamente que o aproveitamento de todas as vantagens que poderiam ser esperadas, em teoria, do paralelismo entre a estrutura dos dados e os procedimentos de elaboração, não é completa, nem muito menos perfeita, nas

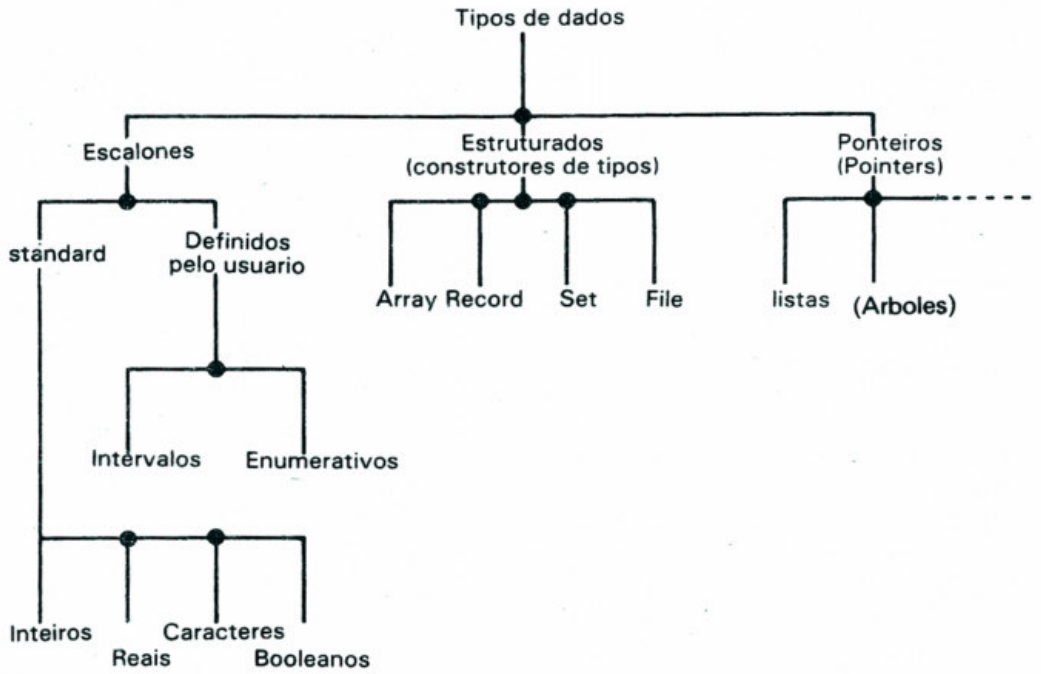


Figura 5 — Quadro sinótico dos tipos oferecidos pela linguagem Pascal.

linguagens estruturadas do mundo real. O ideal, de fato, seria que fossem oferecidas possibilidades mais amplas para definir, junto aos tipos, operações características sobre eles. Cortamos a cabeça: nem sequer o tão celebrado Ada (nota 1) cumpre em sua totalidade esta promessa, que estabelece o desdobramento de uma potência e uma flexibilidade enorme, tornando possível, entre outras coisas, mudar as modalidades operacionais sem necessidade de modificar as instruções.

De todo modo, agora que estamos a ponto de introduzir-nos no mundo do Pascal popular, não deve esquecer um conceito que tentaremos desenvolver amplamente: As estruturas dos dados e dos programas estão intimamente ligadas.

Fixemos bem esta frase em nosso cérebro, porque vale para todos os ambientes de programação, por pobres espartanos que sejam. Com isto nos adiantamos a uma pergunta que é ouvida freqüentemente: mas se eu tenho que trabalhar em BASIC, para que me serve estudar o Pascal se depois, em BASIC, não possuo os instrumentos adequados?

A resposta tem três itens:

- não é nada mal acostumar-se a trabalhar em Pascal;
- o Pascal é altamente formativo, abrange idéias de validade geral;
- muitas destas idéias podem ser transportadas, com adaptações mais ou menos complicadas, também ao BASIC (melhor se for um dialeto moderno, estruturado).

A propósito das estruturas de dados, em BASIC faltam certamente os meios para realizar com comodidade, vamos supor, um tipo pilha (stack). E mais, o tipo stack sempre será um clandestino, que subiu a bordo sem documentos, ou melhor, escondido sob um disfarce falso. Não teremos que apanhar com um vetor, o que trabalha com grandes responsabilidades, porque o controle do stack e da legitimidade de suas operações deverão ser feitas tudo por nós mesmos (o intérprete BASIC não nos ajudará em absoluto, pois não sabe nada de pilhas nem de qualquer outra coisa que não seja do tipo integer, real, string e vetor). A idéia de um stack é válida nos casos em que este seja oportuno e convém que pensemos nele. Também com o BASIC.

NOTA 1. A já citada Ada, condessa de Lovelace, era, em vida, a filha do poeta inglês do século XIX Lord Byron. Apaixonada pela matemática, colaborou ativamente com o que pode ser definido

com o inventor do primeiro computador, com um programa registrado em memória: Charles Babbage (1792-1871). Porisso, a bela Ada é considerada, ainda que muito exagero, a primeira programadora da história. Daqui o nome dado à linguagem.

CAPÍTULO III

A TARTARUGA GUIA NOSSOS PRIMEIROS PASSOS EM PASCAL

Bottom-up: programação ascendente (ou quase)

Depois da panorâmica top-down dos capítulos anteriores procederemos agora justamente ao contrário com uma orientação bottom-up (programação ascendente).

No desenvolvimento concreto dos programas é uma prática bastante freqüente que os detalhes de procedimento (temos que nos acostumar a este termo, que em Pascal designa o que em outras linguagens é denominado subprograma ou subrotina) sejam desenvolvidos em primeiro lugar, especialmente se tratar de partes delicadas, que realizam um papel central no programa e que talvez possam ser reutilizadas mais vezes em outro lugar com ou sem modificações. Exemplo: um processo (se diz algorítmo) de ordenação ("sort") ou, mais banalmente, a procura do maior elemento em uma matriz de dados. Para ligar as peças do quebra-cabeças tem que identificar pelo menos as principais.

Não contradiz tudo isso o processo top-down? Em absoluto: uma visão de conjunto é indispensável, só que na fase "construtiva" freqüentemente faz mais de "fundo" que outra coisa. De todo modo trata-se de estilo e gosto pessoal; portanto deixemos os discursos e entremos de cheio na matéria. Para fazê-lo, adotamos a técnica de exposição do professor Kenneth Bowles, da Universidade Californiana de San Diego. É o pai da versão Pascal UCSD (University of California at San Diego). Em realidade, o UCSD Pas-

cál é, mais que uma linguagem, um sistema operacional, que integra ao Pascal. Por desgraça, os fins que nos propomos e o pouco espaço disponível não permitem fazer mais que uma rápida alusão às modalidades com as que no ambiente UCSD são editados e desenhados os programas.

A aproximação de Bowles é muito útil no plano pedagógico, já que não requer dos alunos nenhuma noção prévia particular, nem sequer de matemática elementar, pelo menos no primeiro momento. De fato, faz referência a uma tartaruga (Turtle), imaginário animalzinho que está no centro da linguagem Logo. Como que acrescentar que as instruções inerentes à tartaruga não fazem parte da linguagem standard, mas foram acrescentadas com fins didáticos, seguindo a técnica de "biblioteca".

Mas, vejamos como podemos nos familiarizar inicialmente com a tartaruga, e assim dar também uma olhada rápida ao UCSD Pascal. Ao princípio aparece uma linha de ordens simplificadamente do tipo seguinte:

Command: E (dit, R (un, F (ile, C (ompile, X (ecute).

É um simples menú interativo, cujas opções são selecionadas teclando somente a letra inicial da ordem: E por Edit, F por File, etc. Em nosso caso teclaremos a X de eXecute, depois da qual escrevemos a seguinte palavra:

TURTLE

É o abre-te Sésamo correto. Em palavras mais sérias quer dizer que chamamos ao programa homônimo da biblioteca de programas (já tem que estar no disco, senão será produzido um erro). Depois do habitual zumbido da unidade de disco (drive), aparece no centro da tela a flechinha que representa o imaginário animalzinho, que poderá então ser movido à vontade. A Figura 1 ilustra o efeito de algumas ordens. É fácil de compreender, mas de todas as formas aproveitamos para recordar que a "Turtle Geometry" é uma geometria de movimentos relativos. Estes se referem à posição atual da flecha: pode ser girada em sentido anti-horário (ângulo positivo) ou horário (ângulo negativo) ou movida para frente (valor positivo) ou para trás (valor negativo). No caso da figura 1 a sucessão é: ao princípio a tartaruga está em posição de partida; depois se vê o efeito da ordem MOVE (40) que a move 40 passos elementares; TURN (90) a faz girar um ângulo reto; a tartaruga é logo movida 20 passos. Na quinta figura, ao contrário, apa-

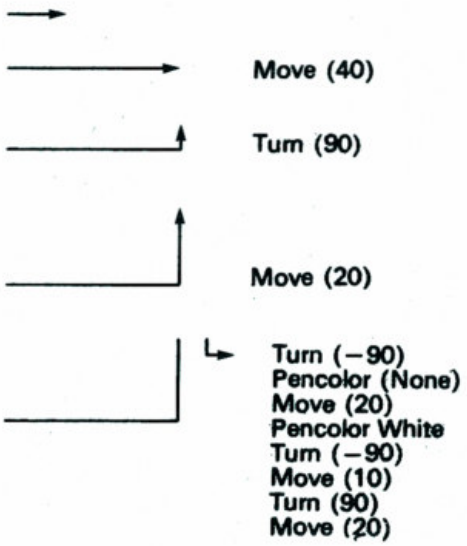


Figura 1 — Movimentos típicos da tartaruga (representada por uma pequena flecha). No quinto gráfico se tem o resultado da série de ordens assinaladas ao lado. PENCOLOR(NONE) dá lugar a um deslocamento sem rastro visível.

rece o efeito de uma seqüência de ordens entre as quais são encontradas PENCOLOR (NONE) e PENCOLOR (WHITE), que servem, respectivamente, para anular e restabelecer a ação da pena imaginária associada à tartaruga; no primeiro caso, o animalzinho será movido sem deixar rastro (com parâmetros diferentes a WHITE é possível, com um monitor em cor, traçar linhas em cores).

Os números assinalados entre parêntesis dependem do computador e são expressos em função da altura e da largura da tela. Uma vez familiarizados, graças ao programa de biblioteca TURTLE, com as ordens, em execução direta, é a ocasião para tentar fazer um programinha. Com este fim deve ser abandonada TURTLE e voltar ao menú geral das ordens, escolhendo desta vez a E de Edit:

```
PROGRAM primtort;
USES TURTLEGRAPHICS;
BEGIN
   MOVE (50);
   TURN (120);
   MOVE (50);
   TURN (120);
   MOVE (50);
   READLN (* espera return *)
```

É um programa de uma banalidade surpreendente, cuja ação, encerrada entre os "delimitadores" BEGIN e END, consiste em traçar um triângulo de 50 unidades de lado. Mas aproveitamos para introduzir a sintaxe pascaliana. Diremos, antes de tudo, que são admitidas maiúsculas e minúsculas, mas que nós usaremos para distinguir as palavras definidas pelo programador das maiúsculas (alguma vez com a inicial ou alguma letra intermediária maiúscula). Isto ocorrerá com os nomes de programas, procedimentos e dados (ou, como é dito na gíria pascaliana, "identificadores") ou então, com os comentários (podem ser inseridos em qualquer ponto, delimitados por um par de asteriscos: *comentário*). Duas observações: a instrução READLN provoca aqui a espera de que seja pressionada a tecla Return (neste caso específico serve para manter a figura na tela, pois de outra forma desapareceria com o END do programa). Segunda observação: a instrução USES TURTLEGRAPHICS serve para invocar o correspondente software de biblioteca (não tem que confundí-lo com TURTLE, que é um intérprete das ordens dadas à tartaruga desde o teclado), com o qual o compilador completa o standard do Pascal com as instruções de tipo turtle. Por motivos de espaço e simplicidade, nos outros exemplos não mais serão inseridos USES TURTLEGRAPHICS, ainda que nos casos reais sempre tenham que ser colocados.

Lamentavelmente, à diferença do cômodo BASIC, o Pascal é uma linguagem não interpretada e semicompilada: a compilação produz um código intermediário chamado "p-code", que por sua vez é interpretado no momento da execução (lhe aconselhamos esquecer isto agora, pois ocorre o perigo de confundir as idéias). Afortunadamente, o USCD Pascal prevê uma espécie de atalho. Terminada a edição e reintegrados ao menú das ordens gerais (com a ordem Q = "Quit") poderá ser escolhido o R de RUN. A primeira vez não é como no BASIC, e efetivamente, aparece uma mensagem ("compiling...") que nos convida a ter paciência. Necessitaremos muita, especialmente com os programas longos, mas se tudo está sintaticamente O.K., nosso programa será lançado e executado automaticamente. Em compensação, uma vez compilado nosso programa (e guardado em disco), sai em seguida e é mais veloz que os de BASIC. Assim, as compilações às quais nos obrigam os compiladores (deixamos que imaginem a dura tarefa de eliminar erros, muito mais complicada que em BASIC, com todas as idas e vindas entre Editor e Compilador... mas, existem os que se divertem com isto!) ficam compensadas por benefícios evidentes.

Mas ao pretender ser este um curso de introdução, lhes reme-

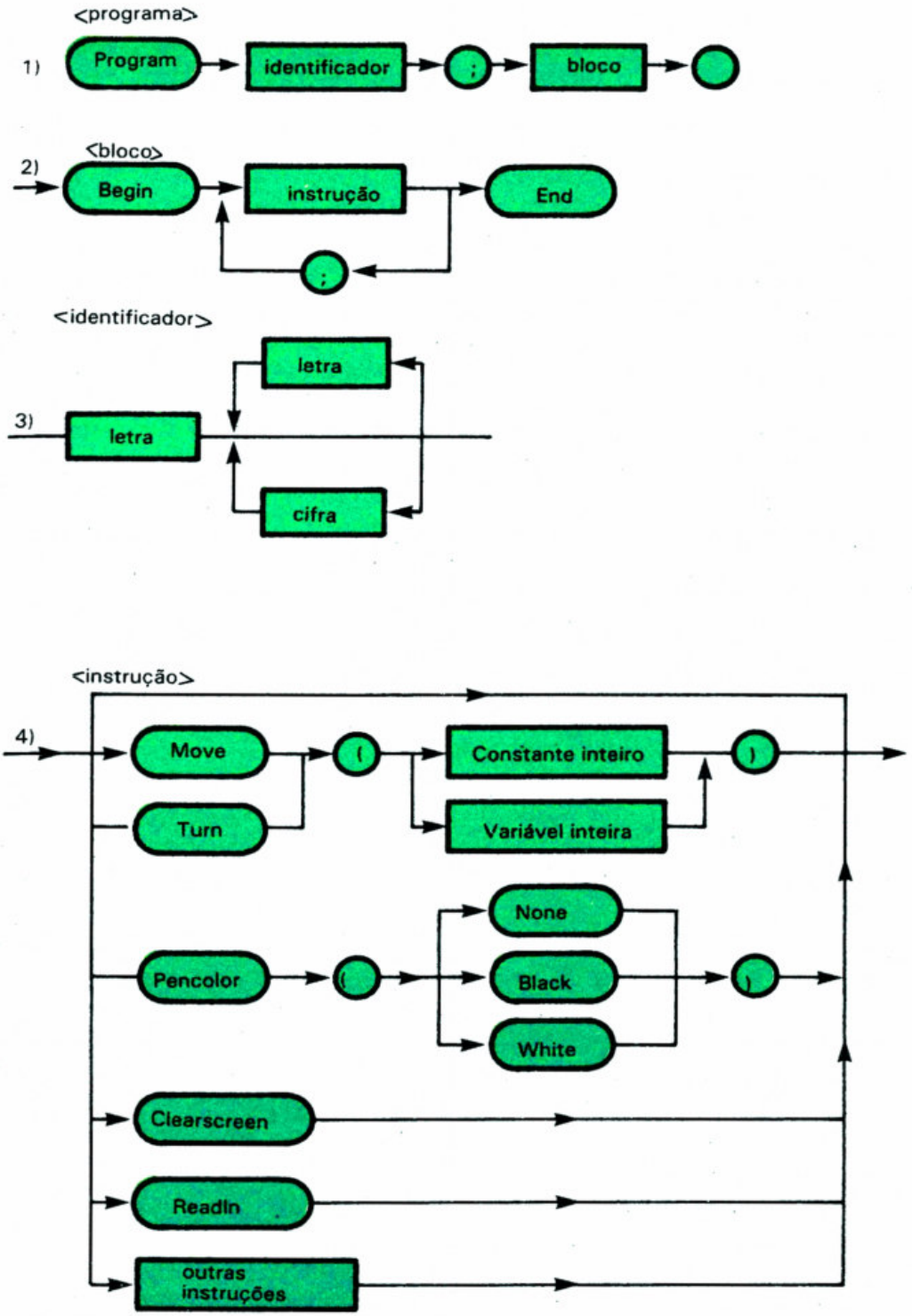


Figura 2 — Quadro da nomenclatura e das ordens do Pascal vistas até agora. Os símbolos retangulares encerram coisas a definir posteriormente.

teremos aos manuais para todos estes detalhes operacionais e passamos agora à sintaxe. Para isto faremos uso da Figura 2, na qual retângulos e círculos expressam as regras formais do jogo, limitadas às poucas vistas até agora. Se procede com o sistema top-down; os retângulos representam coisas ainda por precisar detalhes.

Em 1) se diz que um < programa > (o nome de um objeto está encerrado entre parêntesis "angulares": < programa > , < bloco > , < instrução > , etc.) está composto pela palavra reservada PROGRAM seguida de um identificador, um ponto e vírgula, um < bloco > e um ponto e vírgula. Mas, o que é um bloco? Esclareceremos em 2): em essência é uma série de instruções separadas por ";" entre as palavras BEGIN e END. Este sinal de pontuação (;) é importante em Pascal como já vimos em 1).

Mas continuemos: O que é um identificador? Uma série de letras e cifras cujo primeiro caracter tem que ser uma letra. Acrescentamos que o número de caracteres significativos é oito. Com 2) e 3) é entendido facilmente o significado das concatenações simbólicas: as linhas em paralelo que partem de um nó expressam alternativa (lógica OR) e uma linha sem símbolos significa "possível ausência" - por exemplo, em 3) a primeira letra do identificador pode ser também a última -. Ao contrário, aquelas que voltam atrás denotam possibilidade de repetição, ou dito de uma forma culta, "iteração".

Em 4) (Figura 2) temos finalmente o resumo do pouco visto até agora a propósito de instruções, com dois pequenos acréscimos como CLEARSCREEN, para o apagar da tela e o retângulo OUTRAS INSTRUÇÕES que nos avisa que contudo ficam muitas outras. Quanto aos retângulos que contêm CONSTANTE INTERNA e VARIÁVEL INTERNA damos por conhecidos seus conceitos desde o curso de BASIC (volumes 5 e 6); com instruções tipo MOVE (lado) ou TURN (ângulo) consegue-se uma maior flexibilidade.

Aproximando-nos aos procedures

Fazemos aqui algumas observações:

- a sintaxe em Pascal é indubitavelmente um pouco complicada mas a situação é idêntica em todas as linguagens (o BASIC é somente mais pobre, isso é tudo);
- o símbolo de atribuição em lugar de " =" comum ao BASIC e as linguagens como FORTRAN e COBOL, é ": =" (significa, que a variável do primeiro membro "toma" o resul-

tado da expressão da direita), enquanto que o " = " fica destinado às expressões lógicas;

- as palavras reservadas (as escritas dentro de semielipses na Figura 2, por exemplo) não podem ser adotadas como identificadores;
- o separador das instruções em Pascal é o ";" unicamente, enquanto que o "." decreta o fim do programa;
- a palavra reservada BEGIN é um separador e não vai seguida de ";", enquanto que a associada END (tem que ter exatamente um para cada BEGIN) requer o ";" (exceto se é o último do programa, em cujo caso colocaremos "."), ainda que, como compensação, a instrução que o precede não o necessita (ver desenho 2 da Figura 2). Idênticas regras sobre os ";" valem para outras palavras reservadas que funcionam como delimitadores , como FOR, DO, IF, ELSE, etc., mas para simplificar não falaremos agora delas, convidando-lhes a seguir os sucessivos exemplos.

Em resumo, contrariamente ao que se acredita, em Pascal Return, espaço em branco e (comentários) servem somente para separar as palavras, e o programa visto (o único até agora) poderia muito bem ter sido escrito assim.

PROGRAM Primtort; BEGIN MOVE (50); TURN (120); MOVE (5); TURN (120); MOVE (50); READLN (* espera return) END.

O importante é que cada ponto e vírgula esteja em seu lugar! Por outro lado, os pascalistas nunca fazem semelhantes seqüências; e mais, recorrem aos famosos escalonamentos que caracterizam o aspecto dos programas Pascal, como veremos. Mas é hora de mostrar algo mais sério.

```
PROGRAM stardust;
VAR escala: INTEGER;
PROCEDURE estrela (diamens: INTEGER);
VAR ind: INTEGER;
BEGIN
  TURN(-18); (* centrar estrela no 'caule' *)
  PENCOLOR (WHITE);
  FOR ind: = 1 to 5 do
    BEGIN
     MOVE (DIAMENS); TURN (144);
    END;
```

```
  TURN (18) ( * tartaruga em posiçao original* )
END: ( estreia* )
BEGIN ( programa principal * )
  escala; = 20;
  TURN (45); MOVE (escala*8);
  estrela (escala*4);
  MOVETO (0,0); TURNTO (165);
  MOVE (escala*4)
  estrela (escala*8)
  MOVETO (0,0); TURNTO (150);
  MOVE (escala*6)
  estrela (escala* 6)
END.
```

O resultado da execução é o traçado das três estrelas da Figura 3b.

Veremos em seguida a filosofia de fundo. O programa Stardust é composto de um procedimento Estrela que é chamado três vezes, cada vez com um valor distinto do parâmetro "dimens". Todo parâmetro é uma variável cujo valor é fixado de forma externa ao procedimento, permitindo-lhe "especializar-se", para dizê-lo de alguma maneira. Na sintaxe pascalina a definição está clara:

```
PROCEDURE estrela (dimens: INTEGER)
```

Isto nos diz que as instruções que seguem fazem parte do procedimento de nome Estrela, que tem um parâmetro, entre parêntesis, chamado "dimens". Os dois pontos que seguem a "dimens" especificam o "tipo" INTEGER (inteiro) no caso em questão. As instruções de Estrela são as compreendidas entre o BEGIN e o END: um pouco mais adiante fica claro para que serve "dimens": MOVE (dimens), seguido de TURN (144) diz que o lado da Estrela pode variar enquanto que a rotação é fixa, e igual a 144 graus. No programa principal (main) no qual, para maior clareza, foi colocado o comentário (* programa principal *), Estrela é invocada três vezes, cada vez com um valor diferente entre parêntesis. Quando é chamado um PROCEDURE que usa parâmetros, na instrução de chamada deverão ser incluídos, em cada caso, os valores que queremos que tomem os parâmetros nessa ocasião. Assim, por exemplo, a instrução Estrela (escala *8) "passa" - realmente se diz assim - um valor igual a 8 vezes a variável "escala", ao parâmetro "dimens" de Estrela. Esta é a maneira de conseguir que o procedimento Estrela possa desenhar estrelinhas de diferentes tamanhos

a)

```
PROGRAM Stardust
VAR:.....

  PROCEDURE Estrela (...
  VAR:....

    PROCEDURE Hipotet;
    VAR:...
    BEGIN


    END; (*Hþotet*)

  BEGIN
  TURN (-18)


  END; (*Estrela*)

BEGIN (*programa principal*)
Escala: = 20


END
```

b)

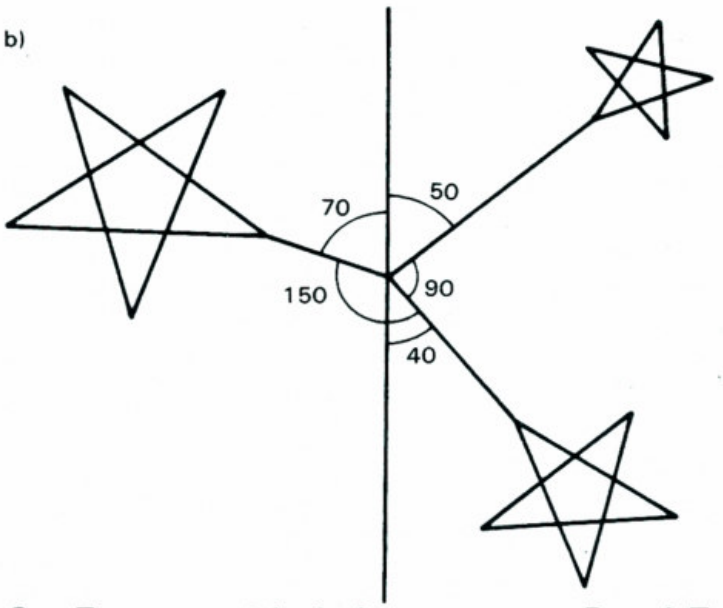


Figura 3 — Estrutura aninhada de um programa Pascal. Em a) está incluído um procedimento imaginário Hipotet, interno a estrela. Em b) é visto um desenho traçado pelo programa Stardust incluído no texto.

(Fig. 3).

Quem sabe BASIC notará a analogia com as GOSUB, salvo que aqui não existem números de linhas, os parâmetros são passados desde o exterior e a subrotina é chamada diretamente por seu nome: realmente é como se fossem acrescentadas novas ordens à listagem de instruções.

