

O Pascal pode aprender-se sem mestre. É uma linguagem de computador concebida para promover um modo lógico e simples de abordar a programação.

Até há pouco, esta linguagem útil apenas equipava grandes computadores e alguns microcomputadores mais caros. Hoje, contudo, existem muitas versões disponíveis na maior parte dos pequenos microcomputadores, tais como o «ZX Spectrum».

Este livro mostra quando é preferível usar Pascal em vez de BASIC e como avaliar as diversas versões da linguagem. Explica o que é «programação estruturada» e as ideias que estão por detrás do Pascal. Mostra também como programar nesta linguagem – com bastantes exemplos práticos e com particular incidência na versão mais vulgarizada em microcomputadores: o Pascal UCSD.

Todos os programas deste livro foram verificados e testados pelo Gabinete Verbo de Informática.

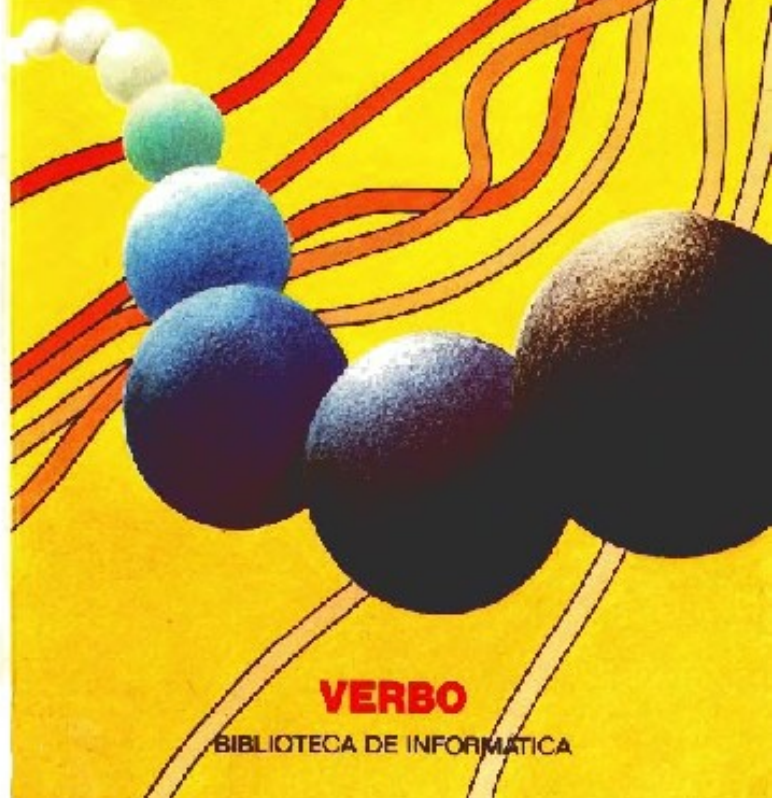


BIBLIOTECA VERBO DE INFORMÁTICA

INTRODUÇÃO AO PASCAL B. ALLAN

# INTRODUÇÃO AO PASCAL

BORIS ALLAN



BORIS ALLAN

# **Introdução ao Pascal**

**Verbo**

# Índice

Prefácio	7
Introdução	11
1 Linhas gerais do Pascal	21
2 O Pascal ao alcance da mão	38
3 Os procedimentos do Pascal	56
4 Tipos em Pascal	85
5 A organização do Pascal	99
6 A sintaxe do Pascal	117
7 Programação em Pascal	127
8 Perspectivas em Pascal	135
Apêndice A: Modula-2, um descendente do Pascal?	159
Apêndice B: Diagramas sintácticos do Pascal	145
Apêndice C: Descrição da sintaxe em AP	165
Apêndice D: Pascal-P e a máquina código-P	173
Apêndice E: Símbolos especiais do Pascal	178
Apêndice F: Procedimentos e funções <i>standard</i>	181
Apêndice G: Bibliografia	186

Título do original inglês: *Introducing Pascal*  
 Tradução do Eng.º Luis Moreira  
 Revisão científica de Paulo Maria  
 © Copyright by Boris Allan 1984  
 Direitos reservados para a Língua Portuguesa  
 EDITORIAL VERBO — Lisboa / São Paulo  
 Ed. n.º 1673  
 Composto por Fotocompográfica  
 Impresso pela Empresa Litográfica do Sul  
 em Janeiro de 1986.  
 Depósito legal n.º 19 501/85

# Prefácio

Pensamentos elevados pedem linguagem elevada.

*As Rãs, de Aristófanes*

Os principais objectivos deste livro são: primeiro, dar noções gerais de Pascal ao leitor que não conheça esta linguagem; segundo, dar ao leitor experiente uma compreensão mais profunda da programação em Pascal.

Estes objectivos não são, de modo algum, incompatíveis. Para que um novato compreenda Pascal adequadamente é necessário que apreenda as suas linhas gerais e os motivos que estão por detrás do seu desenvolvimento. Este tipo de compreensão falta, muitas vezes, naqueles que aprenderam Pascal através de técnicas tradicionais.

O Pascal, enquanto linguagem, tenta encorajar uma abordagem sistematizada da programação: a abordagem «estruturada» da elaboração de programas, normalmente usando o método chamado *top down* («de cima para baixo» ou «do fim para o princípio»). Na abordagem *top down* começamos por uma ideia geral daquilo que pretendemos e, à medida que se desenvolve o programa, vamos pensando nos detalhes.

A programação *top down* é como planear uma viagem. Antes de mais, necessitamos de um mapa com indicações gerais das estradas, para determinarmos os pontos por onde devemos passar.

Para planear a viagem entre os pontos determinados já serão necessários mapas mais pormenorizados. Começar com estes mapas é uma receita para a confusão; saltar-se de mapa para mapa inutilmente, à procura das indicações para traçar o plano da viagem. Se examinarmos a maioria dos textos sobre Pascal, vemos que a linguagem é apresentada de baixo (declarações e mecanismos de comando) para cima (procedimentos e funções).



Muitos textos sobre Pascal começam por dizer como imprimir «Ola» e só depois do meio do texto começam a estudar os procedimentos.

Apesar do empolamento dado à abordagem *top down*, os textos sobre Pascal são normalmente orientados da primeira forma (ou seja, *bottom up*), sendo assim também para muitos cursos sobre Pascal: conhecemos, por exemplo, um curso não superior que apenas começa a estudar procedimentos após o primeiro período.

Receio que um dos motivos por que se divulga a abordagem estruturada (*top down*) é o ser moda «académica» nos meios informáticos; outro motivo é o supor-se que, se um aluno estuda a abordagem estruturada, virá a desenvolver um pensamento estruturado. Por vezes somos levados a pensar que as verdadeiras consequências da abordagem estruturada ainda não chegaram aos professores, pois não ensinam de maneira *top down*.

Julgamos ser o método sistemático (e *top down*) o que faz mais sentido para a programação prática; e também o que faz mais sentido para entender linguagens. Este é, por isso, um trabalho *top down*.

Este livro deve ser lido do início até ao fim, e saltar páginas ou capítulos não resulta. **NÃO FAÇAM ISSO.**

Ser-se adepto do *top down* não significa que se esqueça o outro método e, posteriormente, abordaremos muitas das particularidades do Pascal, quer no texto quer nos apêndices (ao «perito», alguns dos exemplos parecerão quase triviais). O material dos apêndices é bastante variado; no entanto, gostaríamos de chamar a atenção para os apêndices A e D, que julgamos muito importantes.

O apêndice A consiste na exposição breve da linguagem Modula-2, que é um descendente do Pascal; a informação contida neste apêndice não se encontra facilmente em qualquer outro lado. O apêndice D contém uma descrição do Pascal P e da máquina P, informação importante quando se pretenda conhecer o Pascal UCSD (uma variante bastante popular do Pascal). Também neste caso não será fácil obter esta informação noutro lado.

No texto, a falha mais notável reside na brevidade do estudo de

ficheiros e estruturas compactas (*packed*). Ignorámo-las quase na medida em que tão largamente dependem do computador usado. Também prestámos pouca atenção aos registos (*records*) de dados VARIANT, uma vez que julgamos terem pouca aplicação prática.

O impacto da sintaxe (regras de linguagem) e semântica (significado da linguagem) é de grande evidência no Pascal, pelo que introduzimos uma nova maneira de descrever a sintaxe das linguagens. O formalismo Allan (AF) foi desenvolvido longe de outras influências (exceptuando a de Javaid Qureshi), acabando por ficar parecida com a Notação Backus-Naur (BNF).

Pessoalmente, julgamos que o nosso método é melhor, mas haverá quem pense o contrário.

Gostaria de agradecer a Brendon Gore, antigo editor da publicação *Popular Computing Weekly*, a sua permissão para usar o exemplo EGYPT, e também a David Link, da Hisoft, pela sua ajuda e permissão para usar o seu exemplo na descrição de erros.

Samuel Johnson disse que os dicionários são como relógios, pois um relógio defeituoso era melhor do que nenhum, e mesmo o melhor dos relógios por vezes falha. Com os livros de Pascal sucede o mesmo...

Boris Allan

# Introdução

Originalmente, o Pascal tinha o fim pedagógico de ensinar «bons» métodos de programação. A aceitação do Pascal excedeu, no entanto, essa aplicação particular. Teve tanto êxito que é talvez, a seguir ao BASIC, a linguagem mais usada em micro-computadores.

O Pascal, baseado em noções de programação «estruturada» primeiramente expostas por Edger Dijkstra, foi inventado por Niklaus Wirth, professor no Instituto Técnico de Zurique. Havia já tempo que Wirth se preocupava com o desenvolvimento de linguagens mais eficazes, nomeadamente para ensinar programação de uma maneira mais coerente.

Antes de vermos porque é que Wirth desenvolveu o Pascal, será vantajoso ver como se desenvolveram inicialmente outras linguagens. Por exemplo, porque é que existem tantas linguagens?

## PRIMEIRAS LINGUAGENS DE PROGRAMAÇÃO

Os computadores, para efectuarem cálculos (ou acções, de um modo genérico), usam um conjunto especial de instruções de natureza primária, designadas por «código máquina». A programação nestas instruções mostra-se complicada e, portanto, fastidiosa. Para programar um determinado conjunto de acções em código máquina, devemos colocar valores binários em posições de memória, dado que são esses padrões binários (*bits* e *bytes*) que o computador entende para realizar acções básicas.

Para combater em parte o tédio de escrever programas em código máquina foi inventado o *Assembler*. A partir de então, ficou-se com a possibilidade de escrever programas em código máquina de um modo mais simples, dado que assim podemos associar nomes (mnemónicas) a acções e a posições de memória. Essas mnemónicas, no entanto, variam de fabricante para fabri-

cante, pois o código máquina varia de computador para computador.

Em vez de introduzir a sequência binária que representa a ação «colocar o acumulador a zero», introduz-se

LDA # 0

com o mesmo significado nesta linguagem assembly (imaginária). Em código máquina esta ação corresponde à sequência binária

11011000 00000000

que deveria ser colocada em sucessivas posições de memória. Esta operação pode ser bastante causativa, embora o uso de «assembladores» (programas que traduzem as mnemónicas para as sequências binárias correspondentes) acelere muito o processo.

Na década de 50, a IBM encontrou um método mais expedito, a que chamou FORTRAN I, provavelmente a primeira linguagem que não requeria conhecimentos profundos do funcionamento do computador.

Por exemplo, a multiplicação de dois números em código máquina exige rotinas complexas, enquanto em FORTRAN I a multiplicação de 2 por 3 (por exemplo) é representada escrevendo 2\*3. O FORTRAN, cujo nome vem de *FOR*mula *TRAN*slation, é uma linguagem altamente vocacionada para cálculo científico, técnico e numérico.

Na IBM cedo se modificou o FORTRAN I, e a linguagem começou a aparecer em outros fabricantes. O emprego crescente de FORTRAN forneceu o balanço para a introdução do FORTRAN II e do FORTRAN IV, forma que se manteve até hoje. Em 1977 foi anunciada uma versão modificada de FORTRAN, o FORTRAN 77. Existe ainda uma versão de FORTRAN denominada FORTRAN V na série 1100 da SPERRY UNIVAC.

Embora a versão FORTRAN 77 tenha ganho alguma popularidade, os cientistas continuam a usar FORTRAN IV, que é melhor para cálculo numérico. A linguagem interactiva BASIC (*Beginners' All-Purpose Symbolic Instruction Code*), empregada na maior parte dos microcomputadores, baseou-se, em alguns aspectos, no FORTRAN IV.

Pouco depois do início do desenvolvimento do FORTRAN,

diversas organizações científicas ligadas à informática iniciaram uma colaboração visando o desenvolvimento de uma nova linguagem. Pretendia-se que fosse completamente independente da máquina, isto é, que um programa desenvolvido num computador pudesse ser executado em qualquer outro computador, sem modificações.

Sentiu-se que o FORTRAN era muito dependente da máquina, dado que cada tipo de computador necessita de uma versão diferente de FORTRAN (e por vezes mais do que uma versão). A nova linguagem devia reificar essa falta. O modo como o programa controla dentro de si a sua execução foi também considerada primitiva, e pouco útil no FORTRAN. A nova linguagem deveria, pois, ultrapassar este ponto.

A nova linguagem, chamada ALGOL 60, foi primeiramente anunciada em 1958, e o manual-relatório inicial foi publicado em 1960. O nome «ALGOL» era a abreviação de *ALGO*rithmic *LANG*uage; entende-se por «algoritmo» o modo como se trata um problema, isto é, as «regras» para resolver um problema. Para ajudar a linguagem a tornar-se independente da máquina, não se mencionava o modo como o programa comunicaria com o «exterior» (ou seja, informação sobre entrada/saída - *input/output*).

O ALGOL 60 foi um fracasso, uma vez que nunca foi bem aceite fora dos círculos académicos das ciências da informática. Fora destes era muito mais corrente o uso do FORTRAN IV, linguagem à qual modificavam muitas vezes para se ajustar a uma máquina em particular, e que era rápida, tinha boas vantagens de *input/output* e numéricas que faltavam ao ALGOL 60 (por exemplo, aritmética de dupla precisão).