Estruturas de "caixas chinesas"

Mais que analisar detalhadamente o que faz Stardust, o que nos dá urgência é compreender sua organização. Nos referiremos à Figura 3, onde em a) está representada a armação de Stardust, vazia, por assim dizer, de conteúdos (como a um caranguejo do qual tiramos a carne). E mais, à custa de complicar a vida dos leitores, acrescentamos um terceiro procedimento: Hipotet (ou seja, imaginário), "aninhado", por sua vez, dentro de Estrela. É um jogo de caixas chinesas: o programa principal contém os procedimentos de primeiro nível que, por seu lado, podem conter outros de segundo nível, e assim sucessivamente.

Os PROCEDURES fazem parte de um conceito mais amplo: o de módulos. Com este termo são denominadas, além dos procedimentos, as funções definidas por meio da palavra reservada FUNCTION. Seu significado é bastante intuitivo. Os parâmetros (entre parêntesis) que empregam as funções em sua definição são chamados também "argumentos". Trata-se de nomes familiares por pouco que saibam de matemática.

Tanto os módulos como o mesmo programa global são compostos de duas partes:

- declarativa
- executiva

Como seus próprios nomes indicam a primeira compreende todas as "declarações" necessárias para a correta interpretação do módulo: nome, parâmetros, tipos e variáveis, no entanto a segunda se refere propriamente às sentenças ou ordens executivas.

Os quatro elementos possíveis mencionados na primeira não estão sempre presentes, mas se aparecem devem suceder-se exatamente nesta ordem: nome (ou etiqueta), parâmetros, tipos e variáveis para um módulo. A regra geral prevê a sucessão, no âmbito de um programa completo de Pascal, das seguintes coisas: nome, constantes, tipo, variáveis, procedimentos e funções. Sempre que se use qualquer destes elementos, deverá explicitar-se previa-

mente em Pascal. Para maior facilidade, nos concentramos nas variáveis. Todas as variáveis de um módulo ou do programa inteiro, têm que ser declaradas, precisando o "tipo" de antemão e não no momento em que são necessitadas, como ocorre em BASIC.

Isto não é uma novidade absoluta (também o COBOL prevê uma apropriada DATA DIVISION). As variáveis que seguem à palavra reservada VAR estão deslocadas meticulosamente no âmbito dos módulos aos quais pertencem. Este fato é típico da concepção hierárquica, base do estruturalismo. E não somente é um fato formal, mas que afeta ao próprio desenvolvimento do programa. De fato:

- uma variável definida em um módulo opera também com todos os de nível inferior;
- uma variável "local" - adjetivo contraposto a "global" nesta terminologia - somente opera no âmbito do módulo no qual está definida (e eventualmente naquelas de menor nível) e NÃO deixa vestígio no exterior.

Voltaremos a este ponto com um exemplo concreto. Fixe-se agora em que um módulo que contenha outro de nível mais baixo fica na prática dividido em dois por este. Assim acontece com Hipotet que é interposto entre a parte explicativa de Estrela e a executiva. Em particular, isto ocorre quase sempre com o programa principal, cuja parte executiva - teremos que nos acostumar - está ao final, precedida por todos os procedimentos e funções em jogo. Indubitavelmente isto apresenta algum problema no caso de programas muito longos; tem que olhar a fundo a listagem para "visualizar" o trabalho completo. De todas as formas não daremos ouvido às provocações de quem, acostumados ao BASIC, no qual as subrotinas geralmente são escritas ao final, realizem comentários desconsiderados. Nos limitaremos a observar que:

- esta redação (módulos inferiores precedendo aos superiores) facilita o trabalho do compilador e, acelera a execução (acontece também em BASIC, onde basta iniciar com um GOTO para que se salte as subrotinas escritas ao princípio);
- em realidade, o top-down e o bottom-up programação. ascendente) são duas faces do mesmo problema.

A essência de Stardust

Depois de todos estes úteis formalismos, voltamos a Stardust para descrever o que faz. Vamos começar por PROCEDURE Es-

trela. Para entendê-lo, basta colocar-se na pele (neste caso carcaça) da tartaruga: depois de fixar um ângulo de ataque de -18 e de carregar com tinta branca (WHITE) sua pena, desenha os cinco lados de uma estrela de lado "dimens". Provemos com o goniômetro ou transportador de ângulos: 144 graus é a rotação que tem que realizar para passar, estando na ponta de uma estrela, do final de um lado ao início do sucessivo. Para isto se serve da construção Pascal FOR; sua sintaxe é:

```
FOR < indice > : = < n1 > TO < n2 > DO
  BEGIN

  *

  *

  END (★ loop FOR ★)
```

Em lugar de TO pode ser usado DOWNTO, em cujo caso será contado para trás. Trata-se, em essência, da mesma estrutura (FOR/NEXT) existente em BASIC, ainda que com duas limitações: a conta, no Pascal standard, somente pode ser feito com números inteiros e o passo (step) é sempre a unidade. Para outras ocasiões ficam as construções DO-WHILE e REPEAT UNTIL.

Nos resta agora o programa principal: é uma sucessão normal de deslocamentos da tartaruga e chamadas a Estrela. Entre os primeiros aparecem alguns absolutos, mediante as instruções MOVETO (x,y) e TURNTO (z), que servem, respectivamente, para alcançar o ponto de coordenada (x,y) ou o ângulo "z" do sistema de referência da tela, independentemente da posição atual da tartaruga.

Quanto às chamadas do tipo Estrela (escala *8) é apreciada sobretudo a oportunidade de ter introduzido a variável global "escala": para referir a uma mesma unidade de medida, o parâmetro "dimens". Também é útil a possibilidade oferecida pela linguagem de passar uma expressão como valor do parâmetro: se hipoteticamente tivéssemos tido o capricho de traçar uma estrela de lado (escala + inc)* 3.14/100. teria sido suficiente escrever:

```
estrela ((escala + inc) * 3.14/(100))
```

Diga-o com flores

O método da tartaruga, além de ter a vantagem de aproximar a informática à gente "peixe" em matemática, inclusive crianças,

permite fazer entender que um procedimento não é sempre e necessariamente um conjunto de cálculos, mas também um "fazer algo" (muito útil em robótica, por exemplo).
Vejamos outro simples exemplo:

```
PROGRAM poligonos;
Var escal: INTEGER;
PROCEDURE polig(numlad, comp, x, y,: INTEGER);
VAR i ; INTEGER;
  BEGIN
    MOVETO (x excal;  escal);
    PENCOLOR (WHITE);
    FOR i: = 1 TO numlad DO
      BEGIN
        MOVE (comp   escal);
        TURN (360 DIV numlad)
      END;
    PENCOLOR (NONE)
  END; (  polig  )
BEGIN (  main  )
  escala:  =  10
  polig (5, 16. -40, -40);
  polig (10, 8, 30, -40);
  polig (30, 2, -40, 8)
END.
```

Não é demasiadamente difícil dar-se conta de que o procedimento Polig - que tem quatro parâmetros: numlad, comp, x e y - traça polígonos regulares de numlad lados, cada um de comprimento comp, a partir do ponto de coordenadas (x*escal, y±escal); o consegue ao traçar lados de valor como seguidos de giros de 360/numlad graus e tudo é repetido numlad vezes, como indica o loop FOR. Fazemos notar de passagem que, para os números de tipo integer (até agora não conhecemos outros), o Pascal prevê o operador particular DIV, que proporciona um resultado inteiro ao dividir números também inteiros (a divisão entre números reais se faz com a habitual barra inclinada ou "slash").

Quanto ao main, somente serve para indicar a Polig os distintos valores dos parâmetros. São obtidas figuras que, ao aumentar o número dos lados (e ao diminuir comp, pois se não sairiam da tela), são aproximados cada vez mais a um círculo.

Vamos agora com a seguinte tarefa, que consiste em fazer que a tartaruga desenhe a planta da Figura 4. Notará em seguida que

existem várias coincidências: as pétalas, são repetidas várias vezes e também as flores, todas de cinco pétalas, cada uma composta por dois arcos. Isto nos sugere convidar aos mais voluntariosos para que escrevam um programa sozinhos. Uma pequena ajuda: provem com procedimentos "aninhados", tendo em conta que a flor está feita de pétalas que estão formadas por arcos. Advertimos de todas as formas que surgirão problemas complicados, ainda que resolúveis. Já tentaram? Comparem então sua solução com a seguinte:

```
PROGRAM planta;
VAR escal: INTEGER;
PROCEDURE petala (comp: INTEGER);
PROCEDURE arco;
VAR i:INTEGER
  BEGIN (*arco*)
   PENCOLOR (WHITE);
   FOR i: =  1 TO 10 DO
     BEGIN
      MOVE (comp * escal); TURN (9)
     END;
END; (*arco*)
  BEGIN (*petala*)
      TURN (-45); arco;
      TURN (90); arco;
      TURN (135);
     END; (*petala*)
   PROCEDURE flor (dim: INTEGER);
   VAR i: INTEGER;
   BEGIN (*flor*)
     FOR i:  = 1 TO 5 DO
      BEGIN
        MOVE (dim  * escal);petala;
        MOVE (-dim *  escal); TURN (72)
     END;
  END; (*flor*)
  BEGIN (  main  )
   escal: = 10; PENCOLOR (WHITE);
   MOVETO (0, -60 * escal);
   MOVETO (0,-40 * escal): TURN (-45);
   petala (2); TURN (45),
   MOVETO (0, -20*escal); TURN (45);
   petala (2); TURN (-45);
```

```
MOVETO (0,0);
TURN (-45); MOVE (40     escal);
flor (2);
MOVE (-40 * escal); TURN (45);
MOVETO (0, 20 * escal); TURN (45);
MOVE (20 * escal);
flor (1)
END;
```

Os comentários detalhados seriam muito longos; sugerimos novamente que estude com atenção todos os passos que o programa impos à tartaruga. O procedimento Arco deriva, imaginariamente, de Polig, à parte do loop FOR; consiste em 10 segmentos girados entre eles 9 graus, o qual, se refletirmos um pouco, quer dizer 90 graus no total, ou seja, 1/4 de círculo. Pétala provocará, portanto, a seguinte sucessão de ângulos (relativos): -45;90 (as 10 vezes 9 graus que acabamos de ver); 90 (antes do segundo Arco)

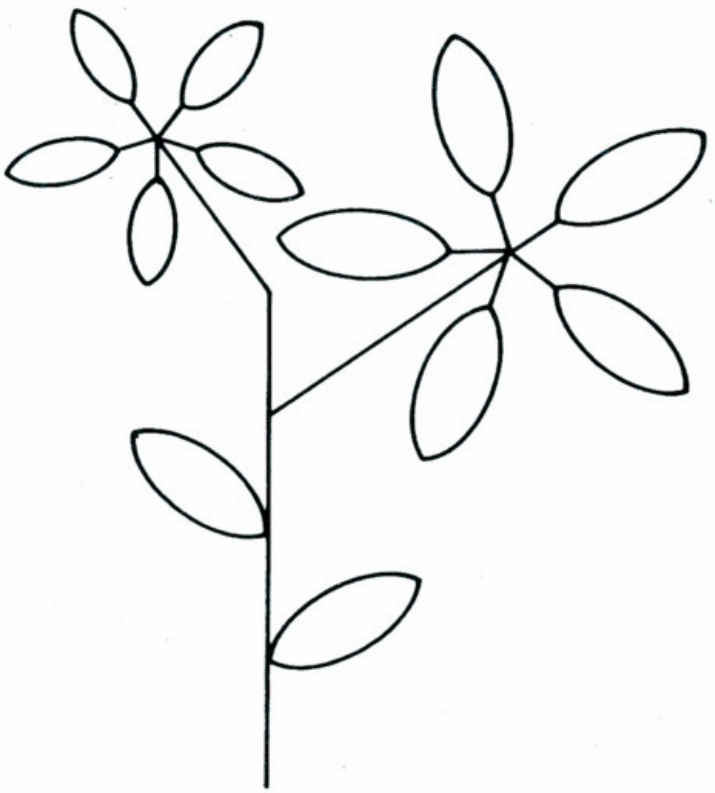


Figura 4 — Diga-o com flores. A programação modular fica fácil e eloquente com exemplos tão amenos.

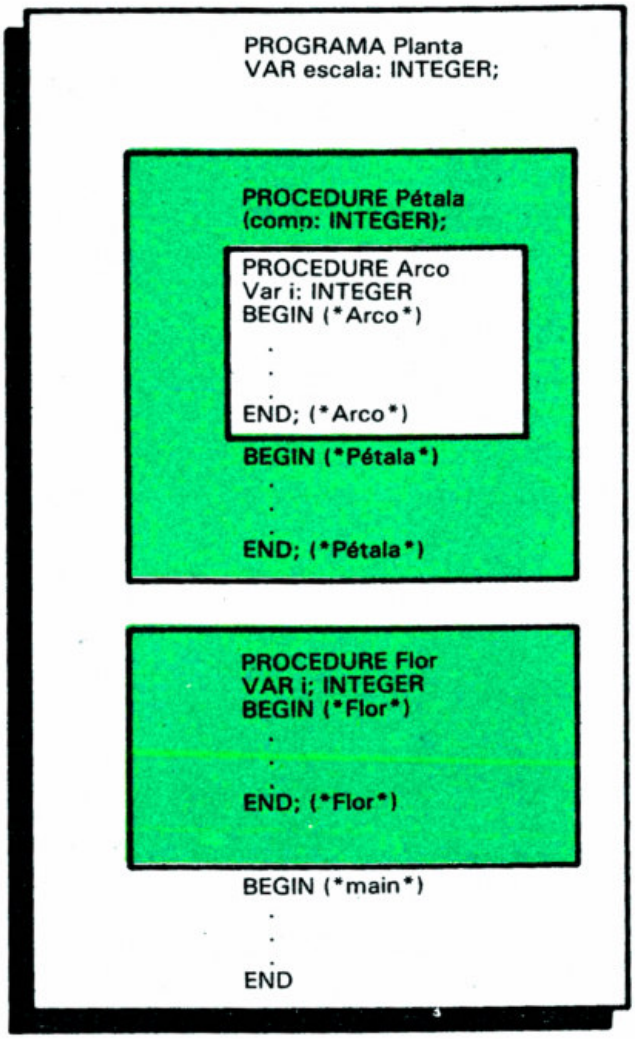


Figura 5 — Estrutura de "caixas chinesas" do programa Planta. Como é comentado no texto, não fica conveniente, como parece à primeira vista, incluir Pétala no interior de Flor. Nem tudo é conzinhar e cantar...

e outros 90 (devidos à realização do segundo Arco). Isto dá um total de 225, aos quais, antes de sair de Pétala, são acrescentados 135 graus, o que completa o giro, isto é, uma vez traçada a pétala a tartaruga fica na mesma situação de quando o iniciou. Todos os módulos seguem o critério de acabar deixando a tartaruga orientada exatamente como na entrada.

A figura 5 contém as "caixas chinesas" com as quais fabricamos o programa Planta. A partir de agora nós as comporemos somente com os esquemas típicos dos programas Pascal (ainda que, repetimos, NÃO obrigatoriamente), que servem praticamente da

mesma forma para nossos fins. Resta um assunto no ar: por que o aninhamento dos procedimentos não foi levado até o extremo que poderia ser esperado? E, mais exatamente, por que Pétala contém a Arco, mas não é incluído em Flor?

A questão é bastante delicada, e para resolvê-la convidamos a imaginar a solução alternativa (que talvez alguns de vocês já tenha feito). A estrutura seria do tipo:

```
PROGRAM planta;
VAR escal:INTEGER;
PROCEDURE flor (dim :INTEGER);
VAR i:INTEGER;
PROCEDURE petala (comp: INTEGER);
PROCEDURE arco;
VAR i:INTEGER;
  BEGIN
  ...
  END;
  BEGIN (*petala*)
  ...
END; (*petala*)
BEGIN ( * flor*)
...
END; (*FLOR*)
...(seguiria main como antes)...
```

Sem dúvida se trata de uma estrutura extremamente delicada e elegante e, efetivamente, o professor Bowles a propõe em seu livro, mas para traçar uma ou mais flores completas somente. Nós, ao contrário, queríamos resolver toda a tarefa traçando outras pétalas (neste caso folhas) isoladas. Seria lícito, nestas condições, invocar Pétala no main? A resposta é negativa. Podem provar a compilar a nova versão: o compilador será bloqueado, negando-se a prosseguir, precisamente ao chamar a Pétala. O motivo está na estrutura hierárquica do processo de compilação. Daqui é derivada a Regra seguinte:

OS PROCEDIMENTOS "VISÍVEIS" PARA UM CERTO NÍVEL E OS ÚNICOS AOS QUAIS PODE CHAMAR SÃO AQUELES DE SEU MESMO NÍVEL OU OS DE NÍVEIS IMEDIATAMENTE INFERIORES CONTIDOS NELE.

Em ocasiões, um procedimento pode invocar a si mesmo ou a procedimentos de níveis maiores..., mas estes são detalhes dos

quais falaremos mais adiante. Com o último delineamento visto do main pode ser invocada somente Flor, desde flor somente Pétala e desde esta Arco. Nós, ao contrário, necessitávamos poder chamar a Pétala no main, assim que o fizermos igual em "dignidade" a Flor. A solução será, talvez, menos bonita, mas funciona perfeitamente (o programa de Bowles serve para Flor mas não para a Planta inteira).

Disto é possível obter uma pequena moral: ainda que o aninhar possa ser útil para aumentar a clareza, aproximar-se ao "divide e domina", e em certos casos, proteger um procedimento interno, fazendo-o de uso "exclusivo" daquele no qual está contido outro (poderia ser o caso de Arco). Em geral é aconselhado manter somente dois níveis, o do main e outro mais no qual serão incluídos todos os demais procedimentos.

Por último, observe que a VARiable "i" está definida em dois procedimentos distintos. Coisas como esta, que suscitam a curiosidade e que pedem explicações, esclareceremos no próximo capítulo (às vezes é necessário um pouco de suspense).

CAPÍTULO IV

APROFUNDANDO COM A TARTARUGA

Outros formalismos úteis

O programa Planta do capítulo precedente se prestava a várias reflexões. Antes de tudo temos que afirmar que aninhar os procedimentos "até o final" não é sempre bom nem obrigatório.

Muitas vezes, no entanto, este delineamento é imposto por sua elegância; desde logo se corresponde melhor com a filosofia do "divide e domina", que consiste em subvidir mediante níveis hierarquizados, um problema complexo em subproblemas cada vez mais simples: Planta, que está formada por Flor, que está formada por Pétala... Aprendemos que um procedimento é convertido em algo privado do procedimento situado imediatamente por cima (algo assim como as variáveis locais), com o que não pode ser chamado desde níveis ainda mais externos: isto dá lugar a alguns problemas.

Quantos aos dados, tem que distinguir entre globais e locais. Desta sutil distinção dependem aspectos como a segurança e a comodidade. A segurança deriva sobretudo do fato de que é eliminado o perigo de que as variáveis necessárias somente dentro de um procedimento possam ser influenciadas pelo exterior, por despiste ou negligência. Imaginemos, por exemplo, que escrevemos em BASIC um subprograma e que queremos reutilizá-lo em outro local: devemos ter muito cuidado com os nomes das variáveis, pois em BASIC são todas globais. Em segundo lugar, e dado que

um dado local não deixa rastro no exterior do bloco no qual está definido, ou mais exatamente, nos blocos de nível superior, enquanto que em geral é válido em todos os de nível inferior contido no bloco de definição, um mesmo identificador pode ser usado em procedimentos distintos; assim não enlouqueceremos procurando nomes de dados.

No exemplo do capítulo anterior isto seria aplicado ao índice "i", usado tanto no procedimento Arco como no Flor. A variável "i" de Arco e a "i" de Flor são homônimas, mas diferentes, porque a primeira somente seria utilizável por Arco e Pétala (que contém a Arco), mas não foi. O compilador lhes atribui áreas de memória distintas. Reafirmamos este ponto com um novo exemplo:

```
PROGRAM telescop;
VAR s: STRING; liv: INTEGER;
PROCEDURE primeira;
  BEGIN
    liv: = + 1;
WRITELN (' ': liv * 2,' começa a primeira');
WRITELN (' ': liv * 2,s);
WRITELN (' ': liv * 2,' acaba a primeira');
liv: = liv-1
  END; (   primeira * )
PROCEDURE segunda;
VAR s: STRING;
  BEGIN
    liv: = liv + 1
WRITELN (' ': liv * 2, começa a segunda');
s: = ' cuco, sou a segunda!';
WRITELN (' ': liv * 2,s);
primeira;
WRITELN (' ': liv * 2, ' acaba a segunda');
liv:  = liv-1;
END; ( * segunda * )
PROCEDURE terceira;
  BEGIN
    liv: = liv + 1;
WRITELN (' ': liv * 2,' começa a terceira);
s: = ' olha que bonito!';
WRITELN (' ':liv * 2, s);
WRITELN (' ':liv * 2, ' acaba a terceira');
liv: = liv-1;
END; ( * terceira * )
```

```
BEGIN ( * main * )
  s: = 'programa principal';
liv: = 0
WRITELN (s);
primeira; WRITELN (s);
segunda; WRITELN (s);
terceira; WRITELN (s);
END.
```

Dado que neste livro, dedicado aos princípios básicos, não existe espaço para um tratamento sistemático completo, vamos explicando certas regras à medida que vão aparecendo. WRITELN é o equivalente ao PRINT do BASIC; conclui com um RETURN. Também é utilizada a instrução WRITE, que funciona da mesma forma, mas sem o Return final. No Pascal UCSD dispomos do tipo STRING (cadeia), ausente no Pascal standard. Uma constante de cadeia é encerrada entre apóstrofos (') em lugar das aspas do BASIC (entre parêntesis). Os objetos a imprimir já são situados sejam variáveis ou constantes. Os dois pontos seguidos de uma expressão indicam o comprimento do campo, no qual vai justificado pela esquerda o elemento que se precede: assim WRITELN (' ': Liv * 2, 'começa a primeira ') fará preceder a expressão "Começa a Primeira", de tantos pares de espaços (' ') quantos o indique Liv*2. Isto servirá para fazer evidente, com o oportuno escalonamento, o nível em que nos encontramos e, a tal fim, a variável Liv (note que é global) será incrementada e diminuída à entrada e saída dos três procedimentos. Chegados a este ponto, convidamos ao leitor a imaginar o que faz o programa.

Se o que pensou coincide com o que segue, felicidades! acertou!

```
Programa principal
   Começa a primeira
   Acaba a primeira
Programa principal
   Começa a segunda
   Cuco, sou a segunda
      Começa a primeira
      Programa principal
   Acaba a primeira
   Acaba a segunda
Programa principal
   Adiante terceira
```

Olha que bonito
Acaba a terceira
Olha que bonito

Seguramente, inclusive quem havia seguido passo a passo e escrupulosamente o programa, terá se encontrado com alguma surpresa. A razão desta estranha aparência reside no fato de que o "s" é global e atua no main como em todos os procedimentos exceto na Segunda, onde temos com o mesmo nome uma variável local. Portanto, a cadeia global "s" permanece inativa (ignorada e "em suspensão") dentro de Segunda, e volta a funcionar enquanto entra em um bloco no qual "s" não é local. Pode ser visto inclusive que quando chamamos desde Segunda a Primeira o valor de "s" que é impresso não é o esperado "Cuco, sou a segunda!", mas "Programa principal", e ocorre isto porque em Primeira "s" volta a ser a global, desativada tão somente em Segunda.

É muito importante dominar por completo este funcionamento: corresponde a um mecanismo preciso do compilador que, quando chega a uma instrução, procura a variável em primeiro lugar na lista das variáveis locais; se encontra a que está em jogo na instrução, se dá por satisfeito, e somente em caso contrário olha nos módulos de nível superior aos que pertence (não nos demais) e, em último extremo, na lista de variáveis globais. Espero que já esteja tudo claro, especialmente o por que com a última instrução do main é escrito 'Olha que bonito' (o procedimento Terceira não tem "s" como variável local e, portanto, atua sobre a "s" global).

A ocasião é propícia para prevenir possíveis críticas por parte dos BASICalistas insensíveis, convidando, sobretudo aos principiantes, a não exagerar com os refinamentos da programação em Pascal, para evitar mal-entendidos. Em concreto, isto pode ser "traduzido" como:

- não usar demasiadamente níveis, adotando-os somente ali onde são lógicos e funcionais;
- distinguir ao máximo (e em caso de dúvida convém recorrer a nomes distintos...) entre variáveis locais e globais.

Outros elementos do mosaico: as construções

Chegou o momento de tratar das estruturas de controle (reler os capítulos 1 e 2 pode ser oportuno).