A forma como se procurou tornar independente a linguagem levou a que o ALGOL 60 se transformasse numa linguagem que era um grande ideal mas uma péssima realidade prática. Foi uma infelicidade, pois o ALGOL 60 introduzia muitas ideias agora classificadas como «programação estruturada».

«Programação estruturada» é um termo de escasso conteúdo, significando normalmente que, ao escrever-se um programa, se tenta produzir um esquema limpo, fácil de seguir. Estes

programas serão escritos com facilidade se a linguagem possuir estruturas de controle mais evoluídas.

Diversos interessados escreveram numerosas versões destinadas a maximizar os benefícios «pensados» pelo ALGOL 60. Uma das versões modificadas de maior êxito foi o ALGOL-W (onde «W» vem de Wirth). A versão de Wirth usava muitas das noções básicas do ALGOL 60, mas arrumou mais o *input/output* e melhorou a capacidade da linguagem de empregar tipos complexos de dados. Um grupo tomou uma direcção diferente na preparação da linguagem que ficou conhecida por ALGOL 68.

O ALGOL 68 era completamente diverso do ALGOL 60, e no mesmo tempo em muitos aspectos o ALGOL 68 era «realmente» um ALGOL. As duas ideias mais importantes do ALGOL 68 eram: 1) a linguagem baseava-se na noção de «cláusulas», cada uma com começo e fim claramente definidos, e cada cláusula, quando executada, produzia um resultado; 2) existia a noção de «ortogonalidade», significando que, enquanto os tipos de dados fossem adequados, qualquer operação podia ser aplicada a qualquer cláusula.

De certo modo, como sucedera com o ALGOL 60, a influência do ALGOL 68 foi muito maior do que seria de esperar, partindo do número de pessoas que a empregavam. Detecta-se a influência do ALGOL 68, por exemplo, no FORTRAN 77 e no SUPERBASIC do QL. Mais interessante é ainda verificar que a influência do ALGOL 68 é também bastante clara no Modula-2 (apêndice A).

O Modula-2 foi concebido por Wirth para suceder ao Pascal, e a influência do ALGOL 68 é bastante interessante, dado que Wirth introduziu o Pascal em oposição ao ALGOL 68. Wirth sentiu que o ALGOL 68 era demasiado grande e demasiado complexo e que as linguagens deviam ser simples e directas.

No desenvolvimento do Pascal, Wirth empregou a sua experiência com o ALGOL-W e ideias de tipos de dados imaginados por C. A. R. Hoare (agora professor de Computação na Universidade de Oxford). O ALGOL-W era um melhoramento do ALGOL 60, e o Pascal pretendia remediar erros no desenho original do ALGOL 60.

O Pascal foi originalmente (1970) introduzido num computador *mainframe* de grande porte e grande capacidade de armazenamento, tanto de memória central como de memória periférica, — o CDC 6600 — e tinha por fim aproveitar a vasta memória e a rapidez do processador do 6600. Note-se que estas características são opostas às dos microcomputadores, que possuem memória relativamente pequena e processadores lentos.

Como se verá adiante, o Pascal não é uma linguagem única, existindo, isso sim, muitas «variedades». As duas de que nos iremos ocupar são o Pascal *standard* e o Pascal UCSD. O primeiro é uma versão de Pascal especificada pelas instituições de standardização da Inglaterra e da América e cuja estrutura examinaremos nos apêndices B e C. O Pascal UCSD, variante desenvolvida na Universidade da Califórnia, em S. Diego, é uma versão destinada a microcomputadores; uma das possibilidades introduzidas no Pascal UCSD é a «tartaruga gráfica» (*turtle graphics*), de que falaremos adiante.

#### UM «PUZZLE» PASCAL

Apesar de, inicialmente, ter tido fraca divulgação, o Pascal aparece agora nos lugares mais surpreendentes para uma linguagem «a sério», incluindo a secção de *puzzles* da publicação *Popular Computing Weekly*.

O *puzzle* número 83 (*Popular Computing Weekly* de 24 de Novembro de 1983) apresentava um problema a resolver através de um programa. Consistia em encontrar os valores das letras de A até I (os valores por elas representados), na multiplicação seguinte:

$$\begin{array}{r} A B C D E \\ \times 3 \\ \hline P Q G H I \end{array}$$

(no original, as letras apareciam como símbolos hieroglíficos).

No estudo da solução (*Popular Computing Weekly*, 5 de Janeiro de 1984) notava-se que à letra A apenas podiam corresponder os valores 1 ou 2, e que à letra E apenas se podia atribuir 0 ou 1. Esta informação simplifica até certo ponto o programa que

vai encontrar a solução, mas não é essencial para a solução programada.

A solução foi apresentada sob a forma de um longo programa BASIC, que parecia baseado no «dialecto» Sinclair de BASIC, provavelmente — dado que as declarações apareciam em linhas separadas — para um ZX 81.

```

10 FOR F=1 TO 2
20 FOR E=0 TO 9
30 IF A=E THEN GO TO 280
40 FOR D=0 TO 9
50 IF C=A OR C=B THEN GO TO 27
60 FOR D=0 TO 9
70 IF D=A OR D=B OR D=C THEN 3
80 TO 280
90 FOR E=2 TO 9
100 IF E=A OR E=B OR E=C OR E=D
THEN GO TO 280
110 LET P=(A*10000+B*1000+C*100
+D*10+E)*3
120 IF P%120000 THEN GO TO 25
130 FOR M=1 TO 4
140 FOR N=M+1 TO 5
150 IF P%100000=P%10000 THEN GO TO 2
160 NEXT N
170 NEXT M
180 LET Q=P%10000
190 FOR M=1 TO 5
200 FOR N=1 TO 5
210 IF P%100000=P%10000 THEN GO TO 2
220 NEXT N
230 NEXT M
240 PRINT P%,Q%
250 NEXT P
260 NEXT D
270 NEXT C
280 NEXT E
290 NEXT F

```

Este programa, disseram-nos, fornece a única solução possível,  $17694 \times 3 = 53082$ .

Examinando este programa para verificar como foi construído, observamos que é algo complexo. Mesmo para quem esteja

muito habituado ao BASIC o programa apresenta dificuldades de análise, particularmente se se tratar de pessoa habituada a um computador diferente do ZX 81 ou do ZX Spectrum. Parte da programação é bastante «opaca» e requer um nível elevado de entendimento da arte do programador.

A competição foi, no entanto, ganha por um programa escrito em Pascal (isto é, Pascal 4T da Hisoft para o Spectrum de 48 k). Esta versão da Hisoft é mais uma versão de Pascal disponível em microcomputadores. O Pascal da Hisoft é uma versão bastante completa embora, tal como o Pascal UCSD, ligeiramente não *standard*. O Pascal da Hisoft também dispõe de tartaruga gráfica, como o Pascal UCSD.

Infelizmente, apesar de o programa em BASIC ser um tanto ou quanto rudimentar, o programa vencedor — em Pascal — não é ainda o resumo mais apurado da arte de programar. Apesar disso, é mais simples de entender pelo iniciado (que desconheça quer BASIC quer Pascal), a partir do momento em que se percebe que  $X \text{ MOD } 10$  dá as unidades do número representado por X, e  $X \text{ DIV } 10$  dá as dezenas.

Apresenta-se a seguir o programa vencedor, intitulado EGYPT:

```

PROGRAM EGYPT;
VAR A,B,C,D,E,F,G,H,I : IN
TEGER;
BEGIN
  FOR A := 1 TO 2 DO
    FOR B := 0 TO 9 DO
      FOR C := 0 TO 9 DO
        FOR D := 0 TO 9 DO
          FOR E := 0 TO 9 DO
            IF (A+B*(C*10+D)+E*10
U 20) DIV 10) DIV 10) MOD
10 = 3
            THEN BEGIN
              I := 3+E*100+10;
              H := 13*D+3+E DIV 10) M0
D 10;
              G := 13*D+(C*10+D+3+E DIV 1
2) DIV 10) MOD 10;
              F := (C+B+(D+8+(3*D+13*D
-3+E DIV 10) DIV 10) DIV 1
0) DIV 10) MOD 10;

```

```

IF ABS(10*(A+3000+B+300+
C+30+D+3-F+1000-3+100-G+10
-H)-I+E*3) < 1E-5
THEN
  IF NOT (
    (A IN (B,C,D,E,F,G,H,I))
  OR
    (B IN (A,C,D,E,F,G,H,I))
  OR
    (C IN (A,B,D,E,F,G,H,I))
  OR
    (D IN (A,B,C,E,F,G,H,I))
  OR
    (E IN (A,B,C,D,F,G,H,I))
  OR
    (F IN (A,B,C,D,E,G,H,I))
  OR
    (G IN (A,B,C,D,E,F,H,I))
  OR
    (H IN (A,B,C,D,E,F,G,I))
  OR
    (I IN (A,B,C,D,E,F,G,H))
  OR
    (G IN (A,B,C,D,E,F,G,H,I))
  ) THEN
  WRITELN(A:2,B:2,C:2,D:
2,E:2,F:2,G:2,H:2,I:2)
END
END.

```

Não é nossa intenção examinar minimamente este programa neste momento. Valerá mais a pena prestar-lhe atenção mais adiante, depois de se ter avançado na leitura deste livro.

O programa, não particularmente bem escrito, faz certas coisas de modo demasiado complicado, como, por exemplo, todas as verificações para determinar se um valor se encontra dentro (IN) de um conjunto de valores. Mesmo assim, só necessitou de 30 segundos para encontrar a solução, num ZX Spectrum.

Num Spectrum, o programa BASIC apresentado demorou 177 segundos para encontrar a solução, o que representa cerca de seis vezes mais tempo.

Voltando ao programa em BASIC: não será executado correctamente no Commodore 64, por exemplo. Em determinadas linhas do programa BASIC (por exemplo, na linha 210) existe um salto (GOTO) para fora de um ciclo. Apesar de estes saltos

serem permitidos em algumas versões de BASIC, a maior parte não admite tal situação.

Existem bastantes mais diferenças, tais como o MID\$ (ver adiante), que podem gerar alguma confusão durante a execução do programa.

Fazendo estas concessões, convertemos o programa para correr num microcomputador BBC, e a solução, em BASIC do BBC, apareceu após 67 segundos. O BASIC do BBC é bem conhecido pela sua rapidez, e de facto é perto de duas vezes mais rápido que o BASIC do Spectrum, para este problema específico. O BASIC do BBC, no entanto, demora mais do dobro do tempo de que necessita o Pascal da Hisoft do Spectrum «mais lento».

## PASCAL «STANDARD»

Qualquer pessoa que tenha aprendido Pascal será capaz de compreender o programa apresentado, dado que o Pascal é uma linguagem com regras *standard* que, com pequenas variações, são comuns a todas as versões. Quando o Pascal foi ideado, teve-se como um dos principais requisitos a standardização da linguagem (apesar de só recentemente se ter estabelecido um *standard* oficial).

Dado que o Pascal foi ideado por uma pessoa apenas, Nicklaus Wirth, tem uma coerência que falta em outras linguagens que foram ideadas por equipas.

Enquanto do BASIC existem muitas versões, cada uma com a sua visão da linguagem, existe apenas efectivamente um Pascal (embora obviamente existam algumas diferenças). Um grande subconjunto desta linguagem é o Pascal P (a base do Pascal UCSD, entre outros), razão por que damos nota do Pascal P e do código máquina P no apêndice D.

Considere-se esta linha do programa BASIC anterior:

```
150 IF P$(M)=P$(N) THEN GOTO 250
```

que significa «se o *Mésimo* carácter da cadeia (*string*) P\$ for igual ao *Nésimo* carácter, vá para a linha 250». Para a grande maioria dos BASICs, isto significaria «se a *Mésima* cadeia da matriz de



cadeias P\$ for igual à Nésima cadeia, vá para a linha 250. Existe aqui uma grande e importante diferença.

Em muitos BASICs, a linha 150 apresentaria o seguinte aspecto:

```
150 IF MID$(P$,M,1)=MID$(P$,N,1) THEN 250
```

(por exemplo, muitos BASICs do estilo Microsoft, o BASIC do BBC para o modelo B ou o Electron.)

Uma das principais vantagens do uso de Pascal, para o utilizador vulgar de microcomputadores, é a velocidade de execução dos programas em Pascal. O programa Pascal exposto não está escrito da forma mais eficiente, mas no entanto é bastante mais rápido que o seu equivalente em BASIC.

No entanto, escrever um programa em Pascal não é mais difícil que escrevê-lo em BASIC, e mesmo a execução de um programa «pobre» (mal escrito) em Pascal é muito mais rápida que a do programa em BASIC. Neste caso, é mais difícil escrever o programa em BASIC, dado que o Pascal tem aspectos úteis, como a construção IN.

Uma outra vantagem bastante divulgada do Pascal é que ele encoraja o desenvolvimento de «melhores» estilos de programação. Esses melhores estilos são geralmente conhecidos sob o título de «programação estruturada», como já se disse, mas sentimos que é mais fácil uma programação pobre e ineficaz em Pascal que noutras linguagens.

De facto, como o Pascal é em alguns sentidos uma linguagem mais poderosa, existe grande tentação de usar o poder da linguagem para disfarçar raciocínios desastrosos. Temos um exemplo bastante bom no programa EGYPT, que apresenta diversas pequenas falhas. A melhor maneira de resolver o puzzle não é abordada nem pelo programa em BASIC nem pelo programa em Pascal; de facto, o programa Pascal ajusta-se quase perfeitamente ao programa BASIC.

Voltaremos a usar o exemplo deste problema, empregando o Pascal para o solucionar. À medida que progredirmos mostraremos como nos podemos servir do Pascal para produzir respostas concisas a este pequeno mas interessante puzzle.

## Capítulo 1

# Linhas gerais do Pascal

Um lugar para cada coisa e cada coisa em seu lugar

*The Book of Household Management*, de I. M. Beeton

Éis o programa Pascal mais simples que se pode escrever:

```
PROGRAM SIMPLES (INPUT, OUTPUT);  
BEGIN  
END.
```

e este programa contém tudo sobre Pascal.

Na primeira linha dá-se ao programa o nome SIMPLES (também conhecido por «identificador»). A seguir ao nome do programa, estão mais dois identificadores (entre parênteses) que referem os nomes dos ficheiros de entrada e saída que o programa vai usar. A linha termina com «;», que, em Pascal, é o modo normal de terminar uma secção do programa.

PROGRAM é uma palavra reservada em Pascal, e serve apenas para indicar o nome do programa. Em alguns textos, as palavras reservadas — ou «palavras chave» — são apresentadas em maiúsculas e em impressão reforçada para as distinguir dos identificadores vulgares.

Uma vez que frequentemente os sistemas para programação em Pascal não dispõem de meios de distinção entre maiúsculas e minúsculas, pode ser confuso mostrar as palavras reservadas em maiúsculas.

Neste livro, apresentamos em maiúsculas quer as palavras reservadas quer os identificadores vulgares. Devemos lembrar, no entanto, que apesar de alguns sistemas necessitarem do texto todo em maiúsculas ou minúsculas, existem sistemas que aceitam os dois tipos.

Logo a seguir ao identificador do nome do programa vêm os identificadores dos nomes dos ficheiros de entrada/saída de

dados. Em algumas versões de Pascal estes identificadores não são necessários, usando-se por omissão os ficheiros de *input* e *output* standardizados (geralmente o terminal de vídeo, mas também a impressora para *output*, leitura de cartões para *input*, etc.; note-se, por exemplo, que o programa EGYPT não tem estes identificadores).

Em algumas versões mais antigas do sistema Pascal UCSD, os parâmetros «nome de ficheiro» eram proibidos, embora a intenção fosse a de vir a usar nomes de ficheiros mais tarde. Na versão *Apple* do UCSD pode ser dada uma lista de parâmetros com nomes de ficheiros a seguir ao nome do programa, mas o sistema ignorá-los-á (UCSD é o acrónimo de *University of California at San Diego*, onde se desenvolveu esta versão de Pascal para microcomputadores).

Uma vez que pretendemos essencialmente usar uma versão aplicável de um modo geral da linguagem de programação Pascal, usaremos o Pascal *standard* (BS 6192/ISO 7185)<sup>1</sup>.

No Pascal *standard*, os identificadores dos ficheiros não apresentam opções, e por isso os colocamos neste programa. No entanto, num programa sem entrada de dados e apenas com saída, basta declarar o ficheiro *OUTPUT*.

Na segunda linha do programa está a palavra *BEGIN* (palavra reservada) que indica o início de um segmento discreto no programa. O nome dado a um segmento discreto é «bloco». Um bloco é uma porção de programa com as suas próprias (e facultativas) declarações de variáveis e linhas de instruções, compreendidas entre as palavras reservadas *BEGIN* e *END*. No caso do bloco do programa anterior, não existem declarações de variáveis.

A terceira linha do programa termina o bloco com *END* e, uma vez que é aqui também o fim do programa, temos igualmente o terminador «.». Juntam-se os dois («*END*» e «.») para dar «*END.*», terminando-se assim o bloco e o programa principal.

Dentro da norma BS 6192, define-se a estrutura de um programa como uma sequência do tipo:

- (a) cabeçalho do programa (*program heading*)
- (b) «;»
- (c) bloco do programa
- (d) «.»

### Maior complexidade

Podemos aumentar a extensão de estrutura simples do programa, fazendo-se por exemplo.

```
PROGRAM MENOS_SIMPLES (OUTPUT);  
VAR NUMERO : INTEGER;  
BEGIN  
    NUMERO := 2;  
    WRITELN (NUMERO:3)  
END.
```

Relativamente à sequência indicada pela BS 6192, o cabeçalho do programa e «;» estão contidos na primeira linha, o bloco do programa vai desde a segunda linha (que começa com *VAR*) até à última linha (*END*) e coloca-se o terminador «.» junto do «*END*».

A segunda linha é uma declaração de variável. Declara-se que a variável *NUMERO* é inteira (isto é, *NUMERO* só pode conter valores inteiros).

A porção de programa entre *BEGIN* e *END* usa variáveis anteriormente declaradas (neste caso apenas uma, *NUMERO*). A variável inteira *NUMERO* é atribuído (feito conter) o valor 2. A declaração de atribuição segue-se «:=», o que significa «fim de uma declaração Pascal, passemos à seguinte».

A declaração (*statement*) seguinte escreve o valor contido em *NUMERO* (que é 2). Não se especifica nenhum ficheiro na instrução *WRITELN*, pelo que, por omissão, a saída é feita no ficheiro *OUTPUT* (que pode ser um *écran* de vídeo, uma impressora ou um ficheiro em disco, o que dependerá da implementação). O valor que a instrução *WRITELN* vai escrever ocupa um comprimento de três caracteres (embora nem todos sejam usados neste caso. O número 2 é escrito precedido por dois

<sup>1</sup> A I.S.O. (International Standard Organization) é uma organização internacional com a função de redigir normas, que na realidade são conselhos aos fabricantes. (N. do T.)

espaços).

A declaração `WRITELN` não é completada com «;» uma vez que o identificador seguinte não é um identificador vulgar, mas sim «END» (contudo, um «;» a mais não faz mal nenhum).

O programa seguinte está incorrecto.

```
PROGRAM ERRORE (OUTLET);
ESTE PROGRAMA NAO FUNCIONA!
BEGIN
  NUMERO:=2;
  WRITE('NUMERO:3');
END;
```

Não funciona porque o sistema Pascal não foi informado quanto à variável `NUMERO` antes que esta fosse usada. A menos que o Pascal seja informado da existência e natureza de um identificador (nome da variável, nome do ficheiro, ou o que for) não aceita o identificador e gera uma mensagem de erro.

A linha que continha a declaração de variável contém agora uma linha de comentário, que pode ser escrita das seguintes maneiras:

```
ESTE PROGRAMA NAO FUNCIONA!
(*ESTE PROGRAMA NAO FUNCIONA!*)
```

Muitos dispositivos de entrada não dispõem de chavetas — entre elas, por exemplo, a maior parte dos teclados de computador —, pelo que, em seu lugar, é possível usar «\*» para começar o comentário e «)» para o terminar.

Por exemplo, as duas declarações,

```
NUMERO:=1000;
NUMERO:LA DEVERIA ESTAR DEFINIDO
2:=1000;
```

são consideradas equivalentes pelo sistema Pascal. Os comentários apenas estão presentes para ajudar à compreensão; não se encontram ali para orientar a execução do programa. Contraria-

mente ao BASIC (por motivos que avaliaremos mais tarde) o uso de comentários em Pascal não atrasa a execução do programa.

## DIAGRAMAS SINTÁCTICOS

Demos atrás a forma correcta de um programa Pascal (ou seja: cabeçalho, «;», bloco de programa e «end») e observa-se este formato no denominado «diagrama sintáctico do Pascal» (ou, por vezes, «diagrama de estados»). Os diagramas sintácticos são apresentados na totalidade no apêndice B mas, por agora, examinemos a figura 1.1.



Fig. 1.1.

Comecemos pelo lado esquerdo, pela palavra `PROGRAM`. Vemos esta palavra dentro de uma linha fechada com cantos arredondados, para dar ênfase ao facto de «PROGRAM» ser uma palavra reservada. A palavra `PROGRAM` aparece tal como a vemos no diagrama. A seguir ao «envoltório» de `PROGRAM`, seguem-se as linhas nas direcções e sentidos indicados pelas setas. Assim, e após `PROGRAM`, espera-se por um «identificador», que dará o nome do programa. O «identificador» é um conceito de Pascal, que recebe uma definição inequívoca dentro da linguagem. O conceito de «identificador» aparece dentro de um rectângulo para salientar a ideia importante de que o nome do identificador pode variar; programas diferentes serão identificados por nomes diferentes.

Imediatamente a seguir ao identificador está um parêntese esquerdo «(» dentro de um círculo, o que indica que aparece como o vemos. A seguir, temos outro identificador, a primeira das designações de ficheiros.

Deixando o rectângulo do identificador e indo para a direita, há dois caminhos possíveis: um deles é um ciclo; o outro, um parêntese direito «)». Se existe mais do que uma designação de ficheiro, seguimos pelo ciclo. Se o fizermos, encontraremos o

símbolo «:», indicando que deve existir uma vírgula entre cada dois identificadores.

Caso existam mais do que dois identificadores, o ciclo repete-se até se processar o último, prosseguindo depois para o parêntese direito. Vemos, pois, que as designações de ficheiros se encontram entre parênteses e separadas por vírgulas.

Após o parêntese direito, temos o carácter «:» dentro de um círculo, antes de se encontrar a entidade conceptual denominada «bloco», que vemos dentro de um rectângulo. Finalmente, aparece o terminador «.».

Atribui-se a estes diagramas o nome de «sintácticos» porque dão as regras para a construção correcta de «frases» em Pascal. A sintaxe é um outro nome de «gramática», e os diagramas sintácticos indicam o modo como se verificam as declarações em Pascal relativamente à correcção gramatical.

A figura 1.2 mostra a sintaxe diferente no Pascal UCSD. Como se observa, não há ciclos ou complexidades para a versão UCDS (e menor estandardização?)

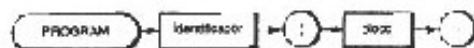


Figura 1.2. Um programa de Pascal UCSD.

## SINTAXE E SEMÂNTICA

Muitas pessoas que falam linguagens vulgares produzem expressões gramaticalmente incorrectas, mas mesmo assim as outras pessoas compreendem-nas. Em linguagens de programação, havendo uma expressão (digamos, uma linha de programa) gramaticalmente incorrecta, o computador *não* entenderá o significado dessa linha.

Entre «significado» e «gramática» existe uma distinção crucial para linguagens de computador, mas não tão importante para linguagens de comunicação verbal vulgares. Por vezes, nestas linguagens, a diferença entre sintaxe e semântica é o elemento fundamental de certo tipo de humor. Um trocadilho, um jogo de

palavras e certas anedotas dependem da habilidade para extrair mais de um significado de uma mesma expressão.

O estudo do significado de uma expressão num discurso vulgar e do significado de uma linha num programa de computador recebe o nome de «semântica». Assim, o estudo de uma linguagem de computador é o conjunto do estudo da sua sintaxe e da sua semântica.

Podemos considerar «naturais» as linguagens faladas, vulgares. Com linguagens naturais, as regras nem sempre são explícitas (como o são no Pascal) e não existe uma forma obrigatória de inglês, por exemplo. As linguagens naturais não são planeadas (como as linguagens de computador) e evoluem. Uma das poucas linguagens humanas que teve de ser planeada (em vez de evoluir, simplesmente) foi o esperanto. A sintaxe do esperanto é tão regular e fixa que esta é talvez a única linguagem humana facilmente computorizável. Não se considera o esperanto uma boa linguagem, quanto à flexibilidade de significados, e isto possivelmente devido à sua rígida sintaxe.

Muitos programas para computador são mal escritos porque o programador usa bastantes vezes na escrita de programas as fórmulas que emprega ao falar a sua língua nativa. É notório que a habilidade para «inventar» e a falta de precisão, aceitáveis numa linguagem oral, já não o são tanto para programar um computador.

Dado que o estudo da gramática se está a tornar cada vez menos importante no ensino actual das línguas, inúmeras vezes os programadores novatos, iniciados, não compreendem a importância da sintaxe na sua própria linguagem falada, quanto mais numa linguagem de programação. Na fala é possível usar a linguagem de modo impreciso (facto especialmente notável na língua inglesa) e, mesmo assim, ser entendido, mas isto já não acontece com um computador.

Existem, pelo menos, cinco razões principais para que um computador tenha dificuldades em entender uma linguagem natural, e cada uma destas razões explica porque é que as linguagens de computador, para serem eficazes, devem ser tão diferentes. As cinco razões também explicam porque não é

factível a programação de um computador em «inglês» (e também noutras línguas humanas) e porque é que as linguagens de computador que sejam do tipo da língua inglesa estão, provavelmente, condenadas ao fracasso.

**Razão 1.** O problema do tamanho e complexidade da sintaxe das linguagens naturais (assumindo que conhecemos essa sintaxe).

**Razão 2.** O problema da ambiguidade, já mencionado.

**Razão 3.** A necessidade de possuir uma grande dose de «senso comum» para entender mesmo o mais simples dos textos. Cada pedaço do texto deve ser colocado num dado contexto, embora alguns ramos da crítica literária digam que isso não é necessário. Vale a pena considerar a quantidade de «senso comum» necessária para entender uma história contada por uma criança.

**Razão 4.** O problema de número de frases que podem ser analisadas em termos do «significado» de partes da frase e, mais ainda, como instruir o computador para entender o significado dos «significados». *Esta razão é importante, por isso lembremo-nos dela.*

**Razão 5.** O problema de tempo e memória. Para que um computador entenda o conteúdo e arranjo mesmo do parágrafo mais pequeno que se refira à razão 4, é preciso grande quantidade de memória e muito tempo para escrever o programa que o vai fazer. É muito difícil de escrever um programa de computador para desempenhar estas tarefas, dado que requer técnicas de pesquisa e codificação bastante complicadas e inteligentes.

#### A APROXIMAÇÃO DO PASCAL À SINTAXE

A sintaxe assume uma importância suprema em Pascal. É crenga geral que, se a sintaxe de um programa estiver correcta, é maior a probabilidade de que a semântica também esteja correcta — ou seja, o programa fará aquilo que se pretende. Em Pascal, a escrita incorrecta de nomes de variáveis (quer por erro quer por falta na escrita) têm menor probabilidade de causar consequências desas-

tradas, dado que o verificador de sintaxe não a admitirá.

Depois de se examinar de perto um programa em Pascal, tomar-se-á evidente que Nicklaus Wirth ideou esta linguagem de programação para que a máquina ultrapasse os cinco problemas de tradução de qualquer linguagem citados na secção anterior.