Na Figura 1 estão desenhados os elementos sintáticos do ciclo

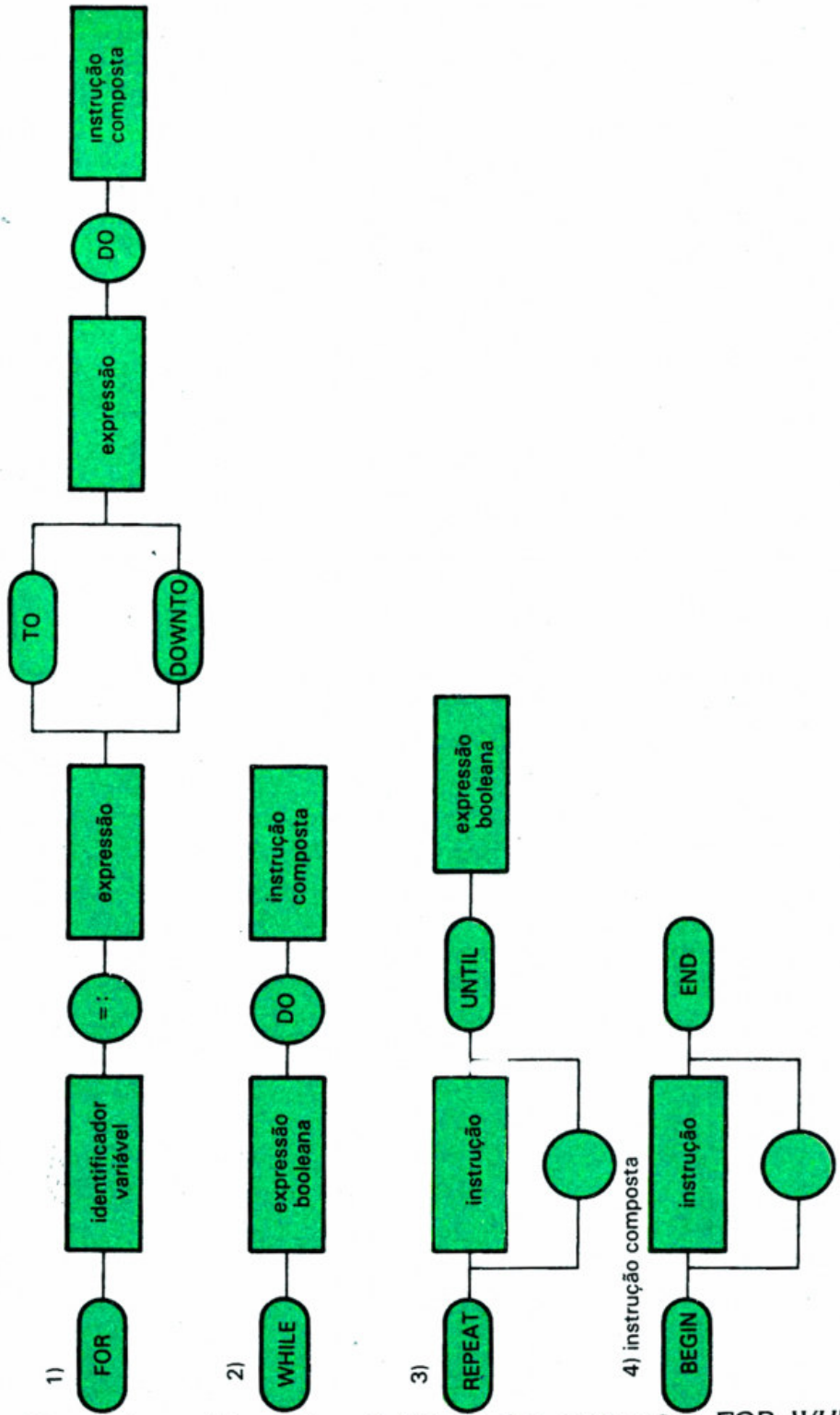


Figura 1 — Elementos sintáticos das contruções FOR WHILE e REPEAT.

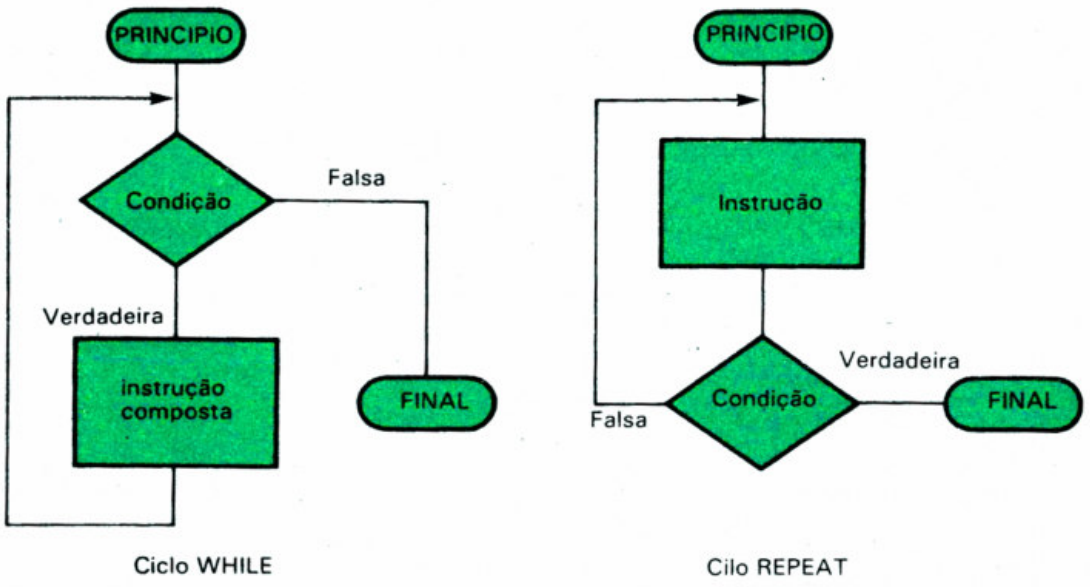


Figura 2 — Diagrama de fluxo das estruturas cíclicas WHILE e REPEAT em Pascal. Recordem que com o primeiro se sai quando a condição é falsa e, com o segundo, quando a condição é verdadeira.

FOR (já visto) e dos famosos WHILE e REPEAT. Como podemos apreciar (diagrama 4), por Instrução Composta é entendido uma série de instruções elementares (chamadas em inglês "statement") separadas pelo habitual ";" e delimitadas pelos familiares BEGIN e END. A estrutura REPEAT, ao contrário, prescinde deles (isto passa em Pascal; em outras linguagens estruturadas existe maior uniformidade a respeito). De fato, o pai do Pascal, Niklaus Wirth, decidiu que as palavras REPEAT e UNTIL constituiam bons delimitadores "naturais".

Na Figura 2 temos os organogramas das duas construções WHILE e REPEAT. São equivalentes e sempre podem ser substituídas uma pela outra.

Apareceu na figura 1 um termo (expressão booleana) que é oportuno esclarecer. Na gíria é chamada booleana a um tipo standard de variável capaz de tomar somente dois valores; TRUE (verdadeiro) e FALSE (falso), palavras evidentemente reservadíssimas. O adjetivo "booleano" é usado em honra a George Boole, inventor da Álgebra do mesmo nome. Este adjetivo caracteriza em Pascal uma variável lógica, da seguinte forma:

VAR lapalis: BOOLEAN;

e depois disto, uma possível atribuição desta variável seria:

lapalis: = (a > b) OR NOT (a > b)

Pergunta, que valor (booleano) assumirá lapalis segundo a condição anterior? É evidente que TRUE, independentemente dos valores assumidos por a e b; ou é certa uma condição ou a outra, não existe escapatória. As "regras do jogo" da expressão booleana, em Pascal, são as mesmas que no BASIC, isto é:

- as condições elementares estão expressas por comparações (com simbologia idêntica: > , < , = , > = , <= , < >);

As comparações devem ser feitas entre tipos homogêneos:

- o símbolo = em Pascal não pode ser confundido com o de atribuição, que, como vimos, está precedido por dois pontos;
- podem ser usados os operadores lógicos AND, OR e NOT.

Vejamos agora um par de exemplos com a tartaruga e, em parte, com as cadeias do UCSD Pascal. Este é o primeiro:

```
PROGRAM whilegrf;
VAR comp, ang, incr: INTEGER;
  resp: CHAR;
PROCEDURE iniscrn;
 BEGIN
  CLEARSCREEN; PENCOLOR (WHITE)
 END;
PROCEDURE proxlin;
BEGIN
 MOVE (comp); TURN (ang);
 comp: = comp + INCR
END;
BEGIN; ( * main * )
 inicrn;
 WHILE (resp  < > 'F') AND (resp < > 'f') DO
  READLN (ang); READLN (incr);
```

```
    comp: =5;
    WHILE comp < =150 DO proxlin;
  READ (resp);
  iniscrn;
END; ( * WHILE maior * )
END.
```

Este programa serve para traçar um típico desenho dos gráficos de Tartaruga, como o da Figura 3 (corresponde a um ângulo = 89 graus). Variando os parâmetros "ang" e "incr" introduzidos poderão ser vistos uma infinidade de resultados (são interessantes ângulos como: 59, 60, 61, 72, 73, 119, 120, 121, 135, 144, 154, etc.). Quem dispor de um computador e quiser divertir-se, pode fazer o seguinte: antes de tudo, inserir depois da linha PROGRAM o indispensável USES TURTLEGRAPHICS e substituir CLEARSCREEN por INITTURTLE. O main está composto por dois ciclos WHILE aninhados um em outro. O exterior toma os parâ-

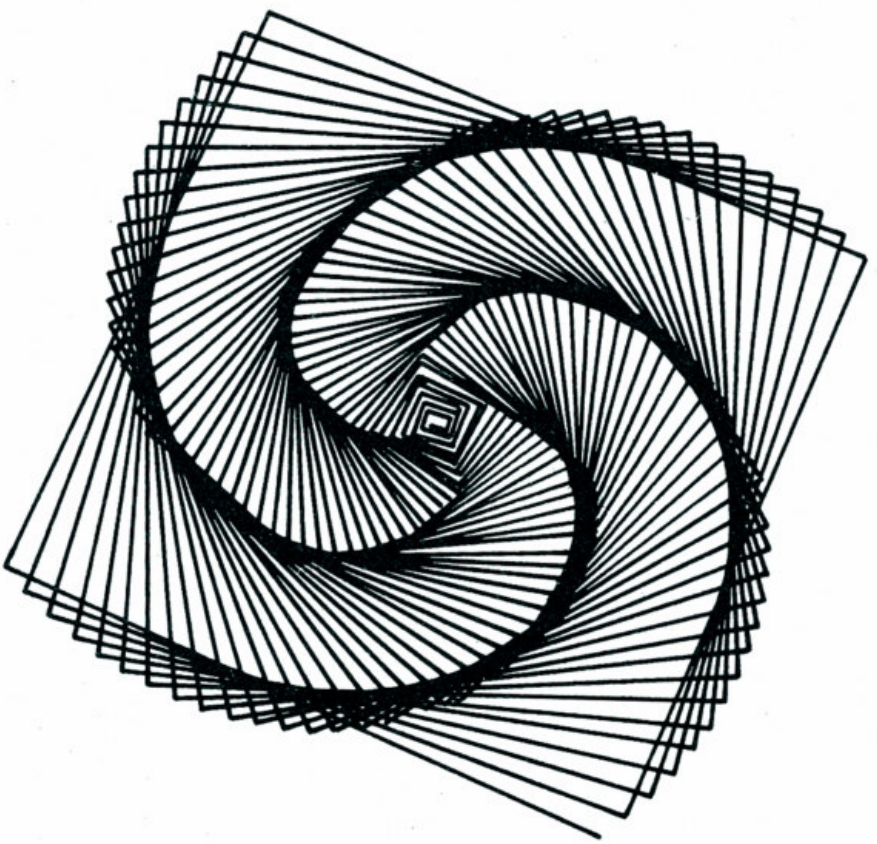


Figura 3 — Um clássico desenho da tartaruga.

metros "ang" e "incr" e, ao final, "resp". Este é um dado de tipo CHAR, isto é, caracter, pelo que READ equivale ao GET de BASIC. Como fica evidente (e é esta a razão tanto de WHILE como de REPEAT) o processamento é repetido "enquanto" (while) a resposta ("resp") for distinta de fim (F maiúsculo ou minúsculo). Uma pergunta: a condição (resp <> 'F') AND (resp <> 'f') é correta, mas substituindo AND por OR, o ciclo não acaba nunca. Saberia dizer-nos por que? Se não for pressionado F nem f (um simples RETURN, por exemplo) é chamado ao procedimento Iniscr de limpeza de tela. Ao contrário, o loop mais interno executa repetidamente Proxlin, que consiste em traçar linhas de comprimento crescente (em um valor incr) seguidas de um giro de "ang" graus até que "comp" supere 150.

Tome nota de que o WHILE mais interno não requereu os delimitadores BEGIN e END porque, como costuma ocorrer em outras linguagens, compreende uma só instrução Proxlin.

Uma última observação: o espartano do diálogo homem-máquina é também devido ao fato de que, introduzindo a tartaruga, se vai à página gráfica, pelo que instruções tipo WRITELN ("Dê-me o ângulo") não aparecem na tela de muitos computadores pessoais, a menos que sejam introduzidas instruções de restabelecimento de modo texto/modo gráfico (TEXT MODE e GRAFMODE no Pascal UCSD do Apple) que complicariam desnecessariamente o exemplo.

REPEAT, IF THEN/ELSE e equivalências

Ao ilustrar agora a construção REPEAT, aproveitaremos também para introduzir a estrutura IF/ELSE, muito conhecida e presente em muitos BASIC.

A idéia original do algorítmo base do seguinte programa é remontada ao célebre Martin Gardner, da revista "Scientific American", e consiste em transformar uma cadeia de caracteres 'S' e 'D' em figuras obtidas fazendo girar a Tartaruga + 90 ou - 90 graus em cada um dos casos. Para complicar as coisas acrescentamos a possibilidade de que o usuário peça a repetição do traçado e de que escolha ângulos diferentes de 90°. Os desenhos obtidos são do estilo dos mostrados na Figura 4: em a) com a limitação de Gardner e em b) com a flexível extensão a ângulos quaisquer.

```
PROGRAM figuras;
VAR sec: STRING; car: CHAR;
```

```
            dim,ang,i: INTEGER;
BEGIN
  WRITE (' dimensão? '); READLN (dia);
  WRITE (' angulo? '); READLN (ang);
  WRITE (' sequencia? '); READLN (sec);
  CLEARSCREEN; PENCOLOR (WHITE);
REPEAT
    i: = 1;
    REPEAT
      MOVE (dim  i);
      IF SEC (i) = 'S' THEN TURN (ang)
           ELSE TURN (-ang)
        i: = i + 1;
    UNTIL i   LENGTH (sec);
    READ (car) (   NOTA: Return acaba a sessão de desenho   )
  UNTIL car = CHR (13)
END.
```

Trata-se de um desses casos nos quais o programa fala por si mesmo, é quase auto-explicativo: até que (until) o usuário não aperte a tecla Return (de código ASCII 13) a tartaruga repete o traçado (recomeçando com i: = 1) da figura (Gardner utilizava o termo "espirolateral"). Se não acaba com a tartaruga na mesma posição que ao início, cada uma das repetições será convertida, em realidade, em fragmento de outra figura maior. A figura "nasce" da interpretação da cadeia *sec*, cujos caracteres são examinados, um a um, com a instrução sec (i), determinando giros à direita ou esquerda e repetindo-os até...; bom, não vamos repetir o que, no fundo, se disse e repete no programa mesmo! Faremos somente umas observações formais sobre os elementos pascalianos introduzidos às escondidas:

- o tipo CHAR é, obviamente, um caracter;
- WRITE ('algo') seguido de READLN (x) equivale a INPUT "algo", x do BASIC, e WRITE, contrariamente a WRITELN, não manda a princípio de linha o cursor;
- sec (i) funciona em uma cadeia como o MID$ (SEC$, I,1) dando-nos o iésimo caracter da cadeia;
- também LENGTH (cadeia) e CHR (n) funcionam do mesmo modo que as funções análogas em BASIC (LEN e CHR$) dão o comprimento da cadeia e o caracter correspondente ao código ASCIL n.

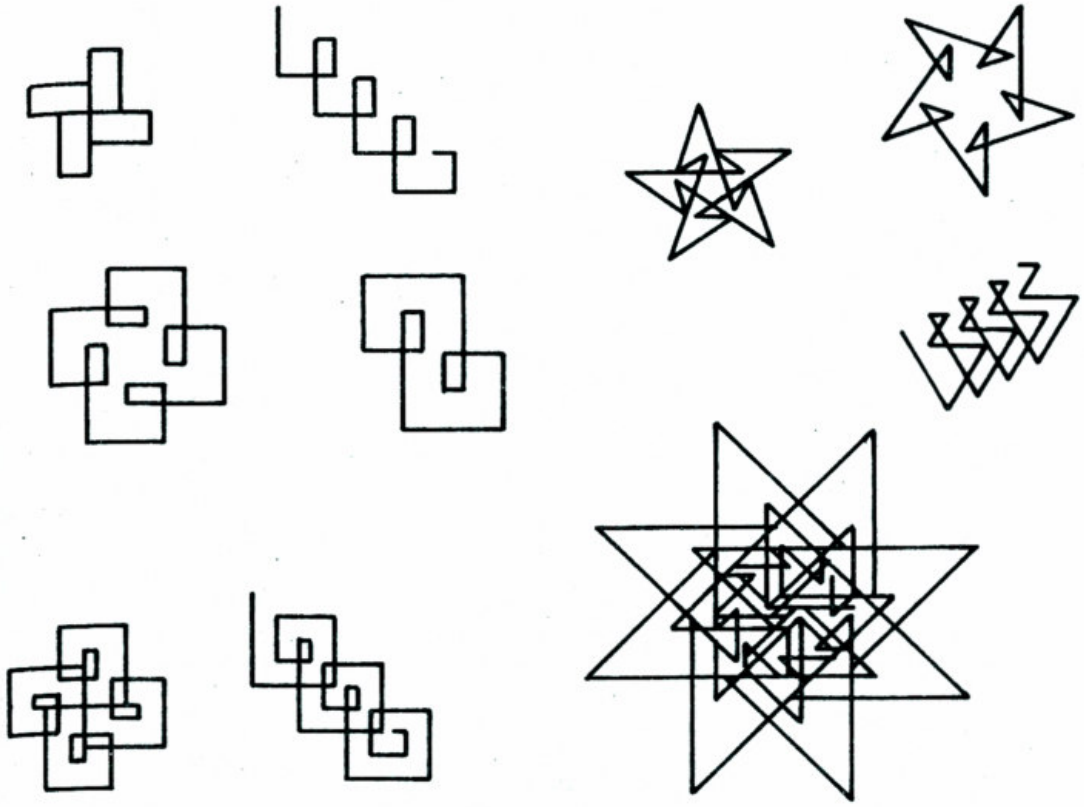


Figura 4 — Figuras obtidas partindo de um algorítmo idealizado por Martin Gardner. Em a) é seguida sua idéia primitiva (ângulo de ± 90 graus); em b) são representados exemplos da variante de Bowles (qualquer tipo de ângulo).

Vejamos agora o problema da equivalência entre REPEAT e WHILE, que já sabemos garantida pelo teorema de Jacopini & sócio.

Em casos como os dos que mostramos em seguida (geração dos quadrados de 1 a 100...) é banal:

```
x: = 0;
WHILE y(100 DO

BEGIN
x: = x + 1;
y: = x * x;
....
END;
```

```
x: = 0;

REPEAT
x: = x + 1;
y: = x * x;
....
UNTIL y > 100
```

Observe que a condição y ≤ 100 do WHILE, foi trocada por y = 100 no REPEAT. Isto acontece porque o teste foi feito em um caso ao princípio e em outro ao final, e pela semântica diferente de WHILE e REPEAT (voltem a estudar a Figura 2).

Podem acontecer, no entanto, situações de relativa ambiguidade.

Vamos supor que temos:

```
READLN (x,y)
WHILE x > =y DO
 BEGIN
  elab (x,y)
....
 END;
```

onde Elab (x,y) seja um procedimento que manipula "x" e "y", transformando-os de forma imprevisível (Elab poderia conter, por exemplo, a leitura de um novo par x,y), de forma que não sabemos *a priori* quando será conseguida a condição prevista. Agora então, o ciclo WHILE, com o teste ao princípio, poderia não realizar-se nem sequer uma vez; assim, uma transposição precipitada ao ciclo REPEAT, que tem o teste ao final, poderia tornar-se errônea. A correta será a seguinte.

```
        READLN (x,y)
IF x   =y THEN REPEAT
        elab (x,y);
...
UNTIL X < Y
```

Não faz falta, contudo, ser muito purista para encontrar esta solução pouco elegante. Vejamos um exemplo totalmente formal do caso contrário.

```
    REPEAT
I1;I2;...;In
UNTIL <CONDIC>
```

Onde I1...In são instruções genéricas.

Para fazer algo perfeitamente equivalente com WHILE teremos:

```
I1;I2;...;In
WHILE <CONDIC> DO
```

```
BEGIN
I1;I2;...;In
END;
```

Note que < condic > é, geralmente, a negação de < condic > . Agora então, se as instruções I1...In formam um conjunto abundante (com bastante código), sua duplicação resultará, pelo menos, antieconômico. Poderia ser evitado adotando uma variável auxiliar booleana (como pode ver, voltamos à problemática do capítulo 2):

```
sw: TRUE;
WHILE < CONDIC > OR sw DO
BEGIN
I1;12;...;In;
sw: = FALSE
END;
```

No primeiro ciclo, o switch sw (desviador), ao ser Verdadeiro, força a primeira execução. Ao acabar esta, o damos a SW o valor FALSE (falso), com o qual, que sejam realizados ou não mais ciclos, dependerá somente de < condic > .

Está claro que, segundo as ocasiões, umas vezes REPEAT será mais cômodo que WHILE e outras ocorrerá ao contrário. Também a presença do loop FOR é útil, ainda que as limitações já vistas nos obriguem, às vezes, a recorrer a WHILE ou REPEAT para conseguir um incremento/diminuição não unitário. Existe também alguns truques para conseguir isto sem usar as mencionadas estruturas: por exemplo, um equivalente a um BASIC do tipo:

```
FOR X = 0 .1 TO 10 STEP 0.1
```

pode ser feito em Pascal assim:

```
x: = 0
FOR i: = 1 TO 100 DO
  BEGIN
x: = x + 0.1
....
END;
```

Funções, procedimentos e suas relações

Como já comentamos previamente, em Pascal, além dos pro-

cedimentos, existem as FUNCTION. Sua finalidade, à diferença dos primeiros, é restituir ao programa (ou subprograma) que as chama um valor evocável através do nome da própria função.

Como de costume, vamos dar em seguida um par de exemplos:

```
PROGRAM cilindro;
VAR volume, r,h: REAL;
FUNCTION areacirc (r:REAL):REAL;
CONST pi = 3.14159265;
  BEGIN
    areacirc: = r * r * pi
  END;
BEGIN ( * main * )
  READLN (r);
  WHILE r > = 0 DO;
  BEGIN
    READLN (h);
    volume : = areacirc(r) * h;
    WRITE (' o cilindro de raio ',r,'e altura ',h);
    WRITELN (' tem um volume ',volume );
READLN(r)
  END;
  END.
```

Aproveitamos o exemplo para introduzir outro tipo de dado, o REAL, cujas características podem encontrar em anteriores volumes da B.B.I. (são números reais, por exemplo, 10, 3.14, 8.1697, etc.) Também introduzimos outro elemento sintático: CONST, que declara como constante o nome que o segue (em nosso caso "pi") e fixa seu valor (3.14159265), "pi" será desconhecida fora da FUNCTION areacirc. Dê-se conta de que se utiliza o sinal = em lugar de : = porquanto se tem identidade mas não atribuição.

Este exemplo é tão simples que seu funcionamento para o cálculo de volumes de cilindros é deixado inteiramente à compreensão do leitor. Em essência nos serviu para introduzir a sintaxe de FUNCTION e para fazer notar que nela, à diferença do que acontece com um procedimento, o nome da função é conduzido, em uma expressão, como uma variável, que pode utilizar de forma direta o programa que a chama. Isto poderia inclusive realizar-se com um WRITELN; no caso visto, teria sido possível fazer o seguinte:

```
WRITE ('o cilindro de raio', r, 'e altura', h);
```

WRITELN ('tem um volume', areacirc * h);

Em outro caso, supondo que Prisma, Pirâmide e Esfera sejam outras FUNCTION, o volume total de um composto sólido seria dado por uma expressão deste tipo:

voltot: = prisma (b1, b2, h1) + pirâmide (b3, b4, h2) + esfera (r)

formado por elas (admitindo que a esfera é sustentada na ponta da pirâmide).

Muitas vezes é possível obter com os PROCEDURES algo muito similar a FUNCTION. Não é um jogo de palavras; por exemplo, em nosso caso teríamos podido recorrer à variante seguinte:

PROCEDURE calcarea (rag: REAL; VAR areacirc: REAL)

Mas cuidado: depois tem que modificar também a declaração das VAR no main. Pode ser feito assim:

PROGRAM cilindro;
VAR volume, areacirc, r, h: REAL;

Desta forma conseguimos que Calcarea devolva, dependendo somente do parâmetro raio, o resultado Areacirc. Então, fica todo o demais como antes? Não exatamente. Tal e como dissemos antes, desta forma podemos conseguir "algo" parecido a uma FUNCTION, mas seu comportamento e manejo seriam distintos. Assim será necessária também uma modificação posterior.

...
READLN (h)
calcarea (r,areacirc); volume; = areacirc*h;

Seguramente a maioria de vocês já esperavam algo parecido, pelo menos os mais atentos, pois recordaram que um procedimento deve ser chamado explicitamente, com seus parâmetros, para ser executado. O que pode lhes pegar de surpresa é a presença de Areacirc junto ao parâmetro "r". De fato, Areacirc é, ao mesmo tempo, um parâmetro e um resultado. Para colocar um pouco de ordem nesta delicada matéria devemos esclarecer que, nos procedimentos, existem dois tipos possíveis de parâmetros:

- parâmetros chamados "como valor",

- parâmetros chamados "como referência" (ou "por situação").

Também são chamados, respectivamente, parâmetros-valor e parâmetros-variáveis.

Os primeiros são como raio, e os segundos são do tipo de Areacirc e são distinguidos dos primeiros por estar precedidos em sua definição da palavra VAR. A diferença essencial entre eles é que os primeiros somente são argumentos, enquanto que os segundos podem ser também funções.
Mas ambos são parâmetros e, portanto:

- tem que passá-los ao procedimento.
- podem ter nomes distintos no main e no procedimento.

Para entendermos melhor, vamos supor que o exemplo precedente seja um pouco mais complicado e tem que ser calculado os volumes de três cilindros distintos de raios r1, r2 e r3, e alturas h1, h2 e h3. Incluímos nas VAR do main não somente estas seis novas variáveis, mas também area 1, area 2 e area 3.

Tente, sabendo isto, escrever a chamada correta a Calcarea para calcular o volume do segundo cilindro (supomos que "volume" é uma variável de trabalho usada para os três cilindros).
Seria:

```
calcarea (r2, area2); volume: = area*h2;
```

Isto é, que a escolha de Areacirc como nome comum para o main e o procedimento era legítima (não produzia moléstias), mas, certamente, não vinculante.

Vejamos um caso simples, mas interessante; para esclarecer outros quantos conceitos;

```
PROGRAM exemplo;
VAR x,y,z: INTEGER
PROCEDURE contador (VAR n: INTEGER; incr:INTEGER);
   BEGIN
      n: = n + incr
   END;
BEGIN ( * main * )
  x: = 0;y: = 2;z: = 1;
WHILE x < 12 DO
      BEGIN
```

```
        contador (x,z);
        contador (y,x);
        contador (z,y);
      END;
END.
```

Depois de quantas voltas se sai do loop WHILE e com que valores de "x", "y", e "z"? Se o fizer verá que após duas voltas, e com valores 17, 25 e 37 (basta um pouco de paciência para comprová-lo). À metade do caminho pode acontecer, por exemplo, que "x", "y", e "z" valem 1, 3 e 4, depois do qual, ao chamar a Contador (x,y) a "x" se acrescenta "z" "passada" como incr a Contador, que devolve um "x" modificado igual a 1 + z = 5.

Em resumo: em uma PROCEDURE os parâmetros usados como referência são empregados para facilitar os resultados obtidos na PROCEDURE ao programa ou subprograma que realizou a chamada e que determinou os parâmetros de valor.

Outra vantagem dos procedimentos com respeito das funções é que permitem a realização de funções com saídas múltiplas. Somente será necessário definir mais parâmetros de tipo VAR. Por exemplo (e sem mais comentários):

```
PROCEDURE multfunc (x,y,z: REAL; VAR f1, f2: REAL);
```

Existe uma última diferença entre FUNCTION e PROCEDURE, mas é um tema delicado que o deixaremos para mais tarde.

Outras aplicações de FUNCTION e IF THEN/ELSE aninhadas

Apesar do exposto anteriormente, muitas vezes podem ser obtidas coisas elegantes com uma FUNCTION, além de instrutivas. Aqui uma:

```
PROGRAM perguntas;
VAR trabalho, chuva: BOOLEAN;
  FUNCTION se: BOOLEAN;
VAR resp: CHAR;
BEGIN
   READLN (resp);
   IF resp = 'S' OR resp = 's' THEN
     SE: = TRUE
   ELSE
     se: = FALSE
```

```
  END; (* se *)
BEGIN (* main *)
  WRITELN (' desperta!'; WRITELN;
  WRITE (' casado ha mais de 5 anos?');
  IF NOT s THEN WRITELN ('beija tua mulher');
  WRITE ('dia laborável?'); trabalho: = sim;
  WRITE ('chove fora?'); chuva: = sim;
  (* Poderiam ser adicionadas muito mais perguntas *)
   IF trabalho THEN
      IF chuva THEN
        WRITELN ('ao trabalho de carro')
      ELSE
        WRITELN ('ao trabalho a pe; fica bem!')
   ELSE
        IF CHUVA THEN
              WRITELN ('fica em casa')
      ELSE
              WRITELN ('vem jogar tenis')
END.
```

A curiosa FUNCTION se (sem parâmetro desta vez; não faz falta) permite recolher a resposta e transferir seu resultado (lógico) a uma expressão IF. Isto ocorre com a pergunta inicial, que é solucionada em seguida com beijo ou não beijo. Ao contrário, nas perguntas seguintes se recorreu às variáveis booleanas temporais (trabalho e chuva) para permitir, mais adiante, o varrer casuístico.

Esse programa nos oferece a oportunidade de realizar alguns esclarecimentos úteis sobre a estrutura IF THEN/ELSE quando aparece aninhada. Dado que a primeira regra é o sentido comum, tem que ter em conta que:

- tanto IF como ELSE fazem de separadores (como BEGIN, END ou melhor, como REPEAT, UNTIL);
- nos casos de instruções múltiplas, as inerentes a uma mesma IF estariam encerradas entre dois separadores; senão haveria problemas (erros lógicos ou mal-entendidos com o compilador).

Um exemplo clássico de incompreensão entre a intenção do programador e a interpretação que dele realiza o compilador é apresentada com a chamada IF "abreviada", ou seja, sem ELSE.

Vamos supor que no nosso programa, por algum estranho

motivo se quer eliminar a resposta "ao trabalho a pé base" (porque o escritório está demasiadamente longe, por exemplo). O principiante pensaria que é muito fácil e o modificaria assim:

```
IF trabalho THEN
  IF chuva THEN
      WRITELN ('ao trabalho em carro')
ELSE'
   IF CHUVA THEN
      WRITELN ('fica em casa')
   ELSE
      WRITELN ('vem jogar tênis')
END.
```

Agora então, indicando as duas respostas possíveis com S e N, que saída dará agora o programa nos quatro casos possíveis? Tentem adivinhá-lo e comparem-o com o que segue:

SS ——— 'ao trabalho em carro'
SN ——— 'vem jogar tênis'
NS e NN ——— NENHUMA MENSAGEM!!!

Surpresos? No segundo caso a mensagem é imoral (ir ao tênis em dia de trabalho porque tem sol), enquanto que nos dois últimos não tem resposta. O fato é que o compilador não faz o mínimo caso dos escalonamentos que utilizemos e associa cada ELSE à ùltima IF que encontra. E se tratarmos de IF sem ELSE? Não fica escandalizado e executa somente a primeira instrução que encontra depois da cadeia IF; se tratarmos precisamente da IF "abreviada" (utilizada no caso do beijo à mulher). Vejamos, para satisfazer a todos, os diferentes casos detalhadamente:

- SS. Funcionam as duas primeiras IF, enquanto que a primeira ELSE provoca o salto até END:
- SN. A primeira ELSE é tomada, também aqui, como alternativa à chuva, depois da qual a nova IF chuva da vida à outra ELSE, daqui a resposta.
- NS e NN. Não existe nenhuma mensagem, porque não verificando-se trabalho, salta diretamente a END. Se refletir descobrirá fácilmente que tudo depende de que o compilador associa, em todos os casos, a primeira ELSE, a segunda IF.

Como poderíamos remediá-lo? Existe duas possíveis soluções para problemas deste tipo. A primeira consiste em recorrer a expressões booleanas:

```
IF trabalho AND chuva THEN
      WRITELN ('ao trabalho de carro')
IF NOT trabalho AND chuva THEN
      WRITELN ('fica em casa')
IF NOT trabalho AND NOT chuva THEN
      WRITELN (' vem  jogar tenis')
END.
```

A segunda solução consiste em encerrar entre um BEGIN e um END a segunda IF:

```
IF trabalho THEN
  BEGIN
    IF chuva THEN
      WRITELN ('ao trabalho de carro')
  END;
ELSE
    IF CHUVA THEN
      WRITELN ('fica em casa')
ELSE
WRITELN (' vem  jogar tenis')
```

Desta maneira se "convence" o compilador para que associe a primeira ELSE à primeira IF.

Finalmente, vamos supor que a alguém ocorra consertá-lo colocando um ";" depois de WRITELN ('ao trabalho em carro')... isto é o pior que se poderia fazer: o compilador ficaria enfadado porque não saldariam as contas das IF e das ELSE.

Segundo dizem os críticos do Pascal, estes contratempos são derivados da ausência de terminadores ENDIF. Existe uma solução possível, ainda que não muito limpa, que consiste em fechar cada IF tipo abreviada com um ELSE (de fato, se ELSE não está seguida de nenhuma instrução, o compilador não fica escandalizado). Em nosso caso, o truque consiste em tirar o BEGIN e substituir seu END por ELSE.

Finalmente, a propósito de BEGIN e END, temos que recordar que se desejarmos executar várias instruções em seguida de um IF ou de um ELSE, não poderiam ser esquecidas, em absoluto. De fato, aqui não temos o que normalmente ocorre em BASIC, onde se em uma linha se tem

```
100 IF < CONDIC > THEN instrl:instr2:instr3:...
```

todas as instruções da linha são executadas ou saltadas.

O Pascal é mais rigoroso, e se nos esquecemos BEGIN e END assume que somente deve executar a primeira instrução.

A estrutura CASE

O Pascal proporciona a estrutura CASE para simplificar e esclarecer as situações de escolhas múltiplas. É uma construção comodíssima, sempre que a cada eleição corresponda uma só instrução. De todo modo, também pode ser aplicada em outras situações, definindo à parte procedimentos adequados, ou então recorrendo ao habitual dueto BEGIN-END.

O exemplo que veremos com a tartaruga serve para movê-la na tela pressionando algumas teclas. As possibilidades previstas são: movimentos para frente ou para trás de 2 passos de comprimento e rotações absolutas segunda a rosa dos ventos, ou seja, 8 ângulos que vão de 45 em 45 graus. É evidente que se aqui não tivesse havido estrutura CASE teria sido necessário inventá-la ou então recorrer a uma tabela, mas esta seria uma solução de menor clareza semântica. Para as duas primeiras ações adotamos as teclas A e S, para as outras, as teclas que geralmente estão dispostas ao longo das 8 direções indicadas. Teremos:

```
PROGRAM tortpasseio;
VAR caract: CHAR;
BEGIN (* main *)
  PENCOLOR (WHITE);
  READ (caract);
  WHILE caract<>CHR(13) DO
  (* se não e return *)
     BEGIN
       CASE caract OF
         'A': MOVE (2);
         'S': MOVE (-2);
         'J': TURNTO(0);
         'M': TURNTO(-45);
         'N': TURNTO(-90);
         'B': TURNTO(-135);
         'H': TURNTO(180);
         'Y': TURNTO(135);
```

```
        'U': TURNTO(90);
        'I': TURNTO(45);
      END: ( * final CASE * )
      READ (caract);
        END; ( * final de WHILE * )
END.
```

CASE é um exemplo tão eloqüente que não merece nenhum comentário. Confirmar somente que a construção é do tipo:

CASE ‹ expressão › OF ‹ lista de casos › END

onde a lista de casos corresponde aos possíveis resultados da expressão. É interessante a possibilidade de reagrupar juntos, separados por vírgulas, casos que prevêm uma solução idêntica. No programa precedente, por exemplo, poderiam ser equiparadas maiúsculas e minúsculas:

```
CASE caract OF
'A', 'a': MOVE (2);
```

No Pascal standard falta a opção ELSE, associada diretamente à CASE, que permite incluir, em uma única ação, os casos não incluídos na lista, inclusive aninhando-o todo. Não se nota muito sua ausência, pois geralmente costuma ser substituído por uma IF/ELSE por cima da CASE.

Para quem nos segue com a devida atenção, fazemos a seguinte pergunta: o que ocorre se, no programa precedente, foi pressionada uma letra que não estava compreendida na lista CASE?

Deixamos no ar a fácil resposta, que confirmaremos ao acabar o capítulo.

O problemático GOTO

Terminamos a relação das estruturas de controle com o discutido GOTO, eliminado ignominiosamente pelo estruturalistas até a morte, apesar de que realmente o GOTO não é uma estrutura, mas um construtor de estruturas. Wirth foi benevolente e somente relegou o GOTO a cidadão de segunda, de forma que todos os manuais de Pascal, seguindo seu exemplo, o tratam ao final e "com pressa".

O GOTO é utilizado em casos extremos e complicados, por exemplo, onde as estruturas disponíveis tornam-se excessivas a duplicação de códigos ou o abuso de booleanas. Para fazer desistir do uso do GOTO, Wirth pensou em introduzir o emprego, um pouco molesto, de etiquetas (label) de destino; são compostas somente de cifras, mas são tipos e NÃO manipuláveis de nenhuma forma e têm que ser declaradas as primeiras no módulo ao qual pertençam. Explicamos com um exemplo:

```
PROGRAM p;
LABEL 1;
.....
   PROCEDURE A;
   LABEL 2;
   .....
   BEGIN
.....
GOTO 2
.....
GOTO 1
.....
2: WRITELN ('pequeno erro ...')
.....
END; (* procedure a *)
.....
1: WRITLEN ('erro trágico!')
.....
END.
```

GOTO, obviamente, é associável a IF, obtendo-se a familiar combinação: IF < condic > THEN GOTO...

No pequeno exemplo anterior, GOTO é útil para a saída antecipada de um loop, especialmente (mas não somente) quando tiver um erro. Portanto, aqui também é válido o princípio de que o salto de um bloco interno a outro externo é lícito, mas não o contrário.

De todas as formas, uma vez que foi delineado um programa com construções WHILE, REPEAT, etc., que originam módulos talvez complexos mas bem encadeados uns em outros e com um único ponto de entrada e saída, GOTO, além de supérfluo, pode tornar-se complicado para seu manejo. Se a isto acrescentamos efeitos colaterais, sobre os que não podemos deter-nos, mas que aconselham muita precaução, é possível que tenhamos que dizer

adeus ao GOTO.

Acabamos este longo capítulo com dois assuntos. Antes de tudo, a resposta à pergunta sobre CASE: evidentemente, se pulsarmos uma tecla não prevista (a distinta do retorno do carro) a tartaruga não se move.

O segundo assunto é a apresentação das cartas sintáticas das Figuras 5 e 6. A Figura 5 contém todas as construções do Pascal, enquanto que a 6 resume a estrutura de um "bloco" de programa. Recordamos novamente que a sucessão deve respeitar a ordem Etiqueta, Construção, Type, Var, Procedimento e Function, que como truque mnemotécnico, equivaleria a: "Esta Casa Tem Várias Portas Falsas".

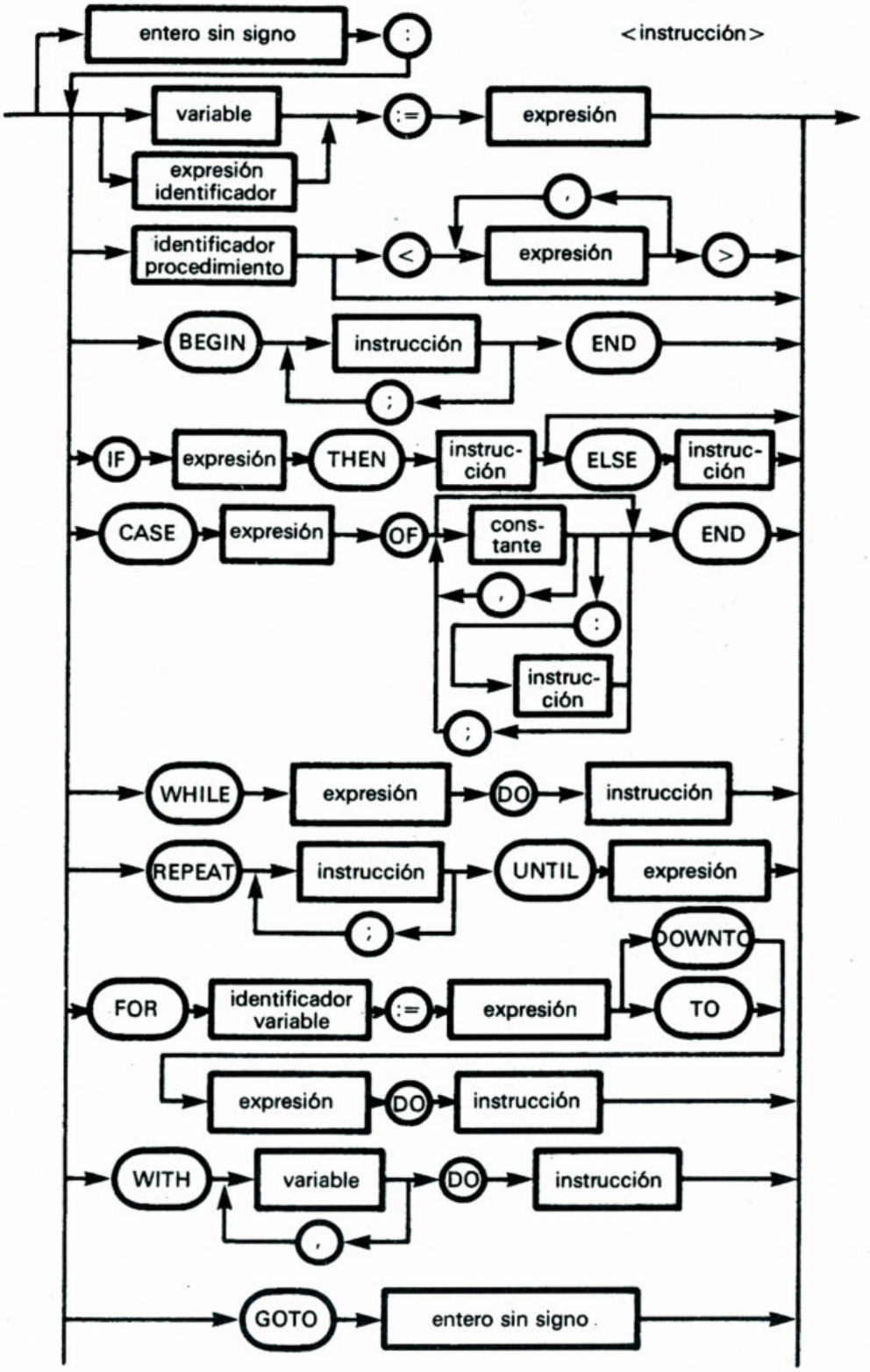


Figura 5.—Sintaxis de las estructuras de un programa Pascal. Para más detalles, le sugerimos consultar con manuales especializados.

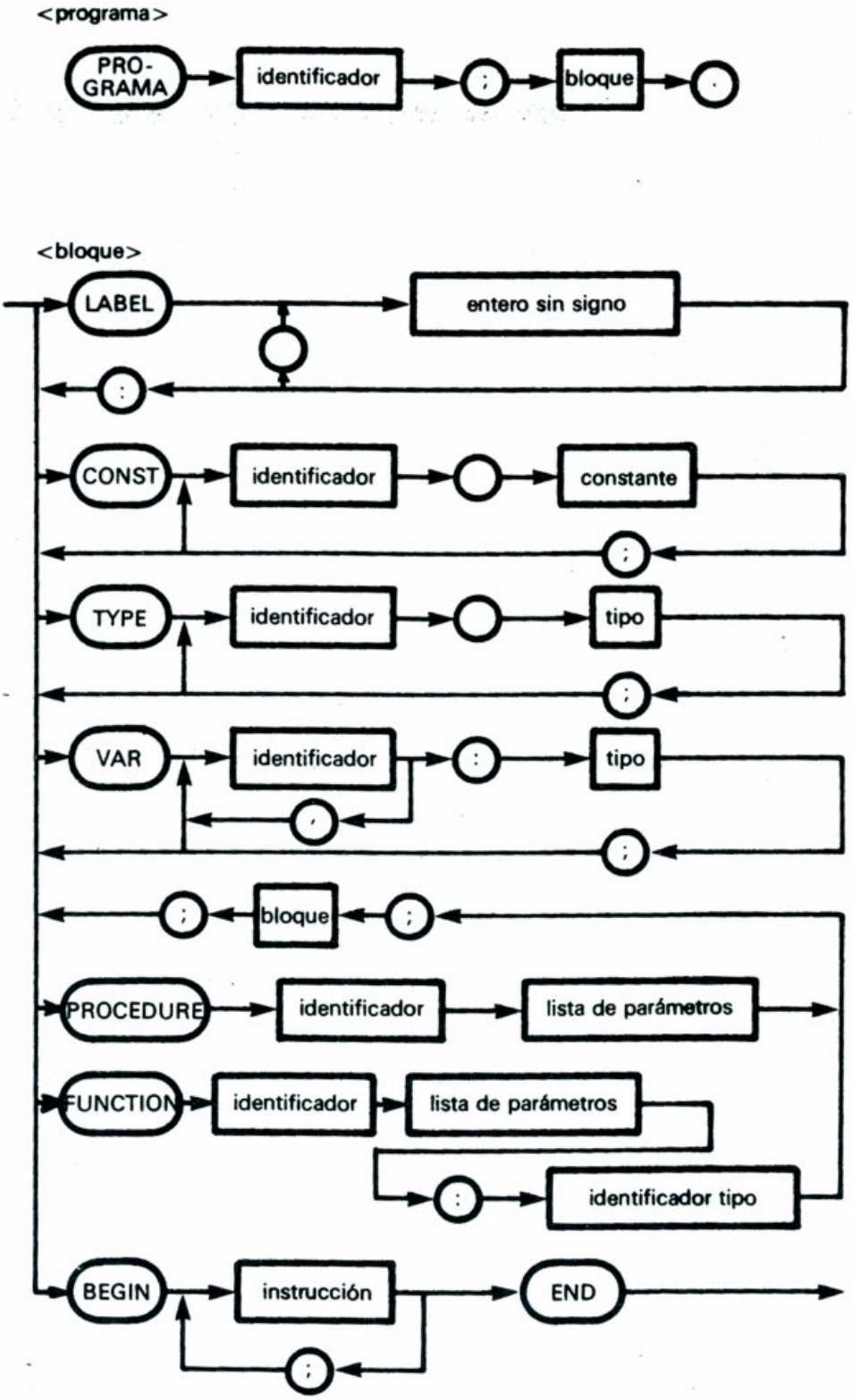


Figura 6.—Sintaxis de la estructura de un bloque de programa en Pascal.

CAPÍTULO V

FUNÇÕES, PROCEDIMENTOS E OS ESTRANHOS ANÉIS DA RECURSIVIDADE

Sejamos sérios...

Antes de defrontar o complexo tema dos tipos de dados que, segundo o que já vimos nos dois capítulos introdutórios estão bastante ligados aos algorítmos, nos propomos ampliar nossos conhecimentos sobre as FUNCTION e aproximar-nos ao mundo da recursividade. Vejamos, pois, um par de exemplos relativos às FUNCTION. O primeiro tem um certo caracter prático. Se refere ao cálculo mensal de IRRF (em um país imaginário) correspondente à base imposta de um assalariado, isto é, à parte do salário bruto que fica depois de haver descontado todos os pagamentos à Segurança Social.
Paciência, pois se trata simplesmente de um exercício. O critério de impostos progressivos prevê distintas faixas:

Faixas para impostos		Tabela de retenção
Até	25000	10%
	33333	13%
	41666	16%
	49999	19%
	62500	22%
	75000	25%
	.....	....

A coluna da direita expressa a porcentagem que é aplicada à base onerada compreendida entre o valor da esquerda e o seguinte. O procedimento que vamos ver é baseado na observação de que ambas as colunas têm valores que se sucedem com regularidade: as porcentagens da tabela crescem em 3%, enquanto que as faixas para impostos crescem à razão de 8.333 até 49.999, e desde aqui são incrementados em 12.500 unidades da moeda daquele país imaginário.

O algorítmo que sugerimos aqui consiste no seguinte:

- aplique 10% sobre a base imposta;
- considere um posterior 3% sobre a diferença entre esta e cada faixa para imposto, até que cheguemos ao que corresponde à base imposta.

Realizadas estas hipóteses, escrever o programa é coisa fácil. Como de costume, animamos aos mais voluntariosos a tentá-lo. Somente uma obrigação: utilizar uma FUNCTION de nome Impbrut. Aqui a solução proposta:

```
    PROGRAM IMPOSTOS:
    VAR SALÁRIO BRUT, RETENC, IMPOSTO, IMP, IMPENET:
INTEGER;
      FUNCTION IMPBRUT (IMPOSTO: INTEGER): INTEGER;
      CONST INICTABELA = 25000; INTERMED = 49999;
      CREC1 = 8333; CREC2 = 12500;
      VAR FAIX.IMP, IMP: INTEGER
     BEGIN
      IMP: = 0
    FAIX.IMP: = INICTABELA; IMPBRUT: = IMPOSTO DIV 10;
      WILE IMPOSTO > FAIX.IMP DO
       BEGIN
        IMPBRUT: = IMPBRUT + TRUNC          (IMPOSTO-
FAIX IMP) * 0.03);
        IF FAIX IMP < INTERMED
        THEN FAIX  IMP: = FAIX IMP + CREC1
        ELSE FAIX  IMP: = FAIX IMP + CREC2
       END ( * WHILE * ) IMPBRUT: = IMP
       END; ( * IMPBRUT * )
     BEGIN ( * MAIN * )
       ...
       READLN(SALARIO BRUT,....);
       ....
```

```
IMP: = SALARIO BRUT-RETENC;
IMP: = IMPBRUT (IMP);
....
END.
```

Obviamente, os "pontinhos" correspondem a "omissões" que cada um pode "rechear" para seu uso particular.

Neste exemplo, ao ter sido mais concreto, se fazem evidentes, precisamente por isto, certas vantagens aparentemente formais do Pascal. É útil, ainda que não indispensável, definir variáveis locais de FUNCTION enquanto servem exclusivamente para o uso interno do cálculo fiscal. Desta forma é evitado qualquer risco de que no main se modifiquem variáveis que NÃO devem ser influenciadas. Este aspecto é decisivo na utilização recursiva de procedimentos e funções que vamos ver agora, e no caso de módulos de "biblioteca" reutilizáveis, como na linguagem Modula 2. Também as constantes inictabela, intermed, crec1 e crec2 permitem uma cômoda atualização no caso de que devam modificar-se posteriormente (por exemplo, para reutilizar a FUNCTION no cálculo do IRRF anual em vez de mensal); é suficiente mudar somente a seção CONST do módulo em lugar de todas as fórmulas afetadas pela mudança (que podem ser muitas no caso de programas importantes). É confirmada assim a maior transparência de uma listagem Pascal com respeito a uma em BASIC. Ainda que não temos dúvidas em admitir nossa afeição pelo popular BASIC e em apreciar seus dotes de imediatismo e simplicidade, temos que proclamar que o Pascal, ainda que freqüentemente necessita maior cuidado e tempo para ser definido e colocado "em ponto", oferece ao final alguns resultados autodocumentados, fazendo desnecessários grande parte dos comentários ou diagramas de fluxo explicativos.