Antes, contudo, veja-se como o Pascal é completamente diferente do BASIC no tocante ao fim e à orientação. Considere-se, por exemplo, o seguinte programa em BASIC.

```
10 INPUT COLOUR
20 PRINT COLOR
```

onde fazemos entrada de um valor que será armazenado na variável COLOUR, e imprimimos o valor contido pela variável COLOR. De acordo com a versão de BASIC utilizada, teremos a impressão (escrita) de um zero (0), o valor que introduzimos ou a informação de erro, que indica a existência de uma «variável não definida».

Em Pascal, este programa teria de ser escrito na forma sintacticamente incorrecta:

```
PROGRAM ERRO_DE_ESCRITA_COLOUR;
CONST
  TRUE;
VAR COLOUR : INTEGER;
BEGIN
  READLN(COLOUR);
  WRITELN(COLOUR);
END.
```

Antes de podermos executar este programa, ele devia ser corrigido para se obter a sintaxe correcta. A sintaxe não está correcta devido à existência da linha

WRITELN (COLOR);

que inclui a variável COLOR, não indicada ao sistema através de uma declaração VAR (ou seja, o identificador não foi «declarado»). Vale a pena salientar que o programa em BASIC tem apenas 3 linhas (duas, sem o END), enquanto o equivalente em Pascal tem seis entidades distintas. Usamos o termo «entidades»

para cobrir termos díspares tais como «BEGIN» e «VAR COLOUR : INTEGER».

Escrever um programa em Pascal é mais complexo do que escrever um programa em BASIC: como recompensa, quando o programa corre correctamente, é mais provável a execução correcta do que se pretende. Neste ponto passa a ser importante a distinção entre sintaxe e semântica. Se bem que um programa pareça (e esteja) sintacticamente correcto, porque não são indicados erros, o significado do programa (a semântica) pode estar incorrecto.

O programa BASIC anterior não produziria erros em muitas versões desta linguagem (ou seja, a sintaxe pareceria correcta), mas o programa não seria executado correctamente (ou seja, a operação desempenhada pelo programa não seria a pretendida). Um erro comum em BASIC é colocar a letra «L» minúscula em vez do número 1: em Pascal isto será normalmente indicado como um erro antes de o programa ser executado.

O BASIC está, de certo modo, mais perto de uma linguagem natural, dado que a sua sintaxe é menos rígida. É muito possível que o autor de uma expressão incorrecta numa linguagem natural diga algo que não pretendia dizer e até talvez gere confusão. É também possível que uma pessoa não queira dizer o que disse e mesmo assim outra pessoa entenda aquilo que na realidade ela pretendia dizer.

É mais ou menos fácil produzir uma linguagem de computador que possa «dizer aquilo que não se queria dizer» (por exemplo, o programa com a troca COLOR/COLOUR) mas é mais difícil produzir um sistema que corrija os erros de significado. Uma sintaxe vaga produzirá semânticas disparatadas. O computador não pode interpretar ambiguidades, dado que deve seguir uma via, e essas ambiguidades serão ignoradas.

A razão 2 diz que é muito difícil a um computador lidar com ambiguidades numa linguagem de programação. O necessário é, pois, uma linguagem de computador que limite a zero o nível de ambiguidade, reduzindo ao nível mais baixo possível a probabilidade de ser executado um programa incorrecto.

Obviamente, o computador deve possuir um programa, nor-

malmente chamado «compilador» no caso do Pascal, para traduzir o significado de um programa escrito numa linguagem de programação para uma linguagem que o próprio computador entenda.

Usaremos o termo genérico «tradutor» para referir um programa de computador com a função de traduzir um programa escrito numa dada linguagem para a linguagem da máquina (vulgarmente chamada «código máquina»).

A necessidade de reduzir o tamanho e a complexidade da sintaxe de uma linguagem, antes de o computador «enfrentar» a tradução, é muito importante no exame das razões que estão por detrás do desenho do Pascal (ver razão 1). Há necessidade da maior compactação e simplicidade da sintaxe, implicando uma relativa restrição no alcance da linguagem, para que seja fácil a tradução para código máquina por parte do computador.

Usando diagramas sintácticos e outras formas de verificar a geração de erros desconhecidos pelo programador (por exemplo, a troca COLOR/COLOUR), espera-se que os programas em Pascal sejam mais capazes de produzir os efeitos desejados (ou seja, é mais provável que resulte uma semântica correcta).

## PRODUÇÃO DE UM PROGRAMA EM PASCAL

Normalmente, para executar um programa em BASIC, introduzimos no computador o programa (código fonte) e de seguida digitamos RUN. Se o programa não é correctamente executado (se produz, por exemplo, um tipo qualquer de erro), pesquisaremos a origem da falha. Por vezes, o programa corre mas não faz aquilo que se pretendia e, nesse caso, o que está errado e deve ser revisto é a lógica do programa.

Noutras ocasiões, ainda, não é a lógica do programa que está errada; em vez disso, terá havido falhas na digitação do programa, tal como aconteceu ao não se escrever correctamente COLOUR. Com certos BASICs não pudemos empregar algumas palavras como variáveis (COLOUR, por exemplo), e o programa apenas reconhece as duas primeiras letras do nome da variável. Nesse caso, surpreendentemente, o programa incorrecto passa a



estar correcto dado que o sistema BASIC — mais primitivo — interpreta COLOUR e COLOR como CO.

Se estivermos restringidos apenas a duas letras (o que é pouco útil), será muito difícil, ao corrigirmos o programa, estabelecer onde ocorreram os erros. Sucede demasiadas vezes que mesmo quem escreveu o programa tem dificuldade em compreender porque é que o programa funcionou ou não.

O Pascal torna muito mais difícil a criação de um programa *ad hoc*. Efectivamente, o programa em BASIC é escrito num determinado *environment*<sup>1</sup>, enquanto que um programa em Pascal passa, normalmente, por um conjunto de estágios antes de, por fim, poder ser executado.

Em primeiro lugar, devemos introduzir o código fonte e, durante a produção de um programa sintacticamente correcto, o processo de organização já está presente. Uma vez que no Pascal se deve informar da existência de todos os identificadores antes do seu uso, haverá grande volume de trabalho já efectuado (o revisto) no papel, antes da produção do programa.

Em segundo lugar, devemos guardar num ficheiro esse código fonte, para mais tarde o retomarmos para correcções.

Em terceiro lugar, o programa em código fonte deve ser tomado e convertido em instruções que o computador entenda (por outras palavras, o processo de tradução). Num processo chamado «compilação», verifica-se quanto à correcção a sintaxe do programa e, se estiver correcta, traduzem-se para código máquina as instruções do programa, gerando o programa em «código objecto». O código objecto é normalmente armazenado como ficheiro do computador.

Como passo final pegamos no ficheiro com o código objecto e executamo-lo. Há maiores probabilidades de que o programa execute aquilo que se pretendia: primeiro, devido ao rigoroso

trabalho requerido inicialmente, ao escrevermos o programa; segundo, devido ao rigoroso controle do tradutor Pascal, antes que o executemos.

Vemos esta sequência na figura 1.3, tendo nós usado o sistema UCSD do *Apple II* como exemplo:

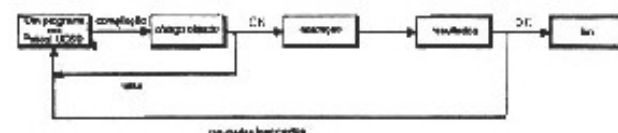


Fig. 1.3

Repare-se que, de modo geral, para os sistemas UCSD os estágios podem ter nomes diferentes, mas o significado global é o mesmo. No *Apple II* existem certos comandos (no NPC, nível principal de comandos, em inglês OCI — *Outermost Command Level*) que gerem o uso de ficheiros e da execução de programas.

Esses comandos são:

**E(DIT)** Premindo E no NPC (nível principal de comandos) carrega-se em memória o programa editor. É no sistema editor que se cria o programa fonte.

**F(ILE)** Premindo F no NPC faz-se correr o programa «Filer», que permite a manipulação de ficheiros em disco (ou *diskette*), incluindo o *workfile*<sup>2</sup> que resulta do uso do editor.

**C(OMP) Usar C** no NPC indica a pretensão de usar o compilador de Pascal. Converte-se um ficheiro fonte em instruções de código máquina, e coloca-se num ficheiro de código objecto. Detectando-se erros durante a compilação, estes são indicados e não se gera o código objecto.

<sup>1</sup> Este termo é muito difícil de traduzir em poucas palavras, por referir todo um contexto. No entanto poderá dizer-se que *environment* (ambiente) diz respeito às condições de trabalho do programador e do funcionamento dos programas, no que toca ao computador. Referencia, então, o sistema operativo, editor de texto, tradutor, etc. (V. de T.)

<sup>2</sup> *Workfile*: ficheiro de trabalho; em UCSD, o *workfile* contém, em princípio, a última edição do texto afechada. Ao fazer E(DIT), os sistemas começam por verificar se existe *workfile*, se existir, carrega-o (V. de T.)

**X(ecute)** O comando X no NPC vai buscar um ficheiro especial de código e executar as instruções em código máquina que ele contém; não deve ser confundido com o comando seguinte.

**R(UN)** Este comando do NPC executa, em sequência, os comandos C(OMP) e X(ecute). Temos, pois, que o uso deste comando faz com que um ficheiro fonte determinado (especificado por nós) seja compilado e, se não existirem erros de compilação, seja executado o código objecto respectivo.

Existem outros comandos, mas descrevemos os básicos para a execução de programas simples em Pascal (programas que não usem segmentos em código máquina e não usem rotinas especiais da «livraria» do sistema).

Vale ainda a pena salientar que a diferença entre sintaxe, semântica e desenho do programa, a que o Pascal atribui maior importância, está bem patente no *Sinclair QL*, no *Spectrum*, no *ZX 81* ou no *ZX80*, dado que cada linha do programa introduzido é verificada a nível sintáctico antes de ser aceite.

Se a linha está sintacticamente correcta, é entregue ao programa; se está incorrecta, é rejeitada, sendo dada indicação de erro. Se existir um programa em BASIC num ficheiro do *Sinclair QL* (SuperBASIC), a sintaxe está verificada, dado que cada linha é introduzida com a própria máquina. Se está incorrecta, é inscrita a palavra MISTAKE entre o número da linha e o conteúdo desta.

Se bem que possam existir apenas linhas sintacticamente correctas num programa, por exemplo, do *Spectrum*, é muito fácil que falhe. Os programas falham quer porque as variáveis têm os seus nomes erradamente escritos ou, mais provavelmente, porque o desenho do programa está completamente errado. Um outro fim do Pascal (e talvez o mais importante) é ajudar a escrever programas que funcionem, dado que o desenho do programa é ajudado pelo estilo da linguagem.

## O USO DE UM INTERPRETADOR

O Pascal é uma linguagem compilada, sendo por isso de execução rápida. O BASIC é uma linguagem interpretada, correndo os programas mais vagarosamente. Quando se faz RUN num sistema BASIC, o efeito é completamente diferente do efeito no sistema *Apple Pascal*.

Num programa BASIC, o interpretador BASIC começa na primeira linha do código fonte, convertendo essa linha em instruções de código máquina. Se a linha não constitui uma declaração BASIC válida, gera-se uma mensagem de erro de sintaxe.

Quando o código máquina respeitante a essa linha tiver sido executado, o interpretador BASIC passa para a próxima linha do programa<sup>1</sup> (entrando em linha de conta com instruções GOTO ou similares), esquecendo a linha que acabou de traduzir. No programa BASIC, de cada vez que o interpretador de BASIC se

```
10 FOR I=1 TO 1000
20 PRINT "H# "
30 NEXT I
40 STOP
```

encontrar na linha 20, as instruções BASIC dessa linha são convertidas nas correspondentes instruções em código máquina. A tradução da linha BASIC em código máquina não é memorizada ao passar de linha para linha.

O que sucede com o interpretador BASIC é que a primeira linha do programa é examinada (vê-se seguida de \*~ a linha para a qual o interpretador faz uma conversão para código máquina).

<sup>1</sup> Os programas em BASIC são sempre executados a partir da linha com mais baixo número, o em ordem crescente desta, com RUN. Existe normalmente a possibilidade de executar um programa a partir da linha N, fazendo RUN N, (N, do T.)

```

10 FOR I=1 TO 1000  *
20 PRINT "H"
30 NEXT I
40 STOP

```

e, assim que essa linha é traduzida, guardará informação acerca do fim do ciclo, o tamanho do «passo»<sup>1</sup> e o nome da variável do ciclo, num troço de memória denominado *stack* (pilha).

A linha do programa a executar a seguir é então traduzida para código máquina, esquecendo-se o conteúdo da linha anterior (neste caso a linha 10), à parte a informação guardada na pilha.

```

10 FOR I=1 TO 1000
20 PRINT "H"
30 NEXT I
40 STOP

```

A tradução da linha 20 significa que os dois caracteres ( ) devem ser escritos (no *screen* ou em qualquer outro dispositivo). O interpretador BASIC traduz então a linha seguinte, ou seja, a linha 30.

```

10 FOR I=1 TO 1000
20 PRINT "H"
30 NEXT I
40 STOP

```

Na linha 30 a instrução BASIC indica que se deve fazer 'NEXT I'.

Esta instrução BASIC é traduzida para código máquina, o qual se refere à informação guardada na pilha para encontrar o valor corrente de I. A referência é tal que o valor do contador do ciclo possa ser incrementado do valor do passo (também guardado na pilha). Se o valor de I não for superior ao limite do ciclo, o comando do programa passa para a primeira linha a seguir ao início do ciclo (ou seja, o controle passa para a linha 20). A linha seguinte do programa não é, pois, a linha seguinte em termos físicos.

<sup>1</sup> Passo = *step* — incremento entre dois valores consecutivos do ciclo FOR. Em termos *standard*, por convenção é +1. (N. do T.)

```

10 FOR I=1 TO 1000  *
20 PRINT "H"
30 NEXT I
40 STOP

```

Embora esta seja, pelo menos, a segunda vez que se está a executar a linha 20, a tradução deve mesmo ser efectuada antes de se obedecer à instrução. A tradução anterior dessa linha não é lembrada.