Terá notado no exemplo a adoção de variáveis tipo INTEGER. Isto é devido ao fato de que no nosso exemplo a moeda não emprega decimais e, além disso, a que os números de 6-7 cifras que utilizamos, expressos em tipo REAL, dariam lugar a resultados em forma exponencial, do tipo $1.0000E^6$ (na realidade, o tipo INTEGER tem uma classe no UCSD que é limitada, aproximadamente, a 32.000, valor a partir do qual seria necessário adotar o tipo LONG). Neste momento se tropeça com a rigidez do Pascal em matéria de tipos: geralmente está proibido misturar "maçãs com peras"; somente é permitido juntar reais e inteiros, com prioridade sempre para os primeiros. Além disso, se o primeiro membro é de tipo inteiro e o segundo tem algum termo real é produzido um erro,

isto é, não é realizada a transformação automática em tipo INTEGER. Por sorte, o Pascal proporciona a função TRUNC (que já utilizamos na fórmula para obter o total de impbrut). Além da DIV, já vista, podemos usar a operação MOD, que proporciona o resto da divisão entre inteiros (ex.: resto: = dividendo MOD divisor).

Queremos aproveitar para esclarecer que entre os objetivos deste livro introdutório não são encontrados temas como:

- cálculo numérico,
- elaboração das E/S em geral;

ou tratamento de arquivos.

Para estes pontos, ou para ampliar detalhes sobre os vistos, recomendamos a leitura de manuais mais especializados.

Voltando ao exemplo, fazemos notar que também poderiam ter sido inseridas instruções como:

```
IMPLIQ: = IMPBRUT(IMP)-DESGR;
WRITELN ('O IMPOSTO LIQUIDO FICA IGUAL A: ',IMPBRUT(IMP)-DESGR);
```

RECORRER À RECURSIVIDADE

Todo o dito pode ajudar a reconsiderar a diferença entre FUNCTION e PROCEDURE. Deixamos como tarefa para os mais voluntariosos a complementação do mesmo programa precedente por meio de uma PROCEDURE que, naturalmente, contenha um parâmetro-variável oportuno, algo assim como:

```
PROCEDURE IRRF(VAR IMPBRUT: INTEGER, IMPONIB:INTEGER);
```

Nos resta comentar uma última diferença entre os dois módulos pascalianos fundamentais: o tratamento da recursividade. Dado que é um dos argumentos mais fascinantes, e em certo modo misteriosos, de linguagens evoluídas como o Pascal, vale a pena que façamos um vão neste livro. Na eterna disputa entre BASICalistas e Pascalistas, os segundos, às difamações dos primeiros ("Onde estão as cadeias?, Onde os arquivos aleatórios?, onde...?") reagem sistematicamente encolhendo os ombros e aludindo com dignidade a este potente e estranho instrumento.

Para começar a apresentá-lo tem que comentar sua estreita

relação com os discursos sobre discursos. Exemplo trágico: disse Epaminondas o Cretense: "o que estou dizendo é falso". Exemplo humorístico: "Havia uma vez um Rei que disse à sua criada: "Conta-me um conto!", e ela começou: "Havia uma vez um Rei que disse a sua criada: "Conta-me um conto!", e ele começou: "Havia uma vez um Rei...".

A série de aspas que fecham o segundo caso tem que ser estabelecida como infinita, porque este conto, formado por uma cadeia de reis que mandam falar a criadas que falam de reis que mandam falar a criada que falam de..., evidentemente não acaba NUNCA. Quanto ao primeio caso, os leitores mais estudiosos, esses que, talvez esporadicamente, leiam "Investigação e Ciência" (de "Scientific American") sabem já que dá lugar à famosa contradição, em base à qual o ambíguo Epaminonda se diz o verdadeiro diz também o falso, e vice-versa. Mas deixemos de divagações e voltemos ao Pascal.

Dito com palavras, a recursividade, em informática, consiste em que um procedimento dado pode, desde seu interior, chamar a si mesmo. O jogo, como se intui, poderia durar até o infinito, mas se inserem critérios adequados sempre internos de finalização ou então uma condição de final que se integra mais ou menos implicitamente na definição.

Em vista de que dizem que a tartaruga foi criada precisamente para demonstrar o fácil que é compreender, inclusive para as crianças, assuntos como a recursividade, começaremos com um exemplo geométrico clássico com a tartaruga:

```
PROGRAM QUADRADOS;
  PROCEDURE QUADRO(LADO,INC, ALFA:INTEGER);
  VARI:INTEGER;
  IF LADO < =200 DO
  BEGIN
     TURN(ALFA);
     FOR I = 1 TO 4 DO
     BEGIN
       MOVE(LADO);TURN(90)
     END
     QUADRO(LADO + INC,INC,ALFA)
  END;
BEGIN ( * MAIN * )
  MOVETO(120,100);PENCOLOR(WHITE);
  QUADRO(5,2,6)
END.
```

Depois do dito, o resultado do programa não será nem imprevisível, nem chocante: partindo do centro da tela, e devido à primeira chamada a Quadro por parte do main, é traçado um quadrado de 5 unidades de lado e com um desvio de 16 graus, terminado o qual a coisa é repetida porque o Quadro chama a si mesmo, mas esta vez com um parâmetro "lado" aumentado em 2. Segundo isto, até onde poderíamos chegar? Teoricamente até o infinito, com quadrados cada vez maiores, se não fosse pelo providencial IF, que bloqueia o processo quando o lado é maior que 200.

Mudando os parâmetros podemos ver os distintos efeitos que provocam seguindo o perigoso costume de quem se põe a jogar com a tartaruga. Por exemplo, sugerimos deslocar a instrução Quadro (lado + inc, inc, alfa) auto-invocadora antes de END que a precede, de forma que agora a autochamada ocorra depois de cada MOVE (lado) e TURN (90):

```
FOR I = 1 TO 4 DO
BEGIN
   MOVE (LADO);TURN(90)
   QUADRO(LADO + INC,INC,ALFA)
 END ( * LOOP FOR * )
 END;
```

Deste pequeno exemplo podemos extrair duas conclusões:

- em alguns casos a recursividade produz efeitos que são obtidos mais facilmente com uma simples iteração;
- portanto, é melhor usar esta técnica sofisticada somente em casos especiais.

Em conseqüência, antes de continuar desenvolvendo-a sem motivo ou causa, vamos tentar aprofundar neste argumento.

FUNCTIONS e PROCEDURES recursivas

Convém fazer em seguida uma pequena prova para satisfazer uma dúvida que seguramente surgiu em muitos de vocês. Eliminem no exemplo precedente (em sua forma inicial) o IF, sem esquecer os respectivos BEGIN...END e anulando inclusive a expansão do quadrado (fazendo no main: Quadro (20, 0, 6). De verdade durará até o infinito o traçado destes 360/6 = 60 quadrados? A resposta é negativa: depois de um certo tempo o sistema pro-

duzirá uma mensagem de "stack overflow" (superação da capacidade da pilha) e parará. Efetivamente. Efetivamente, isto pode ajudar a entender melhor o que estamos a ponto de dizer: o mecanismo da recursividade acumula em uma pilha ("stack") os parâmetros e variáveis locais pendentes da resolução. Agora então, toda pilha dispõe de uma área finita da memória e (a menos que sejam colocados em jogo mecanismos ainda mais sofisticados...) esta, antes ou depois, ficará saturada, chegando a transbordar-se ("overflow").
Vejamos um caso simples: o de uma somatória, desde "1" até "n", a cujo total chamaremos Tot(n). A definição, também em matemática, pode ser dada assim:

TOT(N) = 1 SE N = 1, EM CASO CONTRARIO
TOT(N) = TO(N-1) + N

Assim, por exemplo, se n = 6 teremos Tot(6) = Tot(5) + 6. Não é banal? Certamente, mas a sutileza está em reconhecer que esta definição É RECURSIVA POR SI MESMA, enquanto que nos obriga implicitamente a voltar a um cálculo, ainda que seja em potência e não de fato. Geralmente é preferida a forma direta, calculando primeiro Tot(1), depois Tot(2) = Tot(1) + 2, etc. A essência do programa é óbvia:

```
....
      TOT: = 1;
      FORI: = 1 TO N DO
      TOT: = TOT + I
```

Mas, por que não obrigar ao computador a fazer contas definidas recursivamente? Em Pascal o fazemos em seguida:

```
      PROGRAM SUMAREC;
      VAR N,I,TOTAL:INTEGER;
        FUNCTION TOT  (NUMERO:INTEGER):INTEGER;
        BEGIN
          IF NUMERO = 1 THEN TOT: = 1
          ELSE TOT: = TOT(NUMERO-1) + NUMERO;
        END;
BEGIN ( * MAIN * )
REPEAT
      READLN(N);I: = N;
IF N < = 4000 THEN
```

```
BEGIN
   TOTAL: = TOT(N);
   WRITELN:( * RETORNO DE CARRO * )
   WRITELN ('SOMATORIA ATE ',N', = ,TOTAL);
END
UNTIL N > 4000
END.
```

A chave deste assunto reside no fato de que quando no segundo membro de uma expressão é encontrado um termo que inclui o nome de uma FUNCTION pascaliana é realizada uma chamada a dita função. Isto já havíamos visto a propósito de um WRITELN (e tornaremos a ver); agora nós o encontramos em uma expressão interna à FUNCTION mesma, com o qual supõe uma recursividade. O jogo recursivo consiste em que a auto-evocação seja realizada cada vez com valor distinto (uma unidade menor) do parâmetro "número". A condição de final de processamento é cumprida quando o valor "número 1" de chamada é 1. Quem "se fiar" do fato de que o Pascal sustenta a recursividade tem que "crer" que estas coisas sucedem, e por outro lado, a prova empírica pode ser obtida simplesmente fazendo correr o programa e proporcionando-lhe diferentes valores de "n". A propósito: terá notado uma variante do usual WHILE mediante REPEAT que, por esta vez, permite utilizar um só READLN. De todas as formas, mediante comparação, poderá obter a parada do programa por "stack overflow" (superação da capacidade da pilha) para valores altos de "h" (ao redor de 2000 para um computador pessoal corrente). Isto estimula novamente nossa curiosidade sobre o tema, além de recordar-nos que não devemos abusar da recursividade, especialmente nos peque nos computadores domésticos. Para tentar compreender por nós mesmos este mecanismo, os convido a incluir uma nova linha depois do ELSE de Tot:

```
I: = I-1;WRITELN('RASTRO: ',I);
```

Por que se adotou uma variável global? Pois porque não era muito simples fazê-lo de outra forma: uma variável local teria sido mais adequada, pois se quer seguir as pegadas do que acontece no local, mas era muito emaranhado o problema da inicialização, pelo menos para um principiante. Se colocarmos no main o valor inicial "i" igual a "n", a instrução acima vista parece a mais lógica. Na prática, o que é que acontece? Aqui a saída do programa no caso n = 4:

```
Rastro: 3
Rastro: 2
Rastro: 1
Rastro: 0
Somatória até 4 = 10
```

Seguramente o 0 produziu alguma surpresa. Vejamos o que ocorre se modificarmos a nova linha desta maneira:

```
I:I = 1;WRITELN('RASTRO: ',NUMERO, '    ',I);
```

O resultado da execução seria:

```
Rastro: 1 3
Rastro: 2 2
Rastro: 3 1
Rastro: 4 0
Somatória até 4 = 10
```

Esperamos não ter causado nenhum ataque cardíaco. Ainda que em princípio seja incrível, a associação de valores para "número" e "i" é correta. De fato, a recursividade deixa pendente a realização de todos os cálculos que não podem ser executados (cada vez que o parâmetro "número" decresce), mas conserva os valores com os quais terá de fazê-los na pilha. Assim, quando chegamos à presença da condição IF, com número = 1 teríamos problemas se não tivessem sido conservados um a um todos os valores precedentes de "número" com os quais tem que trabalhar agora. Quando são precisos são repescados *em ordem inversa* à qual foram "empilhados" na pilha usada pelo Pascal. No caso de n = 4 pouco a pouco teremos que acrescentar a Tot: = 1, 2, 3 e 4, nesta ordem.

Os que ainda não estão de todo convencidos perguntarão: por que então os valores de "i" vão desde 3 a 0? Isto depende do fato de que "i" é uma variável global e, portanto, NÃO está armazenada na pilha. No entanto, a execução da instrução i: = i-1 e os sucessivos WRITELN ficaram em reserva pelas chamadas "de telescópio" do Tot (número-1) e somente quando o número = 1 começará a decrescer i: esta é a explicação de por que "i" vai desde 3 a 0, diminuindo-se 4 vezes antes de voltar definitivamente ao main.

Na Figura 1 são representados, em quatro planos distintos, as sucessivas variações da pilha, com a operação evocada (e em reserva) e os parâmetros empilhados. Desde um ponto de vista geral, tudo o que é "local" — isto é, variáveis e parâmetros — é reservado na pilha; é por isto que, em problemas nos quais estão em

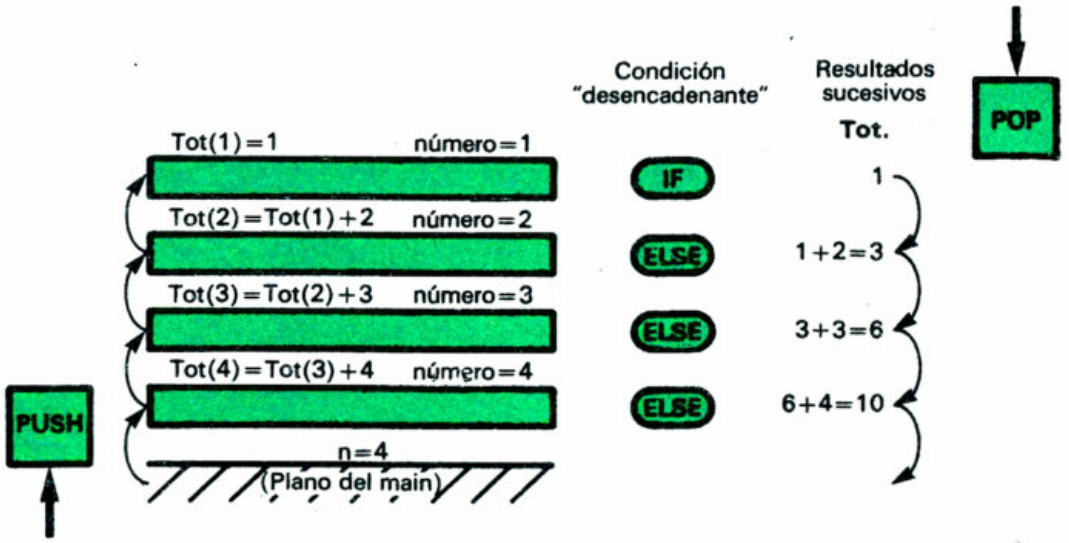


Figura 1 — A figura trata de evidenciar o funcionamento da pilha interna (stack) guardando parâmentros e variáveis locais e mantendo em reserva fórmulas quando é invocada uma recursiva, como no programa da somatória.

jogo um maior número de parâmetros, a pilha é "derrubada" antes.

As flechas que apontam para cima, à esquerda, simbolizam as 4 operações sucessivas de PUSH (empurrar literalmente), enquanto que à direita são representadas palavras condicionais como IF ou ELSE. A presença de IF em cima (último plano) dá lugar a um desbloqueio da situação (que em outro caso chegaria ao infinito ou, na realidade, ao transbordamento da pilha). Então é ativado o mecanismo de cálculo para trás, com os 4 POP (extrair, o contrário de Push), necessários para voltar ao main, com a conseqüente recuperação dos distintos valores.

Por último, vejamos outra possível variante, adotando um PROCEDIMENTO (naturalmente com um parâmetro-variável). Ainda que é evidente que para um caso analítico é preferível uma FUNCTION, propomos isto como um útil exercício explicativo. Nossa solução é esta:

```
PROGRAM SUMAREC;
VAR N,SOMA:INTEGER;
PROCEDURE TOTAL
(VAR TOT:INTEGER;NUMERO:INTEGER);
BEGIN
```

```
  IF NUMERO = 1 THEN TOT: = 1
   ELSE
      BEGIN
        TOTAL(TOT,NUMERO-1); TOT: = TOT + NUMERO;
      END;
    WRITELN ('RASTRO: ',TOT,'',NUMERO)
END;
BEGIN ( *  MAIN * )
.....
IF N < 2000 THEN
BEGIN
  TOTAL(SOMA,N);WRITELN (SOMA)
END
UNTIL...
END.
```

Agora os comentários quase sobram. A distinta estrutura nos fez introduzir uma nova variável global 'soma' para fazê-la passar como parâmetro "de referência" ao procedimento Total. Assim podemos destacar por fim a última diferença entre FUNCTION e PROCEDURE: nas operações de cálculo, a primeira está, por assim dizê-lo, melhor disposta para a recursividade, enquanto que a segunda deve chamar-se explicitamente e deve ser seguida pela instrução que maneja o parâmetro-VAR; o mecanismo também está garantido, mas ao estar a segunda operação em reserva pela evocação de Total, automaticamente fica guardada na pilha habitual (uma pergunta: o que teria sucedido se nós tivéssemos esquecido de encerrar as duas operações entre BEGIN e END...?

Talvez exista a pequena vantagem da maior comodidade com a qual pode se ter agora um rastro dos cálculos "para trás" (aqui convém fazer outra pergunta: qual é o melhor lugar para situar a instrução seguinte:
WRITELN ('Rastro: 'tot,'' número);? Tem que raciociná-la antes de prová-la no computador pessoal.

Árvores e dragões

Esta longa e prolixa conversa à toa sobre temas elementares teria fins unicamente pedagógicos, pois, como muitos de vocês sabem, a somatória dos inteiros de "1" a "n" vem dado diretamente pela fórmula:

S(n) = n*(n + 1)/2
Exemplo: S(4) = 4*5/2 = 10

Como já aprendemos muitas úteis sutilezas podemos passar a desenvolver bastante velozmente exemplos mais complexos. Ainda que nos limitaremos a casos gráficos da tartaruga, podem estar seguros de que a recursividade encontra aplicações muito sérias, sejam ou não numéricas, nos mais diversos campos, tendo como terreno privilegiado o da chamada Inteligência Artificial, permitindo mecanismos de procura em árvores (decisões, movimentos estratégicos e similares) que, em caso contrário, seriam extremamente incômodos, para não dizer impossíveis de realizar em forma iterativa. Geralmente, e a grandes rasgos, pode ser afirmado que a recursividade é cara em termos de memória (pela grande quantidade de memória que necessita para a pilha) e de velocidade de execução (por culpa dos PUSH e POP, ainda que isto possa variar de um problema a outro), mas tem a seu favor a vantagem de sua potência expressiva, além de computacional, e seu caráter sintético.

Passemos agora ao exame de outro caso interessante, muito citado nos manuais de Logo.

```
PROGRAM CURVASDODRAGÃO;
USES TURTTLEGRAPHICS;
VAR NIV:INTEGER;
  PROCEDURE DRAGAO(NIVEL, LADO:INTEGER);
    BEGIN
      IF NIVEL = NIV THEN
      MOVE(LADO)
      ELSE
        BEGIN
          TURN(45);
          DRAGAO(NIVEL + 1,ROUND(0.707 * LADO);
          TURN(-90)
          DRAGAO(NIVEL + 1,ROUND(0,707) * LADO);
          TURN(45)
          END; ( * ELSE * )
        END;
BEGIN ( * MAIN * )
    INITTURTLE
    FORNIV: = 0 TO 8 DO
    BEGIN
      PENCOLOR(NONE);MOVETO(70,50);
      PENCOLOR(WHITE); DRAGAO(1,140);
      READLN
```

```
  END ( * FOR * )
END.
```

Se colocaram a propósito as instruções para fazer funcionar o programa concretamente com o Pascal do Apple, com o fim de permitir ao menos aos possuidores deste computador realizar um experimento, o que sempre ajuda a compreender o mecanismo. De fato, as 8 vezes que é realizado o ciclo FOR do main são traçados outros tantos fragmentos recursivos da curva, interrompidos pela instrução READLN, assim apertando Return pode vê-los perseguindo um a um. Se, ao contrário, quer ver desenhado tudo de uma vez, basta suprimir READLN.

Examinado agora a definição do procedimento Dragão se vê imediatamente que é composto de três fases: um giro de 45 graus mais a execução de Dragão, um giro de 90 graus no sentido dos ponteiros do relógio, e logo, novamente, o Dragão seguido de uma rotação de 45.

Em ambos os casos tem auto-evocação, com redução do parâmetro lado (passado inicialmente com um valor de 140 desde o main) enquanto que o parâmetro nível é incrementado em uma unidade.

Em base ao que temos dito até agora não é demasiadamente difícil dar-se conta do que ocorre depois dos dois primeiros passos do loop FOR, ou seja quando a variável global "niv" tem valores 0 e 1. No primeiro é verificada a condição nível = niv, logo ELSE nem sequer é levada em conta e é traçado um segmento horizontal de lado 140, devolvendo o controle ao main. Quando niv, no segundo passo, vale 1, é verificada a condição que ativa ELSE, porque agora nível = 0 e niv = 1. Portanto, é conectado o mecanismo de recursividade com um "lado" 0.707 vezes (arredondado no interior com a função ROUND) o valor que lhe foi passado pelo main. Com a segunda auto-evocação é completada a tripla fase descrita mais acima. Para todos aqueles que sabem que 0.707 (o inverso da raiz de 2) é o coeficiente que liga o lado com a diagonal do quadrado não será difícil compreender que teremos o traçado da curva assinalado com "nível 1" na Figura 2. Duas observações:

- a cada novo passo do FOR o par de instruções PENCOLOR (NONE) e MOVETO (70,50) volta a colocar a tartaruga no mesmo ponto de saída;
- como é evidente na Figura 2 a tartaruga acaba sua viagem com um ângulo relativo final zero (+ 45 - 90 + 45).

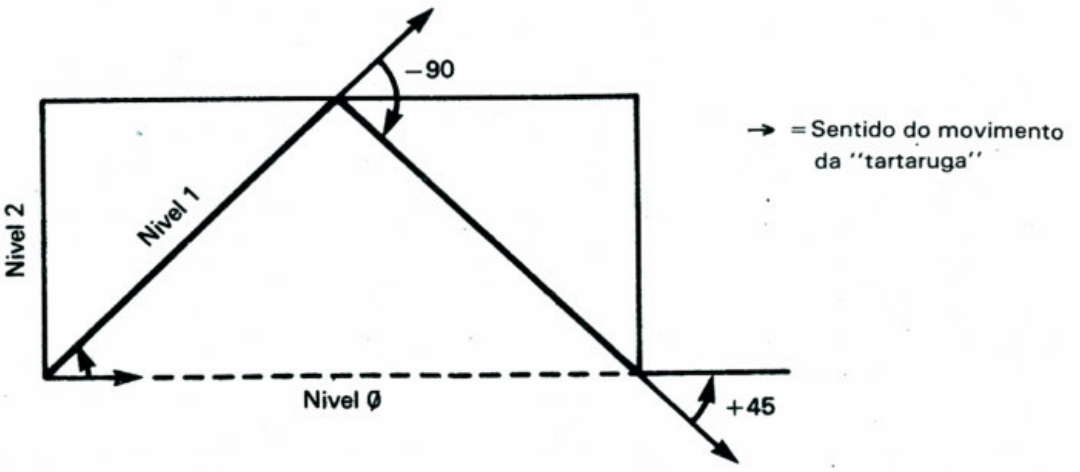


Figura 2 — Percursos da tartaruga no programa Curvas do Dragão nos valores 0, 1, e 2 de nível. No caso do nível 1 são colocadas em evidência as rotações relativas da tartaruga. O ângulo final resultante é nulo.

Se chamamos "Dente de dragão" a este triângulo retângulo (a ponta do dente forma um ângulo de 90), compreender os movimentos sucessivos significa dar-se conta de que as coisas vão de tal forma que é traçado um dente de dragão sobre cada um dos lados que competem ao nível imediatamente precedente.

Nos falta espaço para uma análise mais detalhada e porisso os remeto ao exemplo anterior. Aqui nos limitaremos a fazer uma observação importante, sem a qual qualquer um pode ser despistado facilmente. As chamadas recursivas a Dragão (nível + 1, ROUND(0.707 * lado)) são dois, portanto, nos perguntamos: quando a primeira permanece em reserva, tem que seguir repetindo-a até chegar a nível = niv ou tem que passar a examinar a segunda chamada a Dragão? A resposta correta é a primeira, como pode comprovar se tem a paciência suficiente com a pilha, ainda que basta com "convencer-se" de que o compilador vê como um todo aquilo que tem entre o BEGIN e o END do ELSE, deixando, por assim dizer, em reserva o quadro executivo inteiro.

Concluindo, no nível 2, por exemplo, teríamos a seguinte sucessão de fatos: giro de 45 (nível 1 em reserva): giro 45 (niv 2), verificação de IF e conseqüente execução do primeiro dos 4 lados; giro de -90 (niv 2), segundo lado (devido ao segundo Dragão que também está em reserva), e giro de 45 (com o qual a tartaruga agora está paralela ao primeiro lado do niv 1); segue giro de 90 (agora nível 1!), etc.

Uma última observação: as várias instruções TURN, que não são procedimentos recursivos, ainda que estejam em reserva (poderíamos dizer que por culpa de outros), ao final são atendidas por igual, enquanto que move (lado), condicionada por IF, é executada somente no último nível. Se alguém não se convenceu disto, pode tentar fazer seguir READLN por uma instrução INITTURTLE (CLEARSCREEN em outros sistemas) ou inclusive pode provar a eliminar o FOR com uma única instrução de atribuição tipo niv: = 6, se dará conta de que somente é desenhada uma curva.

Outras indicações

A Figura 3 ilustra o desenho que é obtido fazendo correr o programa precedente até o nível 5. Por cima são obtidos efeitos imprecisos devidos à aproximação da função ROUND (e, em definitivo, à baixa resolução gráfica). Podem ser obtidas variantes de diferentes maneiras, por exemplo, mudando o dente do dragão, introduzindo elementos pseudo-aleatórios e assim sucessivamente. Outros critérios podem ser do tipo ilustrado pelo programa da Figura 4 e sua correspondente execução (Fig 5).

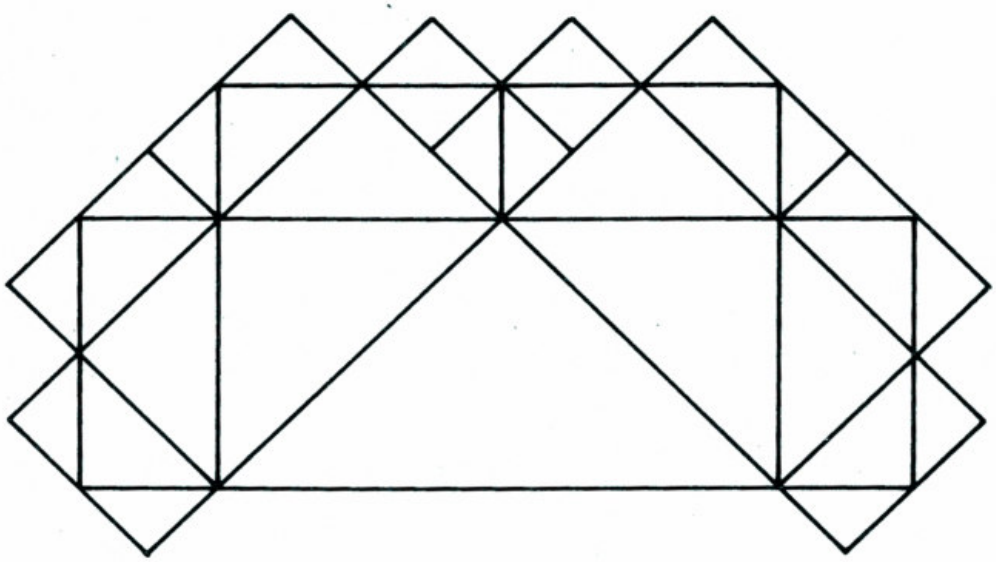


Figura 3 — Curvas do Dragão até o nível 5.

Também se fala de recursividade mútua entre procedimentos... Nos parece que alguém murmura uma pergunta: mas todas estas estranhas coisas, na prática, para que servem? Novamente repetimos que a recursividade serve para muitos outros campos, como a busca de dados em estruturas de árvore, também na base da Inteligência Artificial, onde o varrido com técnicas recursivas de árvores de decisão (por exemplo, o estudo dos movimentos em um jogo) está à ordem do dia, e inclusive é o fundamento de linguagens especiais como o LISP e o Prolog. Em diferentes manuais aparece a solução recursiva do brilhante e instrutivo jogo das Torres de Hanói, que, portanto, nos limitamos a citar.

```
PROGRAM, ARBORESCENCIA;
VAR ESCALA, ORDEM: INTEGER;

PROCEDURE ARBOL (X, Y, COMP, DIR:INTEGER);
PROCEDURE MUDAXY;
VAR DELTAX, DELTAY: INTEGER;
BEGIN
   IF DIR < 0 THEN DIR: = DIR + 8;
   IF DIR > = 8 THEN DIR: = DIR-8:
   CASE DIR OF
     0: BEGIN DELTAX: = 1;DELTAY: = 0;END;
     1: BEGIN DELTAX: = 1;DELTAY: = 1;END;
     2: BEGIN DELTAX: = 0;DELTAY: = 1:END;
     3: BEGIN DELTAX: = -1;DELTAY: = 1;END;
     4: BEGIN DELTAX: = -1;DELTAY: = 0;END;
     5: BEGIN DELTAX: = -1;DELTAY: = 1;END;
     6: BEGIN DELTAX: =  0;DELTAX: = -1;END;
     7: BEGIN DELTAX: =  1;DELTAY: = -1;END;
  END; ( * CASE * )
  X: = X + ESCALA * COMP * DELTAX;
  Y: = Y + ESCALA * COMP * DELTAY:
END; ( * MUDA XY * )

BEGIN ( * ARBOL * )
   PENCOLOR (NONE)
   MOVETO (X,Y);TURNTO(DIR   45);
   PENCOLOR(WHITE);
   CAMBIAXY;MOVETO(X,Y);
   IF LONG > 1 THEN
    BEGIN
     ARBOL(X,Y,COMP-1,DIR + 1);
     ARBOL(X,Y,COMP-1,DIR-1);
```

```
    END;
     END;( * ARBOL *. )
BEGIN ( *  MAIN  *  )
  WRITE ('ESCALA? ');READLN (ESCALA);
  WRITE ('ORDEM? ');READLN (ORDEM);
  ARBOL(0,-ESCALA * ORDEM-100,ORDEM,2);
END
```

Figura 4 — O programa Arborescência ilustra eficazmente o conceito de recursividade.

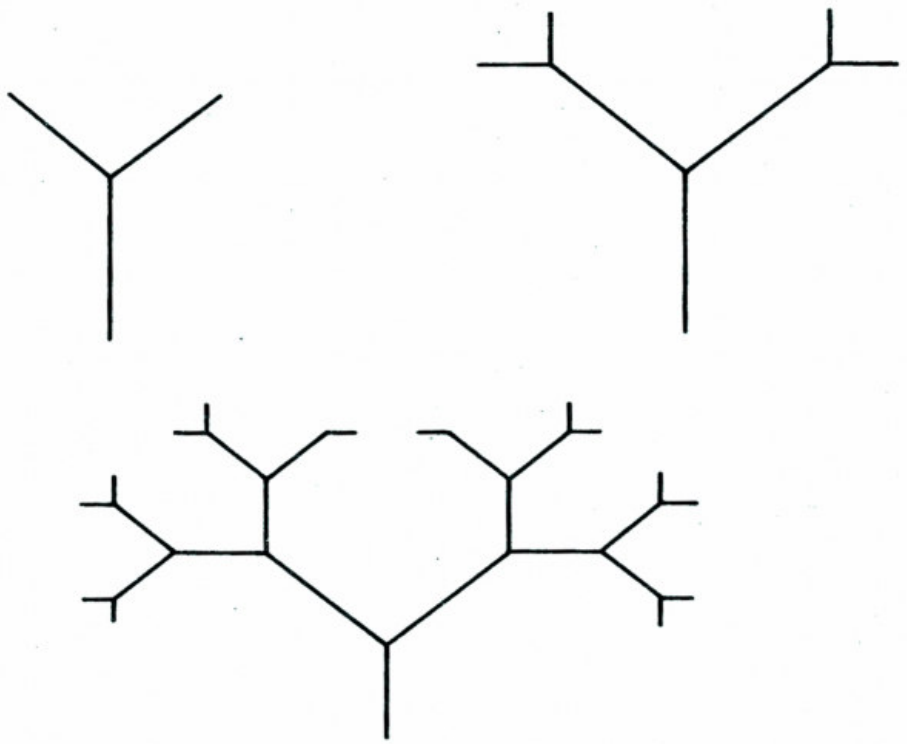


Figura 5 — Árvores de nível 1,2, e 4 criadas pelo programa Arborescência, cuja listagem aparece na figura 4.

O assunto sai dos objetivos deste livro; basta assinalar que técnicas gráfico-recursivas na base de linhas recortadas geradas pelo computador que, variando pacientemente diferentes parâmetros e combinando-os sabiamente com uma migalha de aleatoriedade, dão lugar a supreendentes imagens que evocam montanhas, vales, paisagens lunares fantásticas e irreais. A arte computadorizada é uma mistura fascinante de macânica e criatividade.

E o BASIC?

Nós poderíamos perguntar se é de todo exato que o BASIC não suporta a recursividade. Provemos a verificar o funcionamento deste programa:

```
100 REM RECURSIVIDADE EM BASIC
110 GOTO 500:REM INICIO MAIN
120 REM SUBROTINA RECURSIVA
130 IF I = FINAL THEN 150
140 I = I + 1: PRINT "EVOCAÇÃO NUMERO";I:GOSUB 120
150 PRINT:PRINT "RECURSIVIDADE, SENHORES!";I
160 RETURN
500 REM MAIN
510 INPUT "ME INDIQUE O FINAL",FINAL
520 GOSUB 120
530 END
```

Um BASICalista distraído ao qual fizemos esta pergunta, seguramente contestará que o programa, por exemplo para final = 5, escreve:

```
EVOCAÇÃO NUMERO 1
EVOCAÇÃO NUMERO 2
.....
EVOCAÇÃO NUMERO 5
RECURSIVIDADE,SENHORES! 5
```

Ao contrário, muito surpreso, verá aparecer a mensagem RECURSIVIDADE, SENHORES! contudo 5 vezes. O estranho é que por um lado se tem uma interlinha e, além disso, não são impressos os valores decrescentes, de I desde 5 até 1 como talvez esperaria um pascaliano distraído, porém repetidamente:

```
RECURSIVIDADE,SENHORES! 5
RECURSIVIDADE,SENHORES! 5
.....(etc.).....
```

Dado que também em BASIC (e inclusive em linguagem máquina) as subrotinas são baseadas em uma pilha que acumula os valores de retorno, o primeiro ponto é explicado em seguida: se as operações em reserva são as linhas 150, também faz parte delas o primeiro PRINT e porisso se repete. Quanto à monotonia do valor 5 do exemplo, é devido a que em BASIC não se tem parâme-

tros e todas as variáveis são globais.

Assim compreendemos finalmente como pode ser vantajosa, ainda que pesada às vezes, a distinção entre variáveis de diferente nível que nos pareceu tão pedante.

Por último, tem que provar dando a FINAL valores cada vez mais elevados: nos daremos conta de que esta recursividade do BASIC é verdadeiramente espartana (a mensagem de saturação de capacidade sai para um FINAL de 20 ou 30 e as coisas pioram tragicamente com programas longos).

Se quiser pode construir artesanalmente uma pilha em um programa BASIC, de forma que possam ser reservadas todas as variáveis e parâmetros locais.

É possível, mas é muito mais cômodo tê-la já toda feita de antemão.
Como em Pascal.

CAPÍTULO VI

ESTRUTURAS DE DADOS E ALGORÍTMOS: TUDO GUARDA RELAÇÃO

Tipos definidos pelo usuário

Retomamos agora o tema da estrutura dos dados que já vimos a grandes passadas nos primeiros capítulos. Por questões de espaço teremos que esquematizar muito este tema. Por outro lado, isto é um ensaio (pequeno) e não um tratado. Para aqueles a quem aborrece a teoria queremos recordar que sem a abstração não podem ser dados passos adiante nem sequer na prática. De todo modo, além de fazer um rápido resumo sobre a estrutura de dados do Pascal, colocaremos exemplos ilustrativos (inclusive com seus correspondentes programas equivalentes em BASIC). Assim trataremos de dar também resposta ao angustioso (e por agora não resolvido) problema do delineamento estruturalista de uma linguagem popular que carece de formas estruturadas. Entenda-se bem que falamos de dialetos comuns, nos quais não existe praticamente nem rastro de tipos estruturados. Aludiremos fugazmente também a novos dialetos estruturados, como o SUPERBASIC, lançado por Sinclair em seu QL.

Para comodidade do leitor repetimos na Figura 1 o quadro sinótico apresentado no capítulo 2. Os standards, isto é, aqueles que vem dados pela linguagem, já os damos por conhecidos, pois fomos apresentando-os à medida que surgia a ocasião. Junto a INTEGER, REAL, CHAR e BOOLEAN, vimos também o tipo STRING oferecido no UCSD Pascal por Bowles (mais generoso que o "avarento" Niklaus Wirth).

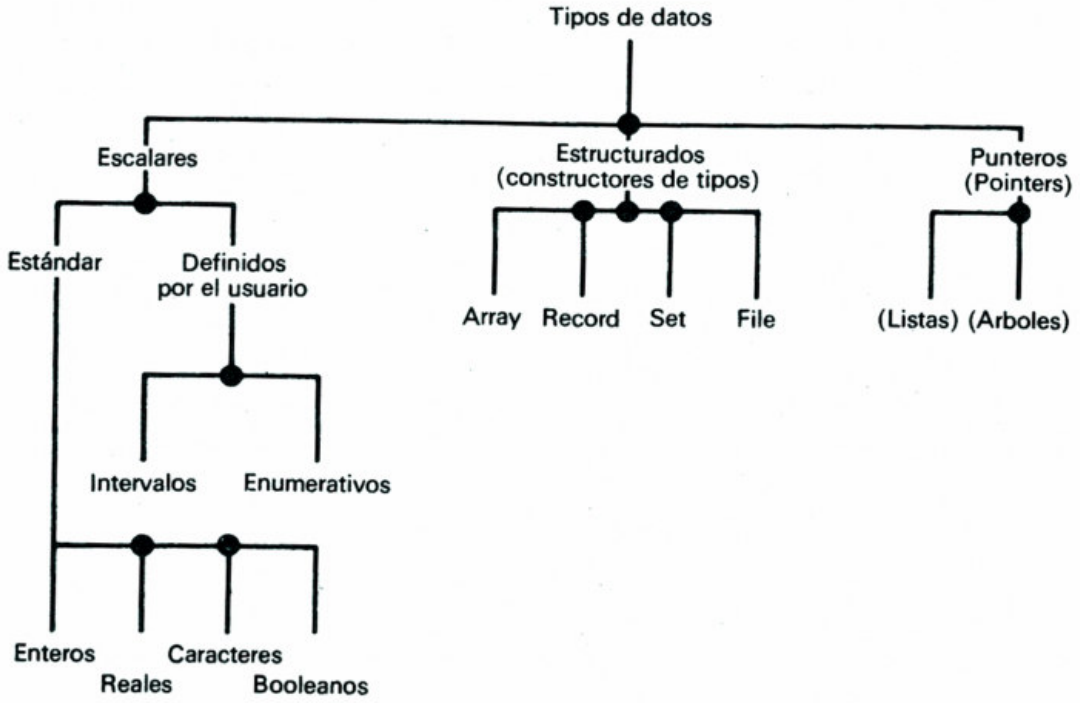


Figura 1 — Quadro sinótico dos tipos proporcionados pelo Pascal.

Segundo a classificação da Figura 1, os standards pertencem aos chamados tipos "escalares", Esta denominação deriva do fato de que podem assumir valores únicos que formam uma espécie de escala de valores, em fila um após o outro. Se pensarmos bem esta "granularidade" dos dados escalares não é adaptada, à primeira vista, aos números reais, que formam um domínio contínuo no numerável; mas estas sutilezas não têm sentido na aritmética de precisão finita do calculador, onde também os números reais estão formados por um número máximo de cifras.

Um amigo nosso definiu os tipos escalares definidos pelo usuário como "a desprezada do Pascal", mas sua utilização inteligente permite uma grande clareza nos programas e alguma vantagem inesperada.

Estes tipos escalares descritos pelo usuário são definidos de duas formas: por intervalo (como subconjunto de outro tipo, excluídos os reais) ou então por enumeração e que, uma vez definidos, nos permitem desfrutar de:

- as funções Standard SUCC(x), PRED(x) e ORD(x) que pro-

porcionam, respectivamente, o elemento sucessor, ou o antecessor de um dado "x", ou seu número de ordem;
- as relações > , <e=.

Aqui um exemplo instrutivo. Se refere ao problema do transbordo em uma barca por um camponês de um lobo, uma cabra e uma couve-flor. Esta é a solução que propomos:

```
PROGRAM CABRALOBOCOUVEFLOR;
TYPE PERSONAGENS = (CAMPONES, LOBO, CABRA,
     COUVEFLOR);
     PASSAGEIRO = (LOBO...COUVEFLOR);
VAR PAS1, PAS2: PASSAGEIRO;
.....
IF (ORD (PAS1)) = (ORD(PAS2)-1) THEN COMILONA
.....
```

O exemplo ilustra tanto a definição por enumeração (no TYPE personagens) como a de intervalo (no TYPE passageiro).

O problema deste programa é que deve ser avaliado em um determinado momento, se existe o risco de que seja ativado um procedimento (genérico e intuitivo) chamado Comilona. Agora então, a condição de IF é adaptada perfeitamente. Tanto ao lobo que come a cabra, como a esta que come a couve-flor sem necessidade de utilizar IF aninhados nem CASE: ubi maior minor editur (o peixe grande come o menor).

O tipo ARRAY

Com este tipo, chamado às vezes "vetor", já que somente tem uma dimensão, pode ser feito quase tudo em BASIC (em particular se consegue imitar estruturas como as listras e as árvores, segundo veremos seguidamente). Para ser breve e supondo que tratamos com pessoas que sabem BASIC (senão basta recorrer a outros volumes da B.B.I.) daremos por conhecidas as noções sobre dimensão e índice. A este último, em Pascal é chamado também "seletor", vai colocado entre colchetes [ ] e é modificável.

À diferença dos loucos da eletrônica, que concebem um array como um conjunto de células colocadas na memória, os estruturalistas pascalianos o vêem como um tipo abstrato assimilável ao conceito matemático de matriz. Vejamos a definição genérica de um tipo-array em Pascal:

```
TYPE Reparto = ARRAY [Ind.1, Ind2...] OF Tipo.
```

Tipo pode ser por sua vez de qualquer tipo(escalar, estruturado, standard ou criado por nós) enquanto que Ind1, Ind2... têm que ser escalares. Isto permite uma variedade de combinações impensáveis no BASIC.

```
PROGRAM DIETARIO:
TYPE MESES = (JAN,FEV,MAR,ABR,MAI,JUN,JUL,
              AGO,SET,OUT,NOV,DEZ);
   DINHEIROMENS = ARRAY[MESES] OF INTEGER;
VAR MES:MESES;ENTRADAS, SAIDAS:DINHEIROMENS;
.....
SAIDAS[DIC]: = SAIDAS[DEZ] + PAGAEXTRA;
SALDO: = 0;
FOR MES: = JAN TO DEZ DO
  BEGIN
    ENTRADAS[MES]: = ENTRADAS[MES]-SAIDAS[MES];
    SALDO: = SALDO + ENTRADAS[MES]
END;
WRITELN ('BALANCO ANUAL:',SALDO)
```

Tem que reconhecer que, uma vez definido o ARRAY dinheiro-mens que tem como índice ao tipo-escalar mês, é bem simples manipular com a variável mês, o ciclo FOR para calcular o hipotético balanço anual.

O fundamental é que o Pascal permite para os arrays multidimensionais uma representação mais eloqüente que a DIMX (10, 20, 10) do BASIC.

Para fazer entender melhor a riqueza de expressão que pode ser conseguida com o tipo ARRAY, faremos um exemplo, parecido ao anterior, mas supondo gastos e benefícios divididos em três gêneros: compensações, mercadoria e serviços.

```
PROGRAM DIETARIO:
TYPE MESES = (JAN,FEV,MAR,ABR,MAI,JUN,JUL,
              AGO,SET,OUT,NOV,DEZ);
   GENEROS=(COMPENSAÇÕES,MERCADORIAS,SER-
              VIÇOS);
   DINHEIROMENS = ARRAY[ME-
              SES] OF INTEGER;
   VAR     MES:MESES;VOC:GENE-
```

```
                    ROS;ENTRADAS,SAIDAS:DINHEIRO-
                    MENS;
    .....
```

Quem preferir pode dispor também da fórmula habitual. No caso anterior seria:

```
    DINHEIROMENS = ARRAY[MESES,GENEROS]     OF
INTEGER;
```

Agora, a parte do cálculo pode estar formada por dois loops FOR aninhados um no outro; o interior pode totalizar sobre os três gêneros.

```
    .....
    SAIDAS[DIC,COMPENSACOES]: = SAIDAS[DIC,COMPEN-
SACOES] + PAGAEXTRA;
    SALDO: = 0;
    FOR MES: = JAN TO DEZ DO
      FOR VOC: = COMPENSACOES TO SERVICOS DO
      BEGIN
        ENTRADAS[MES,VOC]: = ENTRADAS[MES,VOC]-
SAIDAS[MES,VOC];
        SALDO: = SALDO + ENTRADAS[MES,VOC]
    END:
WRITELN ('BALANCO ANUAL:',SALDO)
```

Nós alongamos um pouco sobre estas interessantes peculiaridades. Como já dissemos, os índices são escalares, geralmente inteiros, e para estabelecer as dimensões pode se recorrer ao critério da classe, que consiste em separar com pontinhos os extremos do campo ao qual pertencem, por exemplo:

```
    VETOR = ARRAY[100...1000] OF INTEGER
```

É apreciada em seguida a maior flexibilidade com respeito ao BASIC, onde os índices vão sempre de 0 ao valor definido por nós.

Vejamos agora outro caso onde uma estrutura adequada nos serve para esclarecer a resolução de um algorítmo e, sobretudo, sua implementação. Recordamos novamente que os conceitos de inter-relação entre algorítmos e estruturas serão tratados, simplesmente, no volume "Programando como se deve". É conveniente

```
PROGRAM Crivo;
CONS campo = 10000;
VAR hipotprim: ARRAY (2..campo) OF BOOLEAN;
        primo, pr, incr, contaprimos: INTEGER;

PROCEDURE todos primos;
VAR ind: INTEGER;

BEGIN
     FOR ind: = 2 TO campo DO
          BEGIN hipotprim (ind): = TRUE END
     END; (* Todos primos *)

BEGIN (* Main *)
todos primos;
FOR primo: = 2 TO campo DIV 2 DO
     IF hipotprim (primo) THEN
       BEGIN
       incr: = primo; pr: = primo + incr;
         REPEAT
              hipotprim (pr): = FALSE;
              pr: = pr + incr;
           UNTIL pr campo;
         END;
WRITE (''Lista primos no campo'');
WRITELN (''de 2 a'' ,campo); WRITELN;
contaprimos: = 0;
FOR primo: 2 TO campo DO
     BEGIN
          IF hipotprim (primo) THEN
          BEGIN
            contaprimos: = contaprimos + 1;
            WRITELN (primo);
          END;
     END;
     WRITELN;
     WRITE (''Total primos:'', contaprimos);
END.
```

Figura 2 — Programa que realiza o algorítmo do crivo de Eratóstenes. A estrutura de dados que melhor lhe adapta é um array de valores booleanos, ao princípio todos TRUE e trocados progressivamente por valores FALSE. Ao acabar são números primos os índices correspondentes aos valores do array "hipotprim" que permaneceram em TRUE.

que tente desde agora redigir o programa equivalente em BASIC. O algorítmo que vamos ver é o clássico "crivo de Eratostenes", que consiste em supor primeiros primos todos os inteiros do campo a examinar, depois do qual são eliminados todos os múltiplos de dois, logo todo os múltiplos de três e assim sucessivamente. Se quisermos trabalhar com um array, nos damos conta, depois de algumas reflexões, de que a estrutura de dados mais adequada para este simples algorítmo não é um array de inteiros, mas um array de booleanos estabelecidos inicialmente como TRUE e que vão ficando FALSE pouco a pouco com a técnica do crivo (ver listagem da Figura 2).

Seguindo a listagem é descoberta outra questão: os índices, tanto em Pascal como em BASIC, são variáveis eles mesmos e, em casos como estes, podem ser manejados como tais. De fato, quando acaba o processamento do exemplo são impressos como números primos somente os índices correspondentes aos booleanos corretos no array de crivo.

Os menos distraídos se darão conta de que desta vez não introduzimos ARRAY como simples definidor de uma VAR. De fato, ARRAY tem duas caras: define a uma VAR de tipo array, ao mesmo tempo que serve também como construtor de tipo. Geralmente nos convém definir um TYPE (estruturado), já que fica bastante complexo quando se tem várias VAR do mesmo tipo.

RECORD, SET e FILE. Os outros tipos estruturados

O tipo RECORD foi considerado por Wirth como "talvez o mais flexível dos tipos de dado". Historicamente, nasceu com os cartões perfurados, subdivididos em diferentes campos, cada um dos quais contém um dado de distinta natureza, mas que permite identificar um determinado sujeito. A sintaxe em Pascal é a seguinte:

TYPE < tiporecord > = RECORD < campos > END

Dois exemplos típicos: 1) O cartão de registro que contém a série ordenada de sobrenomes, nome, lugar de nascimento, endereço, etc., e 2) uma entrada qualquer de uma tabela que associa o correspondente preço (ou desconto ou qualquer outra coisa que se quer) ao código (ou nome) do artigo. Geralmente, o prestígio das linguagens estruturadas é devido ao fato de que proporcionam uma visão abstrata que nas linguagens tradicionais costumava ser

relegada aos registros (records) situados em suportes externos. De todas as formas, o BASIC não oferece este tipo de vantagens e o programador tem que regular as mesmas com os tipos comuns.

Fixemo-nos, por exemplo, em uma tabela de descontos. Em BASIC teríamos que apanhar-nos com dois arrays, digamos NOMART (NOME ARTigo) e DESC, que logo teremos que fazer andar juntos sob nossa pessoal supervisão (nada difícil, mas aborrecido e pouco operacional). Ao contrário em Pascal seria:

```
TYPE tabeladescontos = RECORD
                         nomart:STRING;
                         desconto:INTEGER
                         END;
```

Em suma, se trata de uma lista de dados, de distintos tipos, fechados entre as palavras reservadas RECORD e END. Outro exemplo clássico:

```
    PROGRAM numeroscompl;
TYPE complex = RECORD rea,imag: REAL END
    VAR x,y,z:complex;
    PROCEDURE somacompl (VAR
somacompl:complj;a,b;complj);
      BEGIN
       somacompl.rea; = a.rea + b.rea;
       somacompl.imag: = a.imag + b.imag.
      END;
    PROCEDURE procdcompl (VAR
procdcompl;complej;a,b:complej);
      BEGIN
         procdcompl.rea: = a.rea * b.rea + a.imag * b.imag;
         procdcompl.imag; = a.rea * b.imag + a.imag * b.rea;
      END;
    .....
    BEGIN ( * main * )
      WRITELN ('me dê 2 numeros complexos ');
      WRITE ('(primeiro a parte real, depois a imaginaria)');
      READLN (x.rea,x.imag,y.rea,y.imag);
      sumacompl (z,x,y); WRITELN;
      WRITELN ('sua soma vale: ',z.rea,' + j ',z.imag);
      prodcompl (z,x,y); WRITELN;
      WRITELN ('seu produto vale: ',z.rea,' + j ',z.imag);
      .....etc.....
```

O exemplo "fala" por si mesmo. É interessante assinalar, no entanto, que ao campo de um registro (record) se tem acesso colocando o nome do registro seguido por um ponto e o nome do campo. Note também que usamos um PROCEDIMENTO; de fato, em Pascal não são admitidas FUNCTIONS que não sejam do tipo escalar.