Ao encontrar de novo a linha 30, o contador do ciclo é novamente incrementado, e o controle volta à linha 20. Se o novo valor do contador do ciclo, também denominado «variável de iteração», for superior ao limite do ciclo, o comando passa para a linha seguinte a nível físico — a linha 40 — e o programa acaba.

Um interpretador pode ser definido como um tradutor de linguagem que traduz e executa uma instrução de cada vez, normalmente sem guardar o código objecto.

Note-se que, mesmo com um interpretador, parte da informação deve ser retida (por exemplo, nomes de contadores do ciclo, limites do ciclo e passos).

Um equivalente Pascal do programa BASIC que acabamos de ver será

```

PROGRAM CARDINAIS (OUTPUT);
VAR I : INTEGER;
BEGIN
  FOR I := 1 TO 1000 DO WRITELN('H');
END.

```

e, apesar de o programa em BASIC ser muito mais fácil de introduzir, vemos como se reduz a possibilidade de erro.

# O Pascal ao alcance da mão

Devagar e sabiamente,  
tropeçam aqueles que correm depressa.

*Romeu e Julieta, de William Shakespeare*

Nicklaus Wirth ideou o Pascal para ensinar estudantes, com o fim de inculcar conceitos de «boa» prática de programação (entendendo-se por «boa», normalmente, «estruturada»). Dado que a linguagem se destinava ao ensino de estudantes e ia ser introduzida num computador *mainframe* usando processamento em *batch*<sup>1</sup>, uma máquina da série CDC 6000, tinha-se também em vista outro objectivo, o de simplificar a escrita e o teste de programas.

O desejo de ajudar no processo de aprendizagem tem muitos aspectos benéficos e levou o Pascal a ser uma linguagem de fácil uso. Outro ponto positivo a favor do Pascal foi uma agradável atenção à importância de ter os programas certos à primeira, ou com o mínimo possível de complicações. O desenho da linguagem de programação Pascal tenta seguir a frase citada de Shakespeare: os programas funcionam melhor quando a sua escrita e preparação são bem planeados, e introduzir programas à pressa no computador (como por vezes sucede com o BASIC) normalmente implica que o programa «tropece» e não funcione devidamente.

Começaremos por estudar um programa errado de Pascal.

<sup>1</sup> *Batch*: processamento em «lote». Os vários processos aguardam numa fila de espera para serem executados, contrariamente ao processamento em *time sharing* (tempo partilhado) em que o computador distribui a sua atenção pelos vários processos simultaneamente. (N. do T.)

## UM PROGRAMA COM ERROS

No manual do programador que acompanha a versão 4T do Pascal HiSoft para o ZX Spectrum, aparece um exemplo muito bom das possibilidades de existência de erros. Os erros são intencionais, para mostrar como pequenas e imperceptíveis alterações podem causar problemas. Apresentamos aqui o exemplo sem alterações; mas primeiro é necessário explicar o que se pretende do programa.

O programa BUBBLESORT ordena valores inteiros que se encontram num vector de nome «NUMEROS». Comparam-se os dois valores de modo a fazer-se uma ordenação em cada par de elementos sucessivos. Se estiverem na ordem errada, são trocados (neste caso, a ordenação é crescente).

A troca efectua-se armazenando o valor de um dos elementos numa variável temporária, aqui chamada «TEMP», atribuindo o segundo elemento ao primeiro, e finalmente atribuindo o conteúdo de TEMP ao segundo elemento. Considera-se agora o seguinte par de elementos, que contém um elemento do par anterior, depois de se ter (ou não) feito a troca, e este processo continua até se atingir o último par de elementos.

Neste momento é verificada uma variável aqui chamada «noswaps»<sup>1</sup> para ver se houve pelo menos uma troca durante a última vez que a lista de números foi percorrida, e se não houve nenhuma troca, o programa acaba.

Eis o programa:

```
10 PROGRAM BUBBLESORT
20 CONST
30   Size = 2000;
40 VAR
50   Numbers: ARRAY [1..Size]
    OF INTEGER;
60   I, Temp: INTEGER;
70 BEGIN
80   FOR I:=1 TO Size DO
    Numbers[I] := RANDOM;
```

<sup>1</sup> As variáveis usadas para este fim são normalmente designadas pela palavra inglesa *flag* (bandeira). (N. do T.)

```

90 REPEAT
100 FOR I:=1 TO size DO
110 NOSWAPS:=TRUE;
120 IF Number[I] > Number[I+1] THEN
130 BEGIN
140 Temp:=Number[I];
150 Number[I]:=Number[I+1];
160 Number[I+1]:=Temp;
170 NOSWAPS:=FALSE;
180 END
190 UNTIL NOSWAPS;
200 END.

```

Neste programa existem certos erros propositados, e vamos identificar esses erros usando os números das linhas onde eles se encontram. Estes números de linha destinam-se apenas a referenciar a sequência de linhas e não podem ser usados para saltar de uma linha para outra no programa (como se pode fazer no BASIC, por exemplo).

- Linha 10 Falta o carácter «:».
- Linha 30 Não é realmente um erro, mas suponhamos que só há 100 elementos para ordenar?
- Linha 50 O vector é declarado com o nome «Número», mas depois todas as referências são feitas a «Número».
- Linha 65 A variável «noswaps» deve ser declarada como BOOLEAN.
- Linha 100 «Tamanho» devia ser «tamanho - 1».
- Linha 110 Esta instrução devia estar entre as linhas 90 e 100.
- Linha 190 «Noswaps» devia ser «noswaps».

Os tipos dos erros deste programa dividem-se em três títulos, a que chamaremos:

**Erros de sintaxe:** produzidos por construções não admitidas pela linguagem (por exemplo o «:» que falta).

**Erros de semântica:** produzidos por erros na escrita dos nomes de variáveis (por exemplo, o erro ao escrever «noswaps» na linha 190).

**Erros de estrutura:** produzidos pelo desenho incorrecto de um programa (por exemplo o erro na colocação da linha 110).

Em última análise, o pior tipo de erro é o estrutural, na medida em que estes erros não são, normalmente, rejeitados por qualquer tradutor de linguagem. Os erros estruturais podem no entanto ser minimizados, tendo uma linguagem com construções que impossibilitem a existência de certos desses erros.

Por exemplo, o uso abundante de procedimentos e variáveis locais ajudam a conter a contaminação entre variáveis nos casos em que, inadvertidamente, o programador dê o mesmo nome a duas variáveis diferentes.

Os erros de sintaxe produzirão sempre erros, o mesmo acontecendo, normalmente, com os erros de semântica. Ao compilar o programa anterior, surgirão mensagens de erro claramente identificáveis. Para compreender a razão de Pascal e o motivo da existência de testes tão restritivos para erros de sintaxe e semântica, devemos considerar a importância do processamento em *batch* do desenvolvimento do Pascal.

Voltaremos a este programa mais tarde, para mostrar como e porquê ele poderia ser alterado.

### PROCESSAMENTO EM «BATCH»

Para quem só tem experiência de microcomputadores ou sistemas interactivos usando BASIC, o conceito deste tipo de processamento é completamente novo.

A «investida» de sistemas interactivos foi uma grande libertação para aqueles cujos processos de aprendizagem em informática eram controlados por processamento em *batch*.

Este conceito diz respeito ao processamento de «pacotes» (conjuntos) de cartões, como entrada, sendo a saída normalmente produzida sob a forma de «pacotes» para uma impressora de linhas.

Vale a pena falar sobre os diversos estágios da produção de um programa por este processamento:

1. Começo da construção do programa.
2. Enquanto o programa não correr com sucesso no computador, executar as seguintes acções:
  - a) Modificar o programa onde for necessário;
  - b) Transferir o programa para cartões perfurados (o código fonte);
  - c) Ler o conjunto de cartões contendo o programa;
  - d) Compilar o programa, obtendo o código objecto (por transformação nesse código das instruções do código fonte);
  - e) Se a compilação não indicar erros, executar o programa em código objecto;
  - f) Examinar as linhas da listagem de resultados, para verificar se a execução do programa gerou os resultados pretendidos.
3. Analisar os resultados do programa.

Muitas vezes o estágio 2 será o menos importante para o exercício de um estudante (ou seja, ver quais são os resultados), sendo o estágio 1 o mais importante, com o 2a) quase tão importante.

Quando se ensina, a parte mais importante da programação é a escrita de programas que funcionem. Em muitos casos, os resultados de programas que funcionam são muito menos importantes. É normal que os resultados sejam conhecidos, sendo o problema o desenvolvimento de um programa que obtenha os mesmos resultados, que servirão depois para controlar a exactidão do programa.

Os estágios 2c) e 2e) eram normalmente realizados neste tipo de processamento por uma instrução chamada RUN, EXECUTE ou similar. Wirth preocupava-se com o facto de que, com certas linguagens como FORTRAN ou BASIC, era muito fácil que um programa incorrectamente construído fosse compilado com sucesso (estágio 2d). Isto significava que o programa, quando fosse executado (estágio 2e), seria capaz de pregar todo o tipo de partidas que desperdiçam o valioso (e caro) tempo do computador.

O Pascal foi imaginado com o propósito de facilitar a escrita de programas correctos (estágios 1 e 2a), de modo a que poucos programas atinjam o estágio de execução, produzindo resultados errados. A linguagem de programação Pascal, pelo seu estilo e natureza, foi, pois, imaginada para evitar que apareçam erros desnecessários ao escrevermos o programa.

Se estudarmos a sequência de estágios (passos) ao preparar um programa em Pascal no *Apple II* (usando Pascal UCSD), encontraremos as correspondências que se vêem na figura 2.1.

Estágio <i>batch</i>	Comando do Pascal UCSD
2a)	E(DIT
2b)	F(ILE
2c), 2d)	C(OMP
2e)	X(EXECUTE
2c), 2d), 2e)	R(UN

Figura 2.1. Estágios de um processamento *batch*.

Esta separação em comandos e a relação com os estágios *batch* é típica da maior parte dos sistemas interactivos em computadores. Os comandos poderão variar de nome e possivelmente também de estilo, mas o resultado pretendido é o mesmo.

O impacto do processamento em *batch* (enquanto situação) dá maior ênfase à necessidade de estilos eficientes de programação. Com microcomputadores, o BASIC mostra-se bastante sedutor, no que toca — por exemplo — à facilidade de emendar erros (depois de os encontrar). Um programador de BASIC, a menos que seja excepcional, gastará grande parte do seu tempo a corrigir erros pequenos (como a confusão entre COLOR e COLOUR, mencionada no capítulo 1).

Por muito simples que seja a correcção de erros, antes de mais é preciso encontrá-los. Como confessará qualquer programador de BASIC, procurar esses erros é uma tarefa bastante maçadora para um programa BASIC de um tamanho razoável.



## OS FINS DO ESQUEMA DO PASCAL

O ensino de estudantes não deve ser a única intenção da concepção do esquema de uma linguagem. No Pascal encontra-se essa «outra intenção»: em certa medida foi a reação a uma linguagem chamada ALGOL 68, que Wirth sentiu possuir, na sua maior parte, características que não devem associar-se a uma linguagem de programação.

Se voltarmos aos cinco problemas de tradução computadorizada de dados no capítulo 1:

1. Tamanho e complexidade
2. Ambiguidade
3. Conhecimento de sentido vulgar
4. Semântica
5. Tempo e memória

parece que o ALGOL 68 agravou muitos destes problemas. O ALGOL 68 era uma linguagem muito poderosa, mas havia muitos símbolos capazes de gerar ambigüidades, aos quais se devia aplicar um «sentido vulgar» de computador.

O simples símbolo «C» tanto significava que se seguia uma expressão aritmética, como representava uma abreviatura das palavras chave IF, CASE, BEGIN, etc. Paralelamente, o parêntese direito «)» tanto podia terminar uma expressão aritmética, como representar FI, ESAC, END, etc. O significado de «/» — ou seja, a sua semântica — tem de ser diferenciado aplicando um «sentido vulgar ALGOL 68» ao texto do programa, em conjugação com uma apreciação da sintaxe correcta.

Tal como com uma linguagem vulgar, esta complexidade reduziu elevada potência, à custa de ter de se dispor de sistemas tradutores (compiladores) grandes e complicados. Muitas vezes a tradução do código fonte (o texto do programa) para código objecto requeria um grande intervalo de tempo e um grande computador.

Para estabelecer o significado de «/» exigia-se que o tradutor percorresse várias vezes o programa, para clarificar possíveis ambigüidades.

De cada vez que o compilador/tradutor percorre o texto do programa tem-se um «passo», tendo existido diversas e interessantes discussões sobre o menor número possível de passos necessários à tradução de um programa escrito numa versão completa de ALGOL 68. Especialmente para processar um elevado número de trabalhos de estudantes o ALGOL 68 mostrava-se demasiado garruloso em relação aos recursos do computador. Este é o motivo por que se verificaram tantas versões de ALGOL 68 com «cortes».

O ALGOL 68 tinha bastantes ideias inovadoras (por exemplo, a modularidade da linguagem, a escrita de expressões dentro de parênteses). Até certo ponto, o ALGOL 68 «candidatou-se» a ser a primeira linguagem para todos os fins mas, devido à generalidade mencionada, era muito difícil a implementação de um tradutor de ALGOL 68 num computador pequeno.

Em termos retrospectivos, os propósitos do esquema do Pascal aparecem-nos agora assim:

1. Pretendia-se uma linguagem ampla e simples (fácil de aprender e de eficiente tradução);
2. Pretendia-se uma linguagem para fins gerais (mas não para todos os fins);
3. Pretendia-se um veículo para a tradução de *software* portátil (ou seja, *software* que pudesse ser implementado com o mínimo de dificuldade no máximo de computadores);
4. Pretendia-se uma ferramenta para programação sistemática (e que fosse adequada para ensinar e para escrita de *software* de confiança);
5. Devia possuir o máximo de eficiência no tocante a recursos do computador (é por este motivo que existem tantas versões para microcomputadores, cada uma bastante completa para os fins em vista).