São interessantes as combinações deste tipo de dados com outros, especialmente com o tipo array. Antes de colocar um exemplo disto, devemos referir-nos ao chamado RECORD "com variantes". Em essência, se trata de um RECORD (registro) que contém pelo menos um campo que condiciona, segundo seus possíveis valores (um número pré-fixado e limitado) o significado de outros campos. Por exemplo:

```
TYPE mes = (jan,fev,mar,abr,mai,jun,jul,
            ago,set,out,nov,dez);
     data = RECORD dia:1..31;mes:mes;ano:INTEGER END
     categart = (confec,agranel,deterioravel);
     producto = RECORD
                  descrip  :ARRAY[1..20] OF CHAR:
                  armaz:RECORD
                          codigma:INTEGER;
                          nomema:ARRAY[1..10] OF CHAR;
                          END:
                 CASE cat;categart  OF
                          confec:(deposito:INTEGER);
                          agranel:(deposito,decim:INTE-
                          GER;um:CHAR);
deterioravel:(deposito:INTEGER: estropeada:BOOLEAN)
                  END;
```

Este exemplo deveria ser suficientemente claro (veja o dito no capítulo 2). Nos permite apreciar como a estrutura RECORD admite cômodas subdivisões em subcampos. Quanto a CASE que aqui está aplicada às "variantes", consente campos de significado e estrutura alternativa: no caso específico de que o campo discriminador "cat" contenha "confec" irá seguido de um campo ("depósito" de tipo inteiro; se, ao contrário, são artigos a granel, o seguirão dois campos inteiros "depósito" e "decim") e um terceiro campo de cadeia com a unidade de medida (abreviada), enquanto que na categoria "deteriorável" se estabelece que "depósito" vai seguido por um campo booleano que adverte se está ou não deteriorada (tenha em conta que isto é somente um exemplo...).

Aqui três possíveis atribuições significativas: uma hipotética VAR prod do tipo "produto" recém definido:

```
prod.armaz.codigma: = 123
prod.cat: = confec
prod.jaqueta: = 150.000.
```

Por último, poderia ser imaginado um ARRAY de elementos do tipo produto:

```
VAR inventar:ARRAY[1..200] OF produto
```

e localizar com inventar [i] depósito (ou parecidos) um campo do i-ésimo produto.

Vejamos rapidamente o tipo SET (conjunto) que permite formar conjuntos de dados. A sintaxe usa as palavras reservadas SET OF seguidas por um tipo que em Pascal somente pode ser escalar ou subclasse.

Uma vez definido um tipo SET podem ser feitos nele uma série de operações de união de conjunto (+), intersecção (*), subtração (-), consistente em quitar do primeiro todos os elementos que também são do segundo). As comparações, além de com = e <> (cujo significado é óbvio), pode ser feitos com < = ou com > =, que significam inclusão. Vejamos um exemplo:

```
TYPE personagens = (lobo,cabra,couveflor,campones);
        brigada = SET OF personagens;
VAR barca,orlaesq,orladir;brigada;
.....
orlaesq: = orlaesq + barca;
IF (lobo IN orlaesq) AND (cabra IN orlaesq) THEN...
```

O programa é capaz de oferecer-nos o momento em que ao subir a barca à orla esquerda pode ser produzida uma situação delicada simulada pela operação união e a sucessiva análise da situação; (lobo IN orlaesq) restitui o valor booleano TRUE se no conjunto "orlaesq" de "brigada" é encontrado o elemento 'lobo' e o mesmo com (cabra IN orlaesq). Se ambas são certas THEN as conseqüências são intuitíveis...

A instrução WITH serve, na fase de elaboração, para manejar comodamente os registros.

Praticamente não falaremos do tipo FILE (fluxo seqüencial); somente confirmar-lhes que corresponde exatamente ao que parece e recomendar-lhes manuais mais especializados. Por outro la-

do é urgente tratar, ainda que seja rapidamente, organizações mais sofisticadas de dados que somente podem ser manejadas com comodidade utilizando as linguagens estruturadas.

As estruturas dinâmicas: pilha (stack) e demais

Ao tratar os distintos tipos pascalianos passamos por alto algumas coisas. Em compensação vamos falar agora, de forma abreviada, do tema talvez mais delicado (depois da recursividade, Depende dos gostos...). Portanto, preparem-se para utilizar sua matéria cinzenta.

Até a poucos momentos estávamos vendo estruturas "estáticas", isto é, dados organizados de uma forma mais ou menos complexa, mas que não mudava no curso do programa. As estruturas dinâmicas, que iremos vendo neste e nos seguintes itens, podem ser: pilhas (stack), códigos (code), listagens e árvores; estes elementos fazem a delícia dos informáticos que projetam compiladores e intérpretes, ou inclusive daqueles que se ocupam da Inteligência Artificial.

As pilhas já vimos antes quando tratamos a recursividade, apontando o fato de que o stack tem uma importância vital no Pascal. Também dissemos que os mesmos microprocessadores adotam uma pilha para guardar nela o estado de seus próprios registros internos e a direção de entrada das subrotinas chamadas (uma vez que se sai delas, é fácil recuperar tudo o que havia sido guardado temporariamente na pilha).

Repetimos que uma pilha é definida intuitivamente como um montão de elementos dos quais somente o último é "visível". Em inglês, uma estrutura como esta é chamada"memória LIFO" (Last In First Out), último a entrar primeiro a sair, dado que sempre é extraído em primeiro lugar o último elemento armazenado (que estava, portanto, em cima da pilha).

Do ponto de vista mais rigoroso, uma pilha consiste em uma estrutura de elementos homogêneos sobre a qual são definidas duas operações fundamentais: PUSH (literalmente "empurra") e POP (extrai), uma contrária a outra. Com a primeira é carregado um elemento (ou qualquer estrutura, já que por sua vez poderia estar composto de mais elementos, como um array ou um record) na parte superior da pilha. Geralmente se acrescenta uma estrutura chamada EMPTY que serve para indicar se a pilha está vazia ou, se preferir, "desinflada".

O mecanismo da pilha é particularmente cômodo para o cál-

culo algébrico na chamada notação polace inversa (RPN - Reverse Polish Notation, utilizada, por exemplo, nas calculadoras HP). Com a RPN, uma expressão de tipo (a + b) * (c - d) seria escrita:

```
a ↑ b + c ↑ d -*
```

que não necessita parêntesis. De fato, ficam subentendidas (como bem sabem os usuários de calculadoras de bolso Hewlett Packard) as duas regras seguintes:

- ao encontrar um dado o carregamos na pilha (operação PUSH);
- se encontramos um operador aritmético é executada a operação correspondente usando o dado situado em cima da pilha (top) e aquele imediatamente por baixo, consequente operação POP.

Na expressão vista antes, a sucessão de acontecimentos se corresponde com a sequência ilustrada na figura 3.

Neste mesmo mecanismo da RPN é baseada a linguagem FORTH (um próximo volume da B.B.I. permitirá aos curiosos aprofundar nele).

Seria interessante (mas deixamos como exercício para os mais avançados) imitar em Pascal ou BASIC um sistema de cálculo similar. Ao contrário, teremos que nos contentar em ver como poderia ser simulada uma pilha em Pascal e em BASIC. Em BASIC isto é indispensável em todos os casos práticos, enquanto que o Pascal, que possui sua própria pilha. Portanto, teremos que nos ajustar, em ambos os casos, com um array.

A realização é mostrada na figura 4. Como se pode ver, o ponteiro "punt" (em BASIC chamado PT) é, na realidade, um vulgar índice que, no procedimewnto PUSH, é incrementado antes de carregar o seguinte elemento do array, enquanto que em POP primeiro é extraído o elemento e depois é diminuido punt. Nos dois exemplos tem características não necessariamente universais, como a escolha de elementos de tipo cadeia no array (a cadeia, única riqueza verdadeira do BASIC, pode tornar-se cômoda para carregar grupos de dados, mas em Pascal pode ser preferível o tipo-record) ou o uso de uma FUNCTION para o POP (alguém poderia preferir que uma mesma variável "z" servisse como bandeira de carregamento para PUSH ou de descarregamento para POP). Outra particularidade é ter escolhido a função Offlimits para assinalar qualquer condição anômala do índice (a outros teria sido melhor que

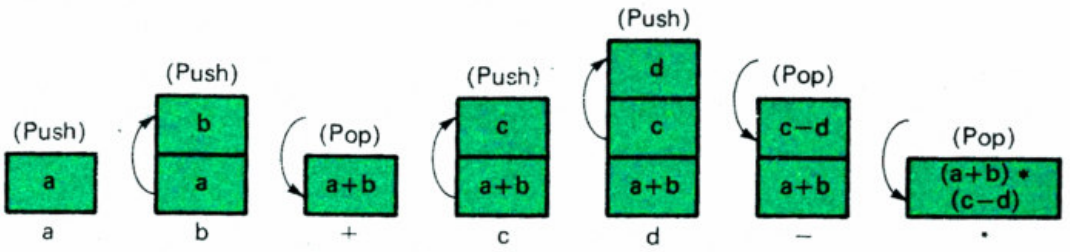


Figura 3 — Sucessivos Push e Pop da pilha necessários no cálculo de uma fórmula aritmética com a notação polaca inversa (RPN).

se distinguisse entre a situação de "empty" - vazia - e a de "overflow" - transbordamento -).

Quanto ao programa BASIC, o paralelismo com o Pascal nos estimulou a utilizar DEF IN, chegando ao ponto de colocar o main debaixo, precedido pela subrotina.

Um parêntesis: o dilema Bottom-Up/Top-Down

Ao ter-nos centralizado na criação de programas para o manejo da pilha (stack) reutilizáveis em outro lugar, deixamos solto o MAIN, constituído por um vazio encerrado entre os terminadores BEGIN e END. Isto nos oferece a oportunidade de fazer uma observação: não é de todo certo que o Pascal se presta exclusivamente a um delineamento botton-up (de baixo para cima). Se o desejar, pode utilizar o top-down (de cima para baixo). Esta é a demonstração:

```
PROGRAM acima abaixo;
VAR o, que, te, parece: STRING;
FUNCTION condic: BOOLEAN;
  BEGIN
    READLN (o)
    IF o = 'preg' THEN condic: = TRUE
      ELSE condic: = FALSE
  END;
PROCEDURE dia;
BEGIN
END;
PROCEDURE tarde;
BEGIN
END;
```

```
PROCEDURE manhã;
BEGIN
END;
FUNCTION passado manhã;
END;
BEGIN ( * main * )
dia; tarde;
IF condic THEN manhã
  ELSE passado manhã
END.
```

```
PROGRAM emuladorpilha;
  CONT maxpunt: = 200;
  VAR pilha: ARRAY (1..maxpunt) OF STRING;
  punt: INTEGER;

PROCEDURE pricpilha;
  BEGIN punt: = 0;
  END;

FUNCTION offlimits: BOOLEAN;
  BEGIN
      IF punt = maxpunt THEN
        offlimits: = TRUE;
      ELSE offlimits: = FAUSE;
      END;
PROCEDURE push (x:STRING);
  BEGIN
      IF NOT offlimits THEN
        BEGIN
          punt: = punt + 1; pilha (punt): = x
        END;
    END;

FUNCTION pop: STRING;
  BEGIN
      IF NOT offlimits THEN
      BEGIN
        pop: = pilha (punt); punt: = punt-1;
      END;
END;

BEGIN (* Main *)
..
..
```

```
END.
```

```
100  MAXPT = 200: DIM Pilha $ (MAXPT)
110  GOTO 500: REM SALTO AO INICIO DO MAIN
120  DEF FN OFFLIMIT = (PT = 0) OR (PT = MAXPT)
130  REM PUSH
140  IF OFFLIMIT THEN RETURN
150  PT = PT + 1: PILHA$ = X$
160  RETURN
170  REM POP
180  IF OFFLIMIT THEN RETURN
190  POP = PILHA$ (PT): PT = PT-1
200  RETURN
500  REM INICIO DO MAIN
```

Figura 4 — Emulação da estrutura de pilha: a) em Pascal e b) em BASIC.

O exemplo é humorístico, mas pretende expor como determinamos com precisão a forma exata na qual terá que comportar-se o main e as peças que o compõem (funções e procedimento), será preciso, antes de tudo, o main mesmo, enquanto que seus módulos poderão estar vazios. Dado que o compilador permanece indiferente ante procedimentos e funções vazias, formadas somente por pares BEGIN/END, poderá ser expurgado exclusivamente o main. Logo faremos os refinamentos passo a passo, definindo os diferentes módulos, com suas correspondentes comprovações, conforme vai sendo completado o esquema.

Como se pode intuir com o exemplo, o desenvolvimento não é, em absoluto, banal: foi necessário inserir a função Condic para emular, mediante o teclado, um fato que proporcionará alternativas. Se todos os main se deixassem reduzir a simples sequências de procedimentos, inicialmente vazios, seria estupendo, mas por desgraça, não costuma passar. Em suma, nos casos reais a conexão entre módulos e programas principais é muito estreita e a aproximação progressiva do delineamento top-down somente contempla um aspecto da programação estruturada.

Antes de acabar com as pilhas devemos referir-nos a outra estrutura: a cauda (queue em francês), que também aparece sob as siglas FIFO (First In First Out, primeiro a entrar, primeiro a sair). As funções básicas de uma cauda (ou lista de espera) podem ser definidas como "põe na cauda" ("coloca") e "tira" ("serve"). É utilizada nos sistemas operacionais para gestionar os armazenamentos intermediários de entrada/saída (onde são guardados os dados transmitidos ou recebidos), e nos sistemas multiusuário para gestionar equanimemente as solicitações de acesso (à impressora, disco rígido, etc.); também poderiam ser utilizadas em um programa de simulação de fatos baseados na teoria das caudas (em estatística, por exemplo).

A emulação de uma cauda, com o uso do habitual e serviçal array, não é muito difícil. Bastará com dois índices-ponteiros, o primeiro para a colocação de dados em cima e o segundo para retirar os do final; desta vez ambas as operações aumentam os ponteiros. Isto provoca um deslizamento contínuo até a parte alta da memória, que faz indispensável dispor de uma organização circular para o array. Não se assustem, que não é muito: bastará obrigar a reinicialização do ponteiro quando chegar ao valor máximo, mediante instruções do tipo:

```
BEGIN
   pontuac: = pontuac + 1;
   IF pontuac = pontuacmax THEN
     pontuac: = 0
.....
```

LISTAS LINEARES

Do ponto de vista exclusivamente lógico, uma lista linear L é um conjunto de elementos dispostos consecutivamente. Geralmente são definidas as funções seguintes (com estes ou outros nomes):

- Primeiro (L), que nos dá o primeiro elemento de L;
- continuação (L), que nos dá o que fica de L depois de ter tirado o que era primeiro elemento;
- inserção de um novo elemento entre dois elementos quaisquer;
- supressão de um elemento qualquer.

Não se trata, como no caso das matrizes (arrays) de estruturas de acesso direto, mas de todas as formas seu caráter dinâmico permite um manejo fácil, baseado em uma modificação contínua dos elementos que fazem parte do conjunto.

O ponto característico das listas (e, como será visto, das árvores) é que não se trabalha com índices, mas com *apontadores*, isto é, com parâmetros, que, normalmente, fazem parte do elemento e que "apontam" ao seguinte. Fala-se de estruturas em "cadeia". Na Figura 5 é colocado em evidência o jogo de apontadores necessários para inserir e para suprimir um elemento.

O típico exemplo de lista linear é um texto, entendido como um conjunto de linhas (ou palavras no qual podem ser inseridos ou suprimidas linhas (ou palavras). Analogamente, os setores de um disco freqüentemente estão dispostos em forma de listas, para permitir uma gestão dinâmica do espaço.

Nas linguagens estruturadas como Pascal (para não falar de Modula 2 ou Ada) qualquer elemento de uma lista está composto por dois campos: o campo do valor e o campo do apontador

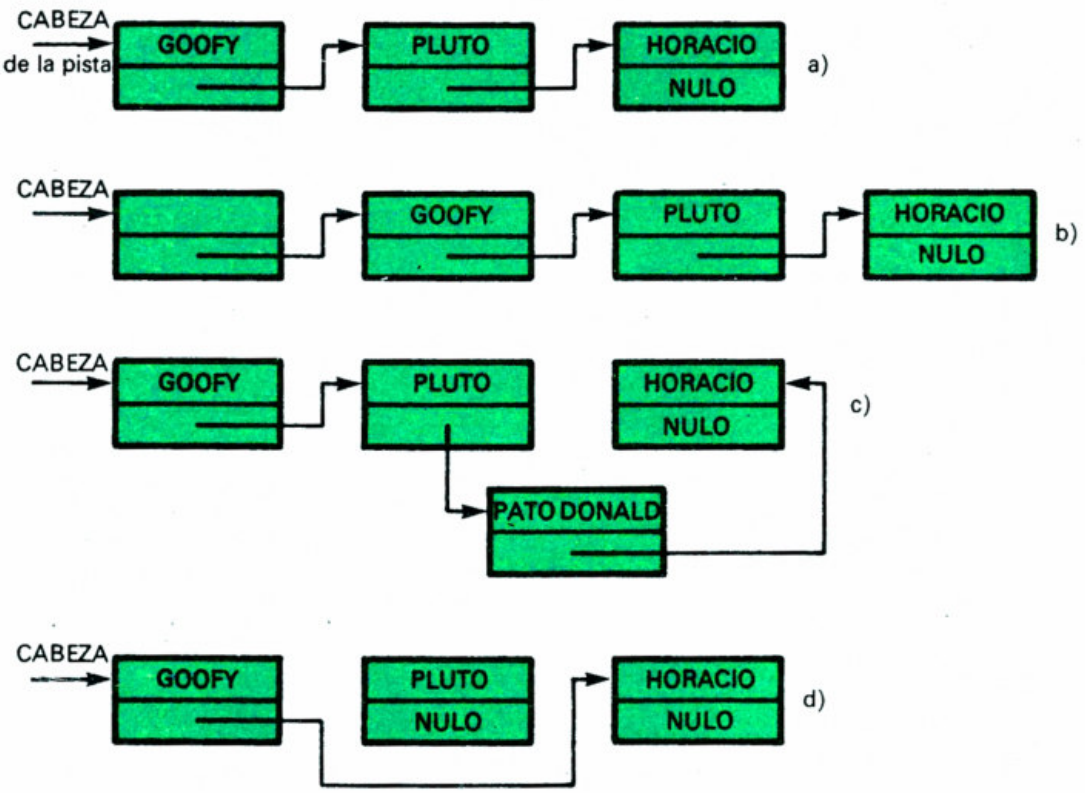


Figura 5 — a) Esquema da organização de uma lista linear. Por norma, o primeiro ponteiro olha para a "cabeça" da lista. Em b) é ilustrada a inserção de um novo elemento. Manejando adequadamente os ponteiros pode ser inserido um novo elemento em qualquer lugar. c) Por último, em d) temos a eliminação de um elemento: é suficiente mudar um ponteiro.

(pointer). O apontador é um tipo muito abstrato. De fato, não se diz como está representado"materialmente": será um índice inteiro?, um código? Não se sabe; depende da implementação. De qualquer forma, não é demais saber que, na totalidade dos casos práticos, se trata da direção do dado atá o qual "aponta". Em Pascal é usada a seguinte definição:

```
Var apont: ↑ elemento;
```

onde a flecha (que também é o símbolo de exponenciação, frequentemente substituído pelo acento circunflexo"^") indica que "apont" é um tipo-pointer que aponta para"elemento". Um valor particular do apontador é NULO (NIL), que significa que aponta...para nenhum lugar. Serve para listas vazias ou para assinalar o final (o último elemento tem um apontador NULO). Vejamos como pode ser definida (recursivamente) uma lista de palavras:

```
TYPE apont: ↑elemento;
     elemento = RECORD
                valor:STRING;
                continuação: ↑elemento;
                END;
```

Vale a pena observar de que este é o único caso do Pascal no qual um "elemento" é referido antes que esteja completa sua definição. O tipo "lista" está definido como um apontador para "elemento" que, por sua vez, é um registro formado pelos campos "valor" e "continuação" - que aponta o seguinte - (obviamente os campos de dados poderiam ser mais de 1, o mesmo que os apontadores para listas mais complexas ou árvores).
Na Figura 6 temos: a listagem do programa que contém procedimentos em Pascal para o manejo de uma lista linear de caracteres (6a) e seu equivalente em BASIC (6b); no segundo caso temos que regular por força com dois vetores, o dos valores e o dos apontadores.

Em vista de que estes temas serão mais detalhadamente tratados em um próximo livro referente ao manejo de dados, nos limitamos a duas notas indispensáveis sobre o tipo pointer:

- um elemento da lista é especificado com uma instrução do tipo pt ↑.valor;
- a função new(pt) dada pelo Pascal serve para criar um apontador pt, ou seja, para deixar espaço cada vez que é enserido um novo elemento (assim será aberto e intercalará um espaço em memória).

```
PROGRAM listalinear;
TYPE lista = ↑ elemento;
       elemento = RECORD
                    valor: CHAR;
                    contin: lista
                    END;
FUNCTION primeiro (a: lista): CHAR;
  BEGIN
  primeiro: = a↑.valor;
  END
FUNCTION sucessor (a: lista): lista;
     BEGIN
     sucessor: = a ↑.contin
     END
FUNCTION empty (a: lista): BOOLEAN;
  BEGIN
     IF a: = NIL THEN
        empty: = TRUE
  ELSE
        empty: = FALSE
  END;
FUNCTION inser (c:CHAR; a:lista): lista;
VAR pt: lista;
  BEGIN
     new (pt);
     pt↑.valor: = c;
     pt↑.contin = a;
     inser: = pt
  END;
FUNCTION busca (c:CHAR:a:lista):lista;
( *da o primeiro elemento que coincide com c * )
VAR enc:BOOLEAN;
  BEGIN
     enc: = FALSE;
     WHILE NOT enc AND (a < > NIL) DO
        IF c = a ↑.valor THEN enc: = TRUE
        ELSE a: = contin(a);
     busca: = a
  END;
END;
BEGIN ( * main * )
END.

100 DIM  LISTA $(200), SEG(200): GOTO1000
150 DEF  FN PRIM $(L) = LISTA $ (L)
200 DEF  FNSUCC(L) = SEG(L)
250 DEF  FNEMPTY(L) = L = 0
300 REM  CRIAÇAO DE NOVO ELEMENTO
```

```
310 REM COM RESULTADO EN X
320 LIB = LIB + 1:REM NOVO COLOCADO
          LIVRE
330 X = LIB
340 RETURN
350 REM
390 REM EMULAÇAO DA FUNÇAO
400 REM INSER $ (C $ :CADEIA: L1:LISTA)
410 REM RESULTADO
420 GOSUB 300: REM EMULA LA new(pt)
430 LISTA $ (X) = C $ :SEG(X) = L1:L2 = X
440 RETURN
450 REM
480 REM EMULAC. DE SOPPRESS (L:LISTA)
490 REM RESULTADO EM L
500 L = FNSUCC (L)
510 RETURN
520 REM
570 REM EMULAC. DE BUSCA (C $ CADEIA; L1:LISTA)
580 REM RESULTADO EN L2
590 REM B = VARIAVEL (PSEUDO) LOCAL
600 B = -1:L2 = L1
610 IF L2 < > 0 AND B THEN 640
620 IF LISTA $ (L2) = C $ THEN B = 0:GOTO 610
630 L2 = FNSUCC(L2): GOTO 610
640 RETURN
1000 REM INICIO MAIN
.....(ao prazer).....
```

Figura 6 — Funções típicas para o tratamento de uma listagem linear, em Pascal (a) e em BASIC (b).

Com referência às funções da Figura 6, recordem que uma lista é localizada apontando a seu primeiro elemento, o que supõe que, recorrendo-a para frente, "são perdidos" os elementos varridos. Se não quisermos que isto aconteça, será necessário utilizar duplos apontadores (listas bidirecionais). Finalmente, temos que dizer que, no programa da Figura 6, a função-lista Inser acrescenta um novo caracter "c" à cabeça da lista "a", porque se quiser uma inserção intermediária deverá usar previamente a função Busca.

Outra lacuna é o apagar, mas ocupá-la é simples. Utilizando sempre um apontador auxiliar (pot, por exemplo), como na inser-

ção, o cancelamento de um elemento que segue a pt é feita com uma só instrução:

pt ↑ .contin: pt ↑ .contin.contin.

Efetivamente, no segundo membro se tem o deslocamento de dois elementos e o apontador resultante é atribuído ao primeiro membro, saltando-se assim o elemento intermediário.

Trepar nas árvores

O exemplo visto na Figura 6 justifica em parte o motivo que induziu Wirth a não incluir o tipo cadeia nem sequer no Modula 2: de fato, uma vez definida como lista de CHAR, uma cadeia é construída e manipulada com grande facilidade.

Outras estruturas de dados dinâmicas são as listas bidirecionais e as circulares: as primeiras dotadas de dois apontadores por elemento, dos quais um aponta ao sucessor e o outro ao predecessor, e as segundas caracterizadas porque o apontador do último elemento olha ao primeiro (não existe nenhum apontador NIL). Estas listas unem as vantagens já vistas -eliminações e inserções fáceis - do varrido seqüencial; por exemplo, se quisermos manter uma certa ordem cada vez que é acrescentado um elemento, em uma lista ordenada segundo um critério, não será necessário recorrer à lista desde o princípio, mas basta ir desde onde estamos até o ponto no qual o elemento novo tenha que ser inserido, atribuindo-lhe então o código intermediário correspondente.

A árvore (tree, em inglês) nasce também como estrutura por esta exigência, mas possui muitas outras vantagens e aplicações. A árvore constitui um conjunto organizado em sentido hierárquico, e está geralmente "invertido", ou seja, representado com o tronco para cima: seu vértice, chamado "raiz", está acima, enquanto que as ramificações estão para baixo. Em conjunto, uma árvore é um "grafo", isto é, um conjunto de nós unidos por ramificações. De cada nó partem ramificações que se dirigem ("apontam") a nós de nível inferior. Usando uma terminologia botânica, os nós terminais são chamados "folhas". Ás vezes também é usada uma nomenclatura de tipo genealógico, e neste caso se falará de nós "pai" e ramificações "filho". Evidentemente, uma estrutura de dados arborescente também pode ser utilizada para representar uma autêntica árvore genealógica: na Figura 7 vemos o pedigree de uma hipotética geração de cachorros de raça. Isto vale para as classificações por gêneros e espécies, nas ciências naturais, no arqui-

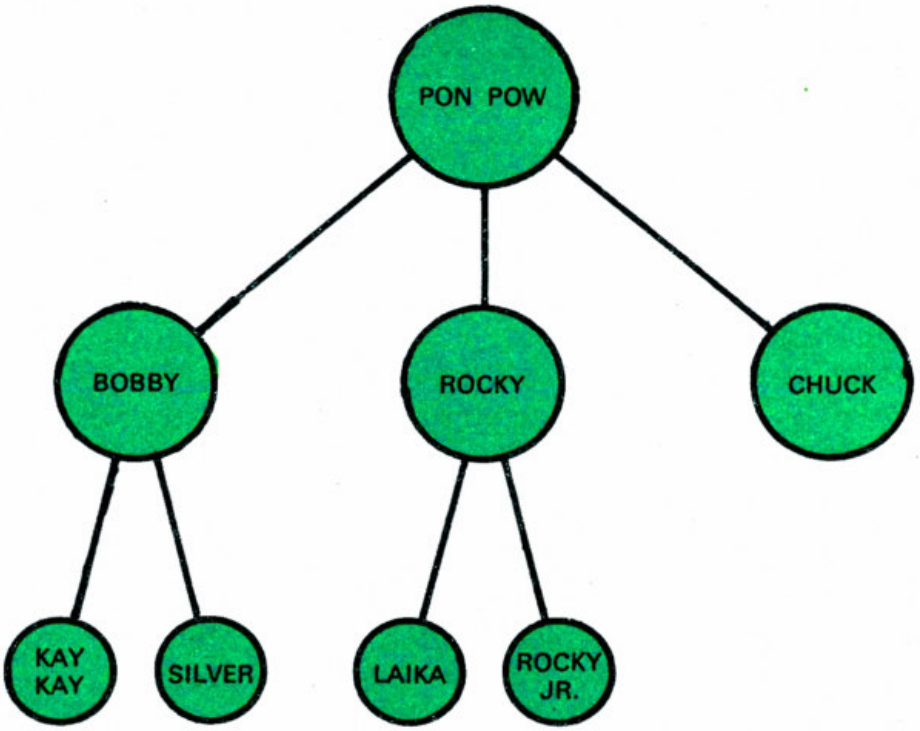


Figura 7 — Uma estrutura de dados em árvore invertida poderia muito bem representar árvores genealógicas, como os pedigrees dos cachorros de raça.

vo de uma biblioteca ou em um armazém de peças de trocas.

Um típico caso informático é o constituído pela organização em árvore dos sistemas de gestão de arquivos (adotado sobretudo no sistema operacional UNIX da AT&T, mas também no pro-DOS da Aplle IIc/IIe e no MS DOS do PC-IBM e compatíveis).

Seguramente é a flexibilidade das árvores, o que preferem os informáticos: podemos ir desde a hierarquia top-down nos programas para avaliação dos movimentos de um jogo (xadrez, dama, etc.) à arquitetura de compiladores, para não falar da Inteligência Artificial, setor onde são considerados como os reis.

Na classificação se fala de níveis (um nó filho é de nível inferior ao de seu pai) e de ordem (número máximo permitido de ramificações) definindo-se como "sub-árvore" toda árvore que tenha como raiz um nó distinto ao do cabeça da série (raiz principal).

Vamos nos limitar a uma categoria particular, muito importante, dentro das árvores: as árvores binárias (aquelas nas quais a ordem é dois). Cada nó tem, portanto, como muito, dois filhos, dis-

tinguidos habitualmente como esquerdo e direito. A árvore binária por sua maior simplicidade. Árvores com uma ordem maior que dois podem ser realizadas com as adequadas regras a árvores binárias.

Estas são as operações básicas que definem a uma árvore binária:

- acesso,
- criação,
- verificação.

O acesso consiste em três operações de leitura: raiz, filho direito e filho esquerdo. A criação é realizada partindo de duas subárvores e introduzindo uma raiz nova. A terceira função comprova se uma árvore está vazia ou não.

De forma análoga ao visto para as listas, uma árvore binária pode ser simulada com três arrays associados: o vetor dos valores (nó) e os dos poteiros esquerdo e direito (ramificações). Em BASIC esta é a única complementação possível, enquanto que em Pascal o tipo pointer serve muito melhor para este fim.

Visitar uma árvore

Uma importante noção ao empregar as árvores é a chamada técnica da "visita", que descreve as possíveis modalidades para recorrer todos seus nós.Com as listas se pode proceder somente em dois sentidos; agora são três as possibilidades. Se chamamos R, E, D, respectivamente, à raiz e aos filhos esquerdo e direto, as seqüências de visitas são:

- visita pré-ordem:R-E-D,
- visita post-ordem:E-D-R,
- visita em-ordem:E-R-D.

Referindo-nos à Figura 8, e tendo em conta que um nó que não seja uma folha é identificado pela subárvore do qual é raiz, as três visitas dão lugar às seguintes sucessões:

- pré-ordem: A-B-D-E-C-F-H-I-G;
- post-ordem: D-E-B-H-I-F-G-C-A;
- em-ordem: D-B-E-A-H-F-I-C-G.

Para entendermos melhor: em essência por "visita" é entendido o alcançar de forma consecutiva determinados nós, respei-

tando algumas regras dadas (segunda a técnica de visita); em algumas ocasiões será necessário, antes de chegar a um nó em concreto, varrer em profundidade outros nós, "deixando-os em reserva". Fazendo praticamente, primeiro tem que sair da raiz, depois da qual, por exemplo, em uma visita post-ordem, teremos que baixar até a folha D (Fig.8) antes de iniciar; uma vez recorrida a subárvore D-E-B passaremos (sem "assumí-la") pela raiz e desde ela baixaremos a H para voltar a iniciar a visita. "Deixar em reserva" algo supõe, neste caso, a utilização de uma pilha (stack), possivelmente automática...

Uma aplicação interessante das árvores binárias e suas correspondentes visitas é dada nas expressões algébricas. Referindo-nos à Figura 8b), uma árvore que represente em suas folhas operadores aritméticos e dados pode dar lugar a três notações:

NOTAÇÃO	VISITA	RECORRIDO
prefixa	pré-ordem	* + *5 3 2 - 8 4
posfixa	post-ordem	5 3*2 + 8 4 - *
infixa	em-ordem	(5*3 + 2)*(8 - 4)

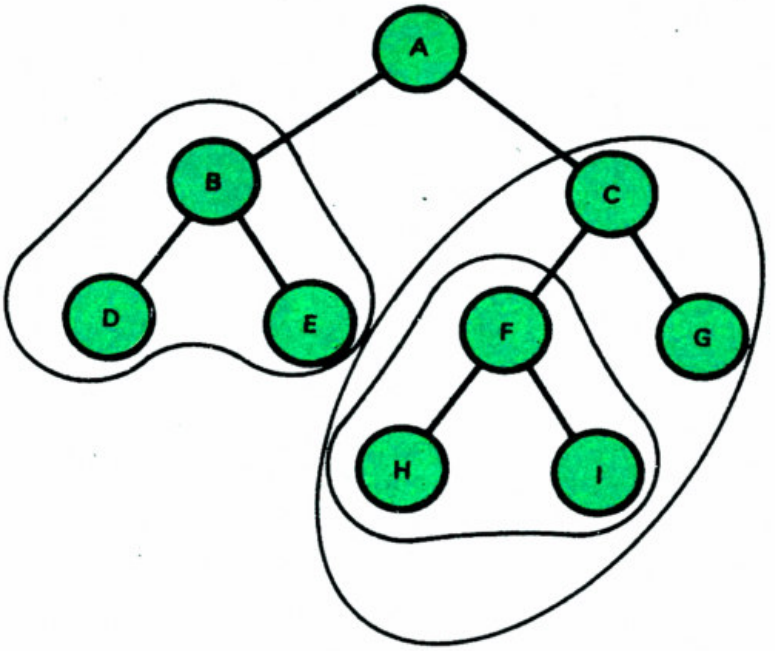
A terceira é a comum, enquanto que a posfixa é a RPN já vista. Na última linha colocamos parêntesis somente para recordar que o resultado vai sendo calculado sem prioridades; será 68, como nas outras duas visitas.

Recursividade: vantagens práticas

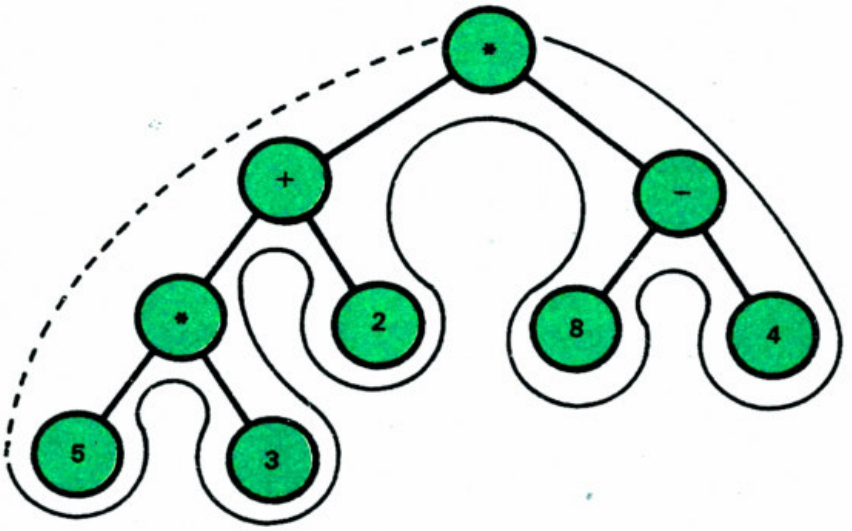
Na Figura 9 apresentamos um programa em Pascal completado desta vez com um pequeno main didático, utilizando para demonstrar as três técnicas de visita no caso da expressão algébrica vista na Figura 8b). A definição do tipo algbin é evidente: com respeito à lista, trata-se somente de um ponteiro mais. Também a FUNCTION Crea tem uma perfeita analogia com a Inser do programa de tratamento de listas da Figura 6, à parte de ter dois apontadores em lugar de um. Depois disto, sugerimos dar uma olhada ao main: na parte introdutória , onde é criada a árvore da Fig.8b), as folhas são definidas nas expressões que contém dois apontadores NIL, enquanto que para os demais elementos Crea segue o mecanismo da visita pré-ordem, adotada com a mais natural:

(+ (5 3) 2) (-8 4)

Uma vez chegados aqui, já se trata somente de um jogo de



a)



b)

Figura 8 — Em a) temos uma árvore binária genérica e em b) o correspondente à representação de uma expressão algébrica. Em ambos os casos podem ser realizados as três técnicas base de "visita". Em a) podem ser apreciadas várias subárvores (de ordem 1) enquanto que em b) a visita em-ordem está representada com linhas de traços para as baixadas ao "filho esquerdo" e linhas contínuas para a visita filho-pai-filho.

```
PROGRAN expresalg;
TYPE algbin =↑elemento:
            elemento = RECORD
                                    valor: STRING;
                                    filho esq: algbin;
                                    filho dir: algbin;
                                    END;
VAR expres: algbin;
FUNCTION crea(val: STRING: fesq, fdir: algbin): algbin;
VAR pt: algbin;
BEGIN
  new (pt);
  pt ↑.valor: = val;
  pt ↑.filho esq: = fesq;
  pt ↑.filho dir: = fdir;
      Crea: = pt;
  END;
  PROCEDURE preordem (a: algbin);
  BEGIN
    IF a < > NIL THEN
      BEGIN
        WRITE (a ↑. valor);
        preordem (a ↑. filho esq);
        preordem (a ↑. filho dir);
      END
    END;
    PROCEDURE postordem (a: algbin);
    BEGIN
      IF a< >NIL THEN
        BEGIN
          postordem (a ↑. filho esq);
          postordem (a ↑. filho dir);
          WRITE (a ↑. valor)
        END
    END;
    PROCEDURE enordem (a: algbin);
    BEGIN
      IF a< >NIL THEN
        BEGIN
          enordem (a ↑. filho esq);
          WRITE (a ↑. valor);



      enordem (a ↑. filho dir);
    END
END;
```

```
BEGIN ( * main * )
  expres: = crea (' * ',crea(' + ',crea('  ',
              crea ('5', NIL, NIL),
              crea ('3', NIL, NIL)), crea ('2', NIL, NIL)),
              crea ('-', crea('8', NIL, NIL),
              crea ('4', nil, nil)));
  preordem (expres); WRITELN;
  postordem (expres); WRITELN;
  enordem (expres)
END.
```

Figura 9 — Listagem do programa Expresalg, que compreende a função de criação de uma árvore binária e a dos três tipos de visita. No main é gerada a árvore de uma expressão algébrica, e é realizada sua visita pelos três modos, imprimindo, para sua verificação, as expressões correspondentes.

paciência: recorrer à função Crea acrescentando os elementos que definem as cadeias em Pascal e as vírgulas que separam os três parâmetros de Crea segundo ('5', NIL, NIL).

Notará imediatamente a comodidade típica do Pascal nas chamadas aninhadas, sem esquecer todas as folhas, como 5, que são traduzidas em Crea das funções (somente são necessários alguns poucos parêntesis), mas isto não é nada com respeito à potência expressiva das auto-invocações recursivas dos três procedimentos Preordem, Postordem e Inordem: a autoinvocação funciona, seja para o filho esquerdo ou o direito, enquanto que a subárvore-parâmetro não seja NIL.

Temos que "acreditar" em tudo o que foi dito sobre recursividade e admirar o fato de que, no exemplo em questão, as três técnicas de visita são diferenciadas somente pela sucessão com que o WRITE (para a impressão do valor do nó alcançado) e as instruções relativas às ramificações esquerda e direita são apresentadas.

Uma aplicação muito interessante das árvores binárias é com os conjuntos de dados "verdadeiros e próprios". O método permite manter um ordenamento de uma forma simples e suficientemente veloz. O algorítmo consiste em criar uma arborescência inserindo sistematicamente os dados inferiores à raiz na subárvore esquerda, e na direita, os dados superiores. Este processamento equivale ao método da bissecção (algorítmo utilizado normalmente para o ordenamento e a busca no array): se opera primeiro com a metade, em seguida com a metade da metade e assim sucessivamen-

te, porque dá melhores resultados que as listas, como já dissemos. De todas as formas, para colocar simplesmente um exemplo, suponha que vão chegando em uma árvore binária, vazia em princípio, os valores seguintes: 17, 12, 5, 80, 10, 35, 8, 6, 11, 94, 27, 24, 20, 48. Se entendeu como funciona o procedimento, tente construir a árvore binária, e depois compare-a com a Figura 10, onde ao lado de cada nó figura entre parêntesis a ordem de chegada. Uma pergunta (será a última): qual dos três tipos de visita - pré,post e en-ordem proporciona uma saída ordenada dos dados em uma árvore binária semelhante? Vejamos a listagem diretamente:

```
PROGRAM inserção;
TYPE algbin = ↑ elemento;
        elemento = RECORD
                        postcp:INTEGER;
                        cidade:STRING;
                        filhoes;filhodir:algbin;
                        END;
VAR albcp:algbin: cp: INTEGER;cid:STRING;
FUNCTION crea...
( * parecida a da fig.8 * )
FUNCTION enordem...
( * parecida a da fig.8 * )
END;
FUNCTION inser (x:INTEGER;y:STRING;a:algbin):algbin;
  BEGIN
     IF a = NIL THEN
      ins: = crea(x,y,NIL,NIL)
     ELSE
      IF x    = a ↑.postcp THEN
        inser: = inser(x,y,a ↑.filhoes)
      ELSE
           inser = inser(x,y,a ↑ .igder)
  END;
BEGIN ( * main * )
  albcp: = NIL;
REPEAT
   READLN (cp,cid):inser(cp,cid,albcp);
  UNTIL cp = 0
enordem(albcp);
END.
```

Foi imaginada a criação com a visita de controle de uma base de dados dos cp (códigos postais) com as correspondentes cidades; conseqüentemente, o registro agora contém dois valores: cp e cidade. Depois, a parte deixada em reserva com os pontinhos é praticamente idêntica à da Figura 9, exceto pequenas variantes que tenham que ver com a nova estrutura do tipo algbin.

Uma vez mais, a FUNCTION Inser é eloqüente. Não é mais que a tradução formalizada e recursiva do algorítmo descrito: se a árvore está vazia cria a raiz, senão Inser(e) o novo par (cp e cidade) lida pelo teclado. Adverte que, graças ao mecanismo recursivo, a condição IF também nos faz sair de Inser, porque o último elemento alcançado encontrou um lugar. De fato, Crea (x, y, NIL, NIL) são adaptados justamente a cada novo nó, raiz de momento sem filhos, mas potencialmente prolífica. Isto é o que significa uma estrutura dinâmica.

Uma breve despedida

Alguém se perguntará: existe um programa BASIC equivalente a este? Desgraçadamente, desta vez não existe nem sequer reflexos de todos os fogos artificiais-recursivos do Pascal. Os caminhos a recorrer poderiam ser os seguintes:

- implantar, segundo já sugerimos em outro lugar, uma pseudo pilha e "enganchá-la" às chamadas recursivas;
- em geral é mais simples emular a recursão com a iteração, e, neste caso específico, o truque consiste, em grandes rasgos em inserir os adequados ponteiros.

Mas ambas as explicações excedem os limites deste livro. Esperamos que aqueles que não tenham a paciência e o tempo necessários para aprofundar um pouco mais nestes temas, pelo menos tenham aprendido algo. Queremos pontualizar algumas questões antes de despedir-nos:

- ultimamente remetemos um pouco a polêmica com os estruturalistas, e inclusive, nos livros mais recentes sobre linguagens de programação, podemos encontrar certa permissividade (em especial na linguagem C)
- no duelo BASIC-Pascal todavia domina o primeiro por dois motivos: o primeiro vai ligado à maior quantidade de programas desenvolvidos; o segundo é a maior flexibilidade do BASIC, sem as obrigações tão formalizadas do Pascal, que, às vezes, podem parecer demasiadamente complicadas aos

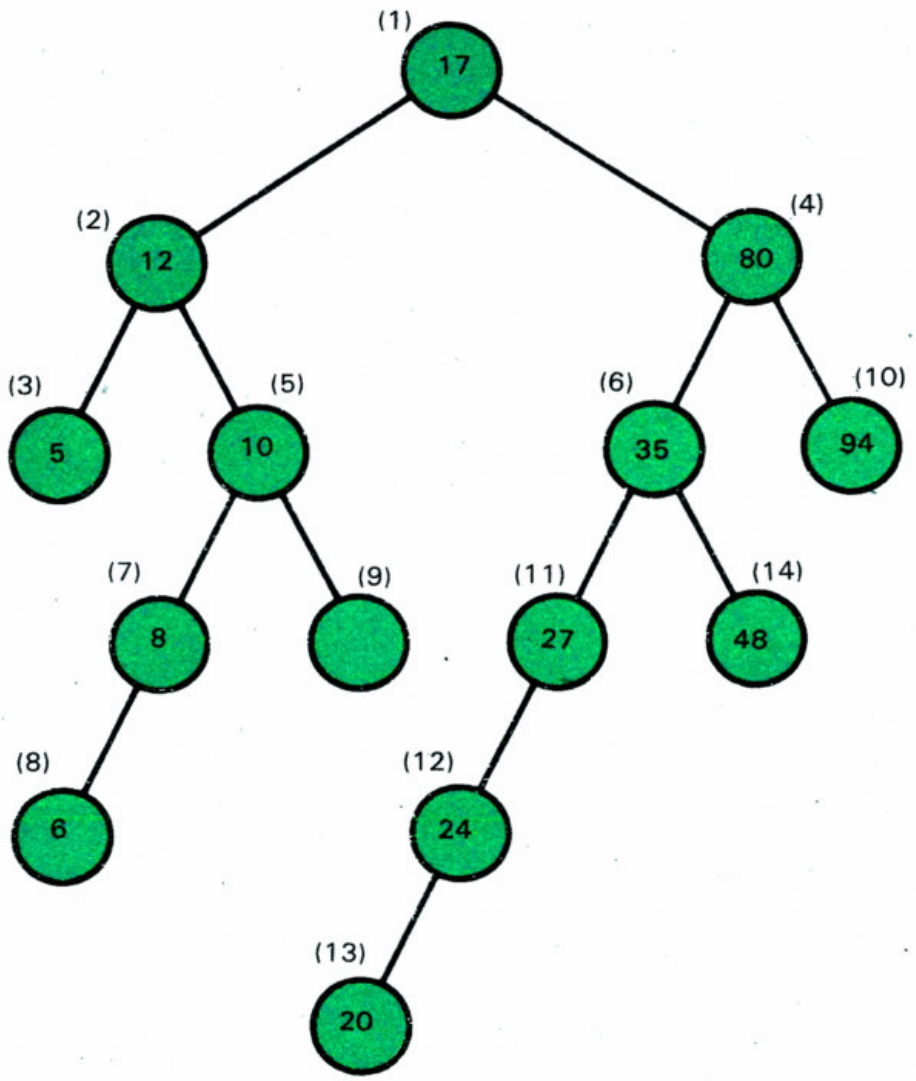


Figura 10 — A figura ilustra como se "dependuram" de uma árvore binária valores dispostos em ordem. Os números externos, entre parêntesis, indicam a ordem de chegada, segundo a sucessão mencionada no texto.

leigos;

- a chegada de BASICs estruturados supõe o triunfo imediato dos princípios da programação estruturada: por exemplo, no superBASIC usado no QL da Sinclair, não falta praticamente nada: construção IF/ELSE (inclusive melhor que a do Pascal, já que tem o terminador ENDIF, que evita certos problemas dos quais já falamos), FOR e REPEAT (com EXIT antecipado de um loop, cuja falta em Pascal é incômoda às

vezes) e procedimentos e funções com variáveis locais e parâmetros.

Sem necessidade de esperar a morte do BASIC tradicional - sempre anunciada, mas ainda longe -, hoje em dia os cânones da programação estruturada são impostos cada vez mais: a gradual inclinação para instrumentos potentes e expressivos é, sem dúvida, sincera por parte de muitos BASICalistas insensíveis.

Falando em termos gerais, esses cânones são a clareza e a precisão, mas autêntica idéia central é a íntima união da estrutura de dados com os procedimentos (algorítmo). Desenvolvê-la, inclusive com os pobres instrumentos das linguagens tradicionais, é quase um dever; oferece grandes vantagens e, bastante satisfações.

BIBLIOGRAFIA

1. Introdução à programação em Pascal
   Findlay
2. Pascal para micros
   James
3. LOGO - Introdução ao poder do ensino através da programação
   e (Goodyear)
4. Inteligência artificial em BASIC
   James
5. Introdução às estruturas de informação
   Lucena
6. Linguagem de programação para micros
   Marshall
7. Dicionário de Informática - Inglês-Espanhol-Francês
   Mansa
8. Pascal for electronics
   Pasahow
9. LOGO for the Apple II
   Abelson
10. Enciclopédia da Linguagem BASIC
    Pereira
11. BASIC prático - Conceitos fundamentais e avançados
    Botelho
12. Computador e programação
    Scheid
13. Introdução à criptografia computacional
    Lucchesi
14. Digital computer fundamentals
    Bartee

15. Sistemas de informação: técnicas avançadas de computação
Cautela

16. Programação estruturada PI/1
Hughes

NOTAS

problema da construção racional e bem organizada de um programa surgiu bastante rápido na história da informática. A pressa por desenvolver uma aplicação por um lado e, por outro, a anarquia dos programadores provocam freqüentemente autênticos desastres, modelados normalmente na escassa legibilidade e a custosa manutenção do software (para não falar dos erros bug, mais ou menos difíceis de localizar).

A programação estruturada nasceu precisamente com o fim de colocar ordem no caos, tratando de impor um estilo e um delineamento claros e metódicos, baseados em regras simples e precisas.

As linguagens estruturadas (das quais, no âmbito dos computadores pessoais, a mais conhecida é o Pascal) são algo muito mais profundo e, por sua vez, mais potente que algumas simples estruturas de controle. Fundamentalmente, se trata da subdivisão em blocos (em níveis hierárquicos) e da definição racional dos tipos de dados. São descobertos assim outros conceitos complexos, mas muito úteis, como a recursividade.

Baseando-nos em Pascal, poderemos entrar na programação estruturada e conseguir, pelo menos, uma estruturação mental que nos permita fazer um programa como se deve.

Cz$ 49,50

B.B.I.

INTRODUÇÃO AO PASCAL