Wirth descobriu que a chave para atingir todos estes fins era a ideia de que o Pascal devia possuir um compilador de um só passo. Ao contrário do que acontecia com o ALGOL 68 ou muitos dos compiladores do FORTRAN, o tradutor devia per-

correr o texto uma vez, e uma vez apenas. Para ser capaz de executar esta tarefa de tradução num só passo, o tradutor devia estar numa posição tal que (em cada estágio do processo de tradução) não existissem ambiguidades ou factos desconhecidos pelo compilador.

A linguagem de programação Pascal foi, assim, concebida para fazer face a estes cinco problemas de tradução computacionalizada, negando-os a todos.

O Pascal foi concebido de modo a:

1. Possuir um tradutor tão pequeno e compacto quanto possível;
2. Ter uma sintaxe claramente definida, para evitar problemas de ambiguidade;
3. Não ter «construções frásicas», sem significado claro quando encontradas pela primeira vez;
4. Assegurar que cada parcela da linguagem introduzida no programa (digamos, um nome de variável) estivesse claramente definida antes que fosse empregada pelo programa;
5. Produzir um meio de tradução rápido e um programa em código objecto compacto em termos de utilização de memória e lento na execução.

Ao satisfazer estes critérios, satisfaziam-se igualmente os fins da concepção do Pascal e o impacto desses critérios pode ser ilustrado pelo programa «BubbleSort», já referido.

#### ASSENTAR IDEIAS SOBRE O «BUBBLESORT»

Imaginemos estar na situação do computador que procura traduzir o programa «BubbleSort». Apenas dispomos de um manual com regras para ajudar na tradução e de um livro de apontamentos para escrever novas palavras ou sequências à medida que vão aparecendo. Se encontramos qualquer coisa que não consta do livro de regras e que não escrevemos no livro de apontamentos, temos de parar.

Começaremos pela primeira linha:

#### 10 PROGRAM BUBBLESORT

Descobriremos no livro de regras que os programas em Pascal começam com a palavra PROGRAM, seguida por um identificador (sequência de letras e/ou dígitos, começada por uma letra). No Pascal Hisofo estará a seguir um «;», mas aqui ele não existe porque a linha seguinte é

#### 20 CONST

Assinalamos, pois, um erro da forma «<>» esperado na linha 10 e não fazemos a tradução para código objecto.

Pedir-nos-ão, pois, que, em vez de continuar com a tradução, voltemos a ler o livro de regras e o livro de apontamentos, para verificar se o resto do programa está certo. Escreveremos «BubbleSort» no livro de apontamentos e, junto a ele, o «identificador do programa».

A linha 20 começa com a palavra CONST, pelo que (de acordo com o livro de regras) esperaremos um identificador a seguir, um sinal de igual e um valor constante. Não há nada mais nessa linha, pelo que se passa à seguinte, que é

#### 30 TAMANHO = 2000;

No livro de apontamentos escreveremos «tamanho» e, junto dele, «identificador de constante». O sinal de igual está no local adequado, tal como o valor da constante (ou seja, 2000) e a sequência termina com «;». No livro de apontamentos anotaremos o valor da constante, 2000. Esta parte do programa estava correcta, mas estas linhas não são traduzidas para código objecto, dado que se detectou um erro antes.

A linha seguinte é

#### 40 VAR

pelo que, segundo o livro de regras, esperamos identificadores, seguidos por «» e uma descrição do tipo das variáveis associadas a esses identificadores. A linha 50 é

```
50 NUMBROS : ARRAY [1..TAMANHO] OF INTEGER;
```

começando com o identificador «numeros» (que deve ser escrito no livro de apontamentos) e, depois de «:», uma descrição do tipo da variável a que «numeros» se refere. Definimos «numeros» como uma matriz de inteiros, sendo a sua gama de índices um conjunto de valores entre um e o conteúdo de «tamanho». No livro de apontamentos encontramos «tamanho» como uma constante igual a 2000, pelo que, junto a «numeros», escrevemos «matriz de inteiros cujos limites são 1 e 2000».

Após «:», o livro de regras diz que devemos esperar por uma palavra chave (como BEGIN) ou então por um novo identificador. Se for este o caso, deverá existir a seguir uma nova definição de tipo:

```
60 I, TEMP : INTEGER;
```

Como a linha não começa com uma palavra chave, esperamos por mais identificadores definidos: no livro de apontamentos escreveremos «I» e «temp», juntando-lhes «inteiros». A linha termina com «:» e a linha seguinte começa com uma palavra chave

```
70 BEGIN
```

pelo que terminaram as declarações de variáveis, e começou o bloco do programa principal.

Agora que se completaram as declarações, não consideraremos o programa inteiro, mas apenas as linhas 50, 110 e 190:

```
80 FOR I:=1 TO Size DO  
110 NUMB(I):=RANDOM;  
190 UNTIL Noswaps
```

No livro de regras encontramos, em relação à linha 80, aquilo que devemos esperar quando usamos «FOR». No livro de apontamentos verificaremos que «1» é um inteiro, e que «tamanho» é igual a 2000. No entanto, no mesmo livro não encontramos o identificador «Número» (é um engano; devia ser «números»), e assinala-se outro erro.

Encontramos «noswaps», mas não existem vestígios de um identificador com esse nome, dado que devia existir outra linha

```
85 Noswaps:BOOLEAN;
```

que indica que «noswaps» é uma variável booleana — ou seja, apenas admite os valores TRUE ou FALSE. Contudo, esta linha não ajudaria a linha 190, dado que «noswaps» é um erro na escrita.

Este tipo de exame rigoroso ajuda a evitar erros graves devidos a enganos na introdução do programa. Em FORTRAN e BASIC, por exemplo, «noswaps» seria apenas encarado como um nome diferente de variável, não se tornando isto como erro<sup>1</sup>. O Pascal, forçando o programador a declarar a natureza de todos os identificadores, protege o deslizado de uma catástrofe.

## ANÁLISE DE UM PROGRAMA EM PASCAL

O conteúdo do resto deste capítulo é um pouco mais complexo e pode ser omitido na primeira leitura. Para compreender a fundo como funciona o Pascal, contudo, deverá vir a ser dominado. Este estudo é também relevante para uma compreensão geral de linguagens de computador e dos meios através dos quais os programas em código fonte são traduzidos para código objecto.

Se pensarmos de novo nos diagramas sintácticos da forma de

<sup>1</sup> No entanto, em FORTRAN, a variáveis BOOLEAN do Pascal têm um substituto, o LOGICAL, e deve existir uma declaração prévia à utilização da variável. Falhar essa declaração pode originar erros. (N. de T.)

um programa Pascal, veremos que usam uma superfície bidimensional (a página deste livro) para ilustrar a situação. Um computador não possui uma visão bidimensional, e na realidade é apenas capaz de um funcionamento de tipo sequencial (e este é o motivo por que se espera que os computadores da 5.ª geração sejam capazes de funcionar de modo não sequencial).

Os diagramas sintácticos são uma ajuda para o leitor humano, mas de modo algum o são para o programa tradutor do computador, que usa as regras sintácticas do Pascal para determinar se o programa está correcto. Éra, pois, necessário um método sequencial («linear») para mostrar a sintaxe.

Uma forma de apresentação linear da sintaxe é a notação BNF (*Backus Naur Form*), desenvolvida para descrever a sintaxe de uma linguagem antiga, o ALGOL 60. O método que usamos aqui é afim do do BNF, mas utiliza mais parênteses, estando mais de acordo com os diagramas sintácticos usados no Pascal (ver apêndice B) e mantém mais facilmente uma abordagem modular (ver apêndice C).

No Formalismo Allan (AF), que foi desenvolvido expressamente para este livro, como método de analisar sintaxes, a definição da estrutura de um programa será:

```
[programa] —>
  <<PROGRAM> [identificador] [parâmetros do
    programa]<>>
  [bloco]<..>
```

havendo correspondência directa entre os elementos dos diagramas sintácticos e a definição em AF (ver o apêndice B para mais pormenores sobre diagramas sintácticos, e o apêndice C para aplicações minuciosas de AF). Se examinarmos a definição agora dada de um programa, os símbolos especiais são usados do seguinte modo:

1. Palavras entre [] em AF são equivalentes a palavras dentro de rectângulos, em diagramas sintácticos.  
Símbolos entre <> em AF são equivalentes a símbolos

dentro de elipses «achatadas» ou círculos nos diagramas sintácticos.

3. O símbolo / em AF corresponde a uma escolha entre caminhos alternativos.
4. O que estiver dentro de () deve ser tratado como uma unidade, para fins de definição.
5. O que estiver entre {} é facultativo, podendo ser repetido se o desejarmos, correspondendo aos ciclos dos diagramas sintácticos.

Por outras palavras, o AF dá uma representação linear dos diagramas sintácticos, não lineares. O que se pretende definir está seguido de uma seta, para lembrar que aquele nome «aponta» para uma definição, e para dar ênfase a uma equivalência ou igualdade.

Para dar mais um exemplo de AF, eis a definição de um identificador:

```
[identificador] —>
  [letra] {[letra]/[dígito]}
```

que indica que um identificador começará por qualquer letra, podendo esta ser seguida, facultativamente, por uma sucessão de letras e dígitos, ou só por estes.

Voltemos agora ao programa «BubbleSort», desta vez apresentando-o ligeiramente modificado, com a adição de comentários para aumentar a clareza da estrutura sintáctica.

```
10 PROGRAM BUBBLESORT
11 C ?
12 C Fim do cabeçalho do prog
   rand, início do bloco ?
13 C Início das declarações ?

14 C ?
20 CONST
30 Size = 2000;
40 VAR
50 Numbers : ARRAY [1..Size
  : 0*] OF INTEGER;
```

```

60 I, Temp : Integer;
61 { }
62 { Fin das declarações, início do programa }
63 { }
64 BEGIN
65   FOR I:=1 TO Size DO NEXT
66   II:=RANDOM;
67   REPEAT
68     FOR I:=1 TO Size DO
69       NOswaps:=TRUE;
70       IF Number[II]>Number[I] THEN
71         BEGIN
72           Temp:=Number[I];
73           Number[II]:=Number[I];
74           Number[I+1]:=Temp;
75           I:=I+1;
76         END;
77       UNTIL NOswaps;
78     NOswaps:=FALSE;
79   END;
80 UNTIL NOswaps;
81 END. { Fin de tudo }

```

Podemos comparar a estrutura deste programa e a da definição de um programa. Primeiro, existe a palavra (símbolo Pascal) PROGRAM, mais um identificador (ou seja, BUBBLESORT), e no Pascal *standard* haveria os parâmetros do programa. O Pascal Hisoft (tal como o Pascal UCSD) não usa parâmetros do programa, pelo que a definição pode ser alterada para

[programa UCSD/Hisoft] —>  
 <PROGRAM> [identificador] <;> [bloco] <.>

que indica que, após o identificador, deve existir <;>. Esse carácter não está presente, pelo que haverá erro de compilação.

De acordo com a definição de programa, após o <;> tem-se o bloco, pelo que devemos definir o que é um bloco:

[bloco] —>  
 {[declaração — de — etiquetas]} {[declaração — de — constantes]}

{[declaração — de — tipos]} {[declarações — de — variáveis]}  
 {[declarações — de — rotinas]}  
 <BEGIN {[instrução]} {<;> [instrução]} <END>

A interpretação desta definição é que um bloco é uma sequência de declarações seguida por um conjunto de instruções entre BEGIN e END.

Em AP, a sequência {[declaração de etiquetas]} significa que podem existir zero ou mais ocorrências de itens conhecidos por «declarações de etiquetas», do mesmo modo que podem ocorrer ou não as declarações de constantes, tipos, variáveis e rotinas. As declarações, contudo, quando existam devem estar presentes pela ordem dada. No nosso caso, o programa «Bubblesort» contém apenas declarações de constantes e de variáveis, nomeadamente

```

20 CONST { Início de declaraç
ao de constantes }
30 Size = 2000;
40 JFM { Início de declaraç
ão de variáveis }
50 Number : ARRAY 11..Size
: OF INTEGER;
60 I,Temp : INTEGER;

```

Existem, claro, definições dentro do Pascal das declarações de constantes e de variáveis, que em AP serão

[declaração — de — constantes] —>  
 <CONST> [identificador] <=> [constante] <;>  
 {[identificador] <=> [constante] <;>}  
 [declaração — de — variáveis] —>  
 <VAR> [indicação — de — variável]  
 {[indicação — de — variável]}  
 [indicação — de — variável] —>  
 [identificador] {<.> [identificador]} <;>  
 [indicador — de — tipo]

Temos agora necessidade de definir «indicador-de-tipo», dado ser o único item não definido. Existem muitos indicadores de tipo:

```
[indicador-de-tipo] —>
  [tipo-ordinal-novo] / [tipo-estruturado-novo] /
  [identificador-de-tipo] / <^> [identificador-
  de-tipo]
```

ou seja, quer um «tipo-ordinal-novo», um «tipo-estruturado-novo», um «identificador-de-tipo» ou um ponteiro (*pointer*) — o símbolo ^ indica um ponteiro — para um identificador de tipo. Os significados destes termos serão claros mais adiante, pelo que apenas daremos um exemplo de um indicador de tipo, a «declaração-de-matriz»:

```
[declaração-de-matriz] —>
  <ARRAY> <[> [tipo-ordinal] {<,> [tipo-ordinal]}
  <]><OF> [indicador-de-tipo]
```

Note-se que na definição de declaração de matriz (que é uma forma de indicação de tipo) existe a referência a «indicador-de-tipo». Na definição de «indicador-de-tipo» existe uma referência implícita a «indicador-de-tipo» pelo que, em última análise, o «indicador-de-tipo» é definido em função dele próprio.

A isto chamamos definição «recursiva», da qual é um exemplo mais claro e directo a nova definição de identificador:

```
[identificador-versão-2] —>
  [letra] {[dígito] / [identificador-versão-2]}
```

que diz que um identificador é uma letra, facultativamente seguida por um dígito ou um identificador, sendo o dígito ou o identificador possivelmente repetidos.

Considere-se, por exemplo, o identificador

A2CD5

que começa por uma letra, ficando 2CD5. O resto começa então com um dígito, 2, deixando CD5, e CD5 é já em si um identificador. Pode então ser analisado em termos do esquema de definição.

O formato do Formalismo Allan, com ponteiros para definições, corresponde de perto ao modo segundo o qual muitos tradutores analisam texto em código fonte, para traduzir programas para código objecto.



## As rotinas em Pascal

Trabalhe, sobretudo, com o propósito de simplificar e melhorar a sua declaração de objectivos. Cartão reduziu a três palavras — e através da sua constante repetição, eliminou finalmente o sofrimento.

*Up the Organisation*, de Robert Townsend

As três palavras que Cartão usou de modo tão eficiente foram *Delenda est Cartago*, que significam: «Cartago deve ser destruída».

Cartão acrescentou estas palavras a tudo o que disse ou escreveu sobre quaisquer assuntos.

O equivalente para o Pascal seria «A confusão deve ser destruída». Neste capítulo estudaremos o modo como o Pascal procura destruir a confusão.

### A IDEIA DE UM PROCEDIMENTO

Já vimos como Wirth tentou tornar o Pascal numa linguagem o mais simples possível, sendo um dos modos principais para produzir tal simplicidade um raciocínio claro. O Pascal procura alcançar esse objectivo através do uso de procedimentos.

O apêndice B mostra o diagrama sintáctico de um procedimento. Na sua essência, a sintaxe de um programa e a sintaxe de um procedimento têm muitas semelhanças. Uma maneira de pensar num programa é considerá-lo como uma forma superior de procedimento; ou seja, o procedimento de maior nível hierárquico.

Voltemos ao mais simples dos programas, escrito em Pascal UCSD, para não existirem parâmetros de programa.

```
PROCEDURE MAIS_SIMPLES;
BEGIN
END.
```

O procedimento mais simples é

```
PROGRAM MAIS_SIMPLES;
BEGIN
END.
```

e o procedimento MAIS\_SIMPLES não faz nada, tal como o programa MAIS\_SIMPLES. Tomemos um programa mais complexo.

```
PROGRAM SIMPLES,
(
)
PROCEDURE MAIS_SIMPLES;
BEGIN
END;
(
)
BEGIN
  MAIS_SIMPLES
END.
```

que também não faz nada. Consideremos agora a estrutura deste último programa SIMPLES.

O programa tem o nome SIMPLES, e de seguida define-se um procedimento de nome MAIS\_SIMPLES, cujo conteúdo é

```
BEGIN
END;
```

que não faz nada. O «;» indica o fim do procedimento (dado que o terminador «.» dá o fim do programa). O corpo do programa principal começa com BEGIN. O tradutor sabe que este é o começo do programa, caso contrário haveria outra definição de PROCEDURE ou FUNCTION.

O corpo do programa principal é, então

```
BEGIN
  MAIS_SIMPLES
END.
```

o que significa que, ao encontrar-se o identificador MAIS\_SIMPLES, se deve pesquisar no «livro de apontamentos». O nome MAIS\_SIMPLES é encontrado, e junto a ele a informação do endereço da memória do computador onde o procedimento

começa, dado que o nome MAIS\_SIMPLES é um ponteiro para esse endereço.

Na altura de executar o programa, activa-se o procedimento MAIS\_SIMPLES; copiam-se pormenores do MAIS\_SIMPLES para uma porção da memória (não a rotina completa, só aspectos relevantes), e então o MAIS\_SIMPLES é executado. Executar MAIS\_SIMPLES é não fazer nada.

O nosso programa SIMPLES pode ser refinado para um programa que já não é tão simples.

```
PROGRAM MENOS_SIMPLES;  
{ 1 }  
PROCEDURE MAIS_SIMPLES;  
  BEGIN  
  END;  
{ 2 }  
PROCEDURE SIMPLES;  
  BEGIN  
    MAIS_SIMPLES  
  END;  
{ 3 }  
BEGIN  
  SIMPLES  
END.
```

Este continua a ser um programa que não faz nada, mas que já demora mais para não fazer nada. A razão por que ele demora mais é que há mais coisas que não fazem nada.

Consideremos o que acontece quando o tradutor Pascal percorre o programa. O primeiro procedimento chama-se MAIS\_SIMPLES e junto desse nome já consta a informação de que, a partir de determinado endereço da memória do computador, estão armazenados diversos pormenores relativos a ele. Não existem mais identificadores relevantes, pelo que não há mais complicações.

Examina-se o procedimento seguinte, SIMPLES, e guardam-se os pormenores relevantes: onde começa o código e quais os parâmetros identificadores.

Faz parte do corpo do procedimento SIMPLES o identificador MAIS\_SIMPLES. Este identificador não é novo, na medida em que já foi anunciado ao sistema na declaração de um procedi-

mento anterior, e por isso o sistema sabe qual a acção a tomar. A acção é guardar informações sobre o início do procedimento MAIS\_SIMPLES.

Acabando as declarações de rotinas, começa o programa principal. O programa principal (nível 3) faz uma chamada ao procedimento SIMPLES, já definido (nível 2), que por sua vez contém uma chamada ao procedimento MAIS\_SIMPLES também já definido (nível 1). É como se existisse a sequência.

```
MAIS_SIMPLES  
SIMPLES -> MAIS_SIMPLES  
MENOS_SIMPLES -> SIMPLES -> MAIS_SIMPLES
```

Antes de ter acesso ao procedimento MAIS\_SIMPLES, o identificador relevante deve ser declarado ao sistema, o mesmo acontecendo com SIMPLES.

### A IMPORTÂNCIA DA ORDEM

O programa seguinte não é legítimo, sendo destinado a mostrar a importância da ordem (é interessante notar que a ordem das declarações é também crucial para a linguagem de programação FORTH).

```
PROGRAM ERRAO_ORDEM;  
{ 1 }  
PROCEDURE SIMPLES;  
  BEGIN  
    MAIS_SIMPLES  
  END;  
{ 2 }  
PROCEDURE MAIS_SIMPLES;  
  BEGIN  
  END;  
{ 3 }  
BEGIN  
  SIMPLES  
END.
```

O tradutor encontra o procedimento SIMPLES e armazena essa informação. Dentro do procedimento, o tradutor encontra o

identificador MAIS-SIMPLES, e não consegue encontrar informação sobre ele, nem no livro de regras<sup>1</sup>, nem no livro de apontamentos (ou seja, no analisador de sintaxe e nas listas de identificadores): é produzido um erro.

Usando *look ahead* (processo onde, face a uma dívida num local do texto, se procede a uma pesquisa daí para a frente, memorizando o sítio de onde se partiu), sabemos que seriam dados os pormenores, se o tradutor pudesse esperar. Mas o tradutor não pode e não vai esperar. O tradutor está concebido para percorrer rapidamente o texto do programa, pelo que o programador se deve acomodar à ordem necessária.

É impossível obter traduções que sejam, simultaneamente, rápidas e precisas, e ainda ter uma sintaxe elaborada, a não ser que o computador com que trabalhamos tenha uma grande capacidade e seja muito rápido. Os microcomputadores são relativamente pequenos e lentos; se for necessária uma computação rápida (o que nem sempre sucede), o Pascal pode ajudar.

A ordem pela qual os identificadores se encontram declarados é importante, dada a diminuição de velocidade no caso de existir uma segunda passagem pelos dados. É claro, é sempre possível «fazer um pouco de batota»:

```
PROGRAM CERTINHO;
(*
PROCEDURE SIMPLES; FORWARD
*)
(*
PROCEDURE MAIS_SIMPLES; FORWARD
*)
PROCEDURE SIMPLES;
BEGIN
  MAIS_SIMPLES
END;
(*
PROCEDURE MAIS_SIMPLES;
BEGIN
  END;
*)
```

<sup>1</sup> Aqui, o tradutor iria procurar algo já definido (por exemplo um procedimento «utilitário») pela própria linguagem. (N. de T.)

```
(*
BEGIN
  SIMPLES
END;
*)
```

As declarações FORWARD são o modo de que o programador dispõe para indicar ao tradutor que deve pesquisar mais à frente. O tradutor chega ao procedimento SIMPLES e descobre que, nesse ponto, não há definição do conteúdo do procedimento. O identificador é, no entanto, memorizado junto com informação relevante acerca de eventuais parâmetros (que neste caso não existem), mas não é ajustado qualquer ponteiro para código objecto (que ainda não foi gerado).

A esta rotina segue-se outra onde a situação é idêntica, ou seja, MAIS-SIMPLES. Existem pois, neste ponto, dois identificadores «à procura» de definições.

A declaração seguinte a um procedimento é a declaração de SIMPLES, que já foi identificado junto do tradutor. SIMPLES será então traduzido para código objecto, e ficará com um ponteiro associado ao início do seu código (endereço).

Como parte da definição de SIMPLES, existe uma referência ao identificador MAIS-SIMPLES. Ora, este identificador já é conhecido do tradutor (compara-se com o programa) devido à declaração FORWARD no início. Fará então parte do código de SIMPLES uma chamada ao ponteiro para MAIS-SIMPLES, que por enquanto ainda não referencia o respectivo código objecto.

Quando a definição de MAIS-SIMPLES for traduzida, ajusta-se o ponteiro respectivo para o endereço do início do código deste procedimento, para que o código objecto de SIMPLES saiba a partir de onde deve executar o código de MAIS-SIMPLES (através de um ponteiro para outro ponteiro para código objecto).

O que acabamos de explicar é usualmente considerado um tanto esotérico, e muitas vezes o ignoram ou deixam de lado até ser demasiado tarde no ensino e exame de Pascal. A necessidade de FORWARD é uma das características mais importantes do Pascal, dado que, com esta declaração, se pode ver claramente como funciona o tradutor: (A) o tradutor funciona numa ordem

estruturas e (B) o tradutor funciona usando ponteiros tanto quanto seja possível.

Este assunto ficará ainda mais esclarecido quando examinarmos procedimentos com parâmetros.

### PROGRAMAS E PARÂMETROS TIPO «FICHEIRO»

A distinção entre cabeçalhos em Pascal *standard* e Pascal UCSD é a distinção entre cabeçalhos com e sem parâmetros de programa.

Temos, nomeadamente, que em Pascal *standard* devemos escrever (por exemplo)

```
PROGRAM PASCAL_STANDARD (IN  
  PUT, OUTPUT);
```

enquanto que em algumas versões seria

```
PROGRAM MICRO_PASCAL;
```

As diferenças estão no uso de parâmetros no Pascal *standard* (neste caso, os canais de dados INPUT e OUTPUT). O modo como as versões diferem entre si pode ver-se melhor pela operação do tradutor.

No Pascal *standard* o tradutor armazena identificadores para STANDARD\_PASCAL, INPUT e OUTPUT, enquanto na versão normal para microcomputador apenas é necessário anotar o identificador MICRO\_PASCAL. Na versão «micro-pascal» o tradutor já dispõe de pormenores sobre os canais *standard* INPUT e OUTPUT.

É sempre possível em Pascal *standard* ter cabeçalhos de programas que não incluem INPUT ou OUTPUT mas, uma vez que este facto varia de implementação para implementação, é mais seguro incluí-los.

Quando se usam comandos de entrada (*input*) e saída (*output*) num programa em Pascal *standard*, ou seja, comandos como

```
WRITE (X);  
WRITELN (X);  
READ (X);  
READLN (X);
```

eles são idênticos a

```
WRITE (OUTPUT, X);  
WRITELN (OUTPUT, X);  
READ (INPUT, X);  
READLN (INPUT, X);
```

onde INPUT e OUTPUT são os ficheiros para entrada e saída de dados *standard*. O comando WRITE(OUTPUT,X) escreve o valor corrente do identificador X no ficheiro conhecido como OUTPUT, enquanto no comando WRITE(X) o ficheiro OUTPUT é assumido pelo tradutor, dado que o primeiro identificador da lista (entenda-se por lista o conjunto de nomes de variáveis que se encontram entre parênteses) diz respeito a uma variável e não a um ficheiro.

WRITE diferencia-se do WRITELN porque, com o primeiro, após a escrita dos valores que constam da lista de variáveis, não há passagem para o início da próxima linha lógica, ao contrário do que ocorre quando empregamos WRITELN. A diferença entre READ e READLN é similar<sup>1</sup>.

Se o cabeçalho do programa for, por exemplo,

```
PROGRAM MAIS_FICHEIROS (INPUT,  
  OUTPUT, FICH-1, FICH-2);
```

serão usados (ou pelo menos conhecidos) quatro ficheiros, pelo programa MAIS\_FICHEIROS. Eles são conhecidos como fi-

<sup>1</sup> O uso de READLN obriga a que a lista de variáveis a ter termine com a tecla ENTER (RETURN), sem uma leitura do teclado; e o uso de READ termina a leitura com a última variável lida para dentro da lista. Para leituras de outros ficheiros, e não o teclado, READLN terminará com o carácter «CHARACTER RETURN», que pode ser o carácter de códigos 10 ou 13 do código usado, normalmente o ASCII (V. de T.).

cheiros «externos», sendo INPUT e OUTPUT os ficheiros externos *standard*. Os ficheiros externos são aqueles que permanecem após a execução do programa terminar, aqueles de que o sistema, de um modo geral, tem conhecimento.

Antes de serem usados dentro do programa, os ficheiros FICH-1 e FICH-2 devem ser declarados, por exemplo como ficheiros de texto (*textfiles*); as implementações variarão eventualmente no modo como isto é feito. Por exemplo:

```
PROGRAM MAIS_FICHEIROS (INP
UT, OUTPUT, FICH_1, FICH_2);
VAR FICH_1: FILE OF CHAR;
    FICH_2: FILE OF REAL;
BEGIN (INPUT, X);
    READLN (INPUT, X);
```

Um ficheiro de texto é aquele que se comporta como se estivesse «disposto» linha a linha, como num *écran*. Os ficheiros podem não ser necessários como ficheiros de texto (cujo em que READLN e WRITELN não fazem sentido, dado que assumem «linhas»), podendo ser definidos, por exemplo, do seguinte modo:

```
PROGRAM MAIS_FICHEIROS (INP
UT, OUTPUT, FICH_1, FICH_2);
VAR FICH_1: FILE OF CHAR;
    FICH_2: FILE OF REAL;
    FICH_1: FILE OF CHAR; (FICHEIRO DE CARACTERES)
    FICH_2: FILE OF REAL; (FICHEIRO DE NUMEROS REAIS)
    FICH_1: FILE OF INTEGER; (FICHEIRO DE INTEIROS)
```

O FICHEIRO-INTERNO é um ficheiro «local» ou «interno», e apenas existe para o que o programa se proponha fazer. Os ficheiros internos empregam-se para fins «internos à execução do programa» (por exemplo, armazenar resultados intermédios) e não têm uso quando o programa estiver concluído.

O «alcance» dos ficheiros difere entre eles. O alcance de um ficheiro externo é global ao computador, como um todo, enquanto o alcance de um ficheiro interno é local ao programa (não

tem efeitos fora deste, sendo um ficheiro temporário)<sup>1</sup>. Esta é uma característica importante do Pascal e de muitas outras linguagens. Num diagrama sintáctico, onde existir um «bloco», quaisquer variáveis declaradas dentro desse «bloco» (o um ficheiro é um tipo de variável) são locais a esse bloco — o seu efeito é-lhe restrito.

Se, dentro do bloco, existirem outros blocos (com as suas próprias declarações), o alcance das variáveis do bloco circundante é «global» aos blocos interiores. As duas formas mais comuns de blocos são o programa, como um todo, e os procedimentos e funções.

## CICLOS LOCAIS

Suponhamos que algures no programa se encontram as seguintes linhas:

```
FOR I:=1 TO 10 DO WRITE (I)
; { PRIMEIRO CASO }

FOR J:=10 TO COUNTD 1 DO
{ SEGUNDO CASO }
BEGIN J:=I+10; WRITELN (I
, J) END;
```

Na primeira dessas linhas, ao executar o programa, o tradutor atribuirá valores à variável «I» desde 1 até 10, com incrementos de 1. De cada vez que se activa o ciclo, o valor corrente do contador de ciclo (I) é escrito no dispositivo de saída por defeito, sem mudar de linha. O resultado será então

1 2 3 4 5 6 7 8 9 10

<sup>1</sup> As versões UCSD e Pascal MT/ — podem manipular ficheiros em memória de massa (por exemplo *diskette*), sendo esses ficheiros posteriormente reenviados fora da execução do programa (como a gestão de uma agenda de telefones). (N. do T.)

O «ciclo» em Pascal surge de duas formas, mas em cada uma delas a alteração do contador do ciclo é unitária. O contador pode ser incrementado (como sucede no primeiro caso) ou decrementado (segundo caso) de uma unidade.

Se, por exemplo, pretendemos um contador de ciclo que tenha incrementos de 10 unidades, temos de empregar um artifício, como fizemos no segundo caso, onde a saída será

10	100
9	90
8	80
7	70
6	60
5	50
4	40
3	30
2	20
1	10

correspondendo a coluna da esquerda aos valores da variável «I», entre 10 e 1, e a coluna da direita aos valores da variável J, que é igual a 10 vezes o valor de I. O ciclo FOR em Pascal é assim estritamente definido, o que ajuda o processo de tradução

Se escrevermos um ciclo do seguinte modo:

```
FOR I:=1 TO 10 DO A-PROCEDURE; {LINHA DE
PROGRAMA}
```

o procedimento fará dez vezes a mesma acção. Se, no entanto, o procedimento for definido como

```
PROCEDURE A-PROCEDURE;
VAR I: INTEGER;
BEGIN
  FOR I:=1 TO 10 DO WRIT
  E(LN(I)); {LINHA DO PROCEDIME
  NTO}
END;
```

o que aconteceu? Poderão eventualmente surgir dúvidas quanto às duas linhas comentadas; existe uma linha de programa que contém um ciclo, e uma linha do procedimento quase idêntica. Tomemos o seguinte programa:

```
PROGRAM OVERLAPPING-LOOPS (
INPUT, OUTPUT);
VAR J: INTEGER; { 0 }
PROCEDURE A-PROCEDURE;
VAR I: INTEGER; { 1 }
BEGIN
  FOR I:=1 TO 10 DO WRIT
  E(OUTPUT, I); { 1 }
  WRITELN(OUTPUT, ' ');
END;
BEGIN
  FOR I:=1 TO 3 DO A-PROCE
  DURE { 2 }
END;
```

onde observamos o problema com maior clareza.

Existem as duas linhas de programa (identificadas com {1} e {2}) que usam o mesmo contador de ciclo, I. A variável I está declarada por duas vezes (ou seja, nas linhas {3} e {4}). O problema é «qual é qual?». Resolvamos o problema — que de facto não existe — fazendo o papel de tradutor.

O tradutor lê a primeira linha e verifica que se vão empregar os ficheiros *standard* INPUT e OUTPUT, e que o identificador do programa é OVERLAPPING-LOOPS. Vai tudo para o livro de apontamentos. A linha seguinte identifica um inteiro de nome «I», informação esta que vai para o livro de apontamentos, e junto a ele escreve-se «o alcance é o de OVERLAPPING-LOOPS».

O procedimento é identificado como A-PROCEDURE, e junto do identificador anota-se «dentro do alcance de OVERLAPPING-LOOPS»: ou seja o procedimento está embebido no alcance mais geral do programa. A primeira linha do procedimento identifica um inteiro que será conhecido por I, e junto a este escrever-se-á «alcance de A-PROCEDURE». A linha seguinte (sem contar com BEGIN) usa I num ciclo: voltando atrás no livro de apontamentos, encontramos o identificador I, e junto



a ele a informação de que o seu alcance é o do procedimento A-PROCEDURE

O tradutor verifica se I está declarado em A-PROCEDURE<sup>1</sup> sendo este o I a usar, e não o primeiro I (cujo alcance é o do programa, e cujos efeitos são mascarados por este I).

Quando se encontra uma referência ao I do programa principal, andando para trás no livro de apontamentos, descobrimos primeiro o identificador I de A-PROCEDURE; o I pretendido não está dentro do procedimento, pelo que a busca continua. Encontramos outro I, e junto a ele a informação «dentro do alcance do OVERLAPPING-I OOPS»; é, pois, este I que se vai usar.

Se bem que exista mais de uma referência a um identificador chamado I, e dado o conceito de «alcance» definido para identificadores, estas variáveis diferentes (que têm o mesmo nome) são tratadas de forma diferente (em diferentes partes do programa, ou diferentes alcances).

O «I» do programa principal é diferente do «I» do procedimento.

O resultado «saída» deste programa é

```
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
```

ou seja, existem 8 linhas e em cada uma delas 10 números. Uma

<sup>1</sup> É característica do Pascal que este tipo de variáveis, também chamadas «variáveis de iteração», devam ser locais aos procedimentos e funções que as empregam, não podendo estar declaradas num nível hierárquico superior. (N. do T.)

simples chamada ao procedimento A-PROCEDURE produziria a seguinte linha

```
1 2 3 4 5 6 7 8 9 10
```

dado que o procedimento é constituído pelas seguintes linhas

```
FOR I:=1 TO 10 DO WRITE(CU
  TPUT,I); C:=C+1;
  WRITELN(OUTPUT," ");
```

escrevendo sucessivos valores de I, desde 1 até 10, completando a sequência com um espaço (ou seja, « ») e passando para o início de uma nova linha (com o uso de WRITELN). A informação é escrita no ficheiro de texto por defeito OUTPUT.

Existem 8 linhas de saída efectuadas com A-PROCEDURE, dado que o procedimento é chamado 8 vezes através de

```
FOR I:= 1 TO 8 DO A-PROCEDURE
```

e esta versão do identificador I (que está declarado ao nível do programa principal) não é, claro, tratado como o mesmo identificador da versão I que está declarado ao nível do procedimento.

O alcance do identificador I, declarado na linha marcada com {4} no programa anterior, está restringido a esse procedimento.

## EXPLORAÇÃO DO ALCANCE DE IDENTIFICADORES

Noutra terminologia (a usada antes) qualquer identificador é «local» ao bloco em que está declarado, e portanto não tem efeito sobre identificadores fora desse bloco. Assim como ficheiros locais eram apenas usados dentro do programa (não tinham efeito sobre o sistema, fora do programa), também os identificadores locais só têm efeito dentro do alcance do seu próprio bloco.

A figura 3.1 mostra a distinção entre identificadores «locais» e «globais». Um bloco Pascal é apresentado como um rectângulo dividido em três camadas. Na camada superior estão os parâmetros. A camada do meio consiste nas declarações, onde uma forma de declaração é associar um identificador a outro bloco (tal como um procedimento ou uma função). A camada inferior é formada pelas expressões executáveis do bloco (uma expressão executável é uma expressão que leva a cabo uma acção, como por exemplo «FOR I := 1 TO 8 DO A-PROCEDURE»).

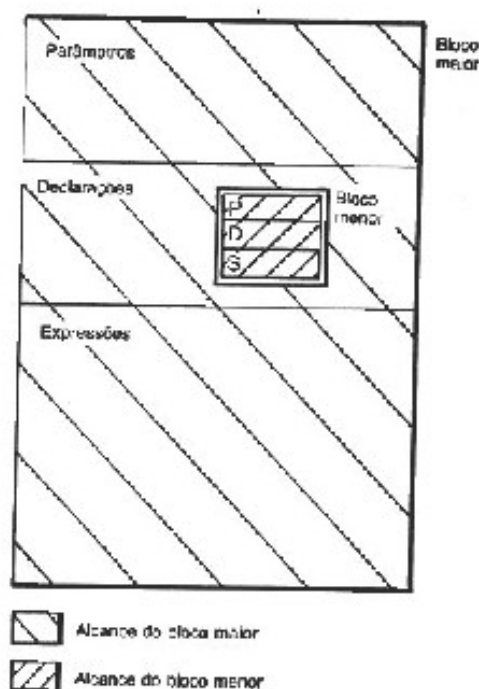


Figura 3.1. Alcance de declarações e parâmetros.

Qualquer identificador dentro do corpo do bloco (ou seja, na camada inferior) deve ser declarada ao bloco, quer seja na camada superior (como um parâmetro), quer seja na camada intermédia (como parte das declarações). O alcance de um parâmetro ou de um identificador declarado é também bloco inferior à «notificação» (termo que abrange a passagem de parâmetros e a declaração de identificadores).

Se, por exemplo, um identificador é declarado num bloco que faz parte de outro bloco, não terá efeitos sobre quaisquer identificadores fora desse bloco.

Identificadores dos blocos exteriores podem, contudo, afectar os identificadores dos blocos interiores (quando em uso), a menos que esses identificadores estejam «mascarados» dentro dos blocos interiores. Por exemplo, o identificador I no programa principal está mascarado pelo identificador com o mesmo nome declarado no procedimento.

Note-se que o bloco interior só pode aparecer na camada intermédia («declarações») do bloco circundante. Na camada inferior, o que aparece é o identificador do bloco (por exemplo, «...A-PROCEDURE»).

O tradutor, quando encontra um identificador de um procedimento, ajusta dois apontadores. O primeiro aponta para o código objecto que contém a tradução das instruções do procedimento; o segundo aponta para uma «pilha de chamadas a procedimentos» que contém informação acerca do procedimento que está actualmente a ser executado, quais os seus parâmetros e identificadores especiais<sup>1</sup>. Quando se completar a execução do procedimento, os pormenores acerca dos seus parâmetros e identificadores são retirados da pilha. O conceito de alcance (apenas para identificadores) pode ser ilustrado com o esquema da figura 3.2:

<sup>1</sup> Os nomes e pormenores das rotinas chamadas são guardados para que o programa saiba para onde retornar, após executar a rotina. (N. do T.)

Declarações no nível		Alcance
Superior	Inferior	
Sim	Não	Total
Sim	Sim	Depende
Não	Sim	Inferior apenas
Não	Não	Nenhum

Figura 3.2. O alcance de identificadores.

Na linha de cima, tem-se que se um identificador estiver declarado no mais externo de dois blocos, o seu alcance estende-se aos dois blocos. Quando o identificador está declarado nos dois blocos, o identificador relevante, ao qual se refere o programa em dada altura, é dado pelo bloco no qual essa referência existe. Se o identificador apenas está declarado no bloco interno, o alcance do identificador resume-se a esse bloco; fazer uma referência fora do bloco constituirá um erro na tradução, que será assinalado.

Evidentemente, se não existir declaração de identificador, quaisquer referências a esse identificador serão ilegais, onde quer se verifiquem.

### PASSAGEM DE PARÂMETROS

Examinemos o programa seguinte, tentando descobrir qual será o resultado final.

```
PROGRAM B_PROGRAM (INPUT, OUTPUT);
VAR I : INTEGER;
{ 1 }
PROCEDURE E_PROCEDURE (J : INTEGER);
VAR K : INTEGER;
BEGIN
  WRITELN (OUTPUT, J);
{ 2 }
```

```
FOR I:=1 TO 10 DO
  WRITE (OUTPUT, I); { 3 }
  WRITELN (OUTPUT, " ");
{ 4 }
END;
{ 5 }
BEGIN { PROGRAMA PRINCIPAL }
  FOR I:=1 TO 8 DO E_PROCEDURE (I); { 4 }
END;
```

O novo procedimento tem um parâmetro, declarado com o nome J, identificando um valor inteiro.

Executar o programa conduzirá ao seguinte resultado:

```
1
1 2 3 4 5 6 7 8 9 10
2
1 2 3 4 5 6 7 8 9 10
3
1 2 3 4 5 6 7 8 9 10
4
1 2 3 4 5 6 7 8 9 10
5
1 2 3 4 5 6 7 8 9 10
6
1 2 3 4 5 6 7 8 9 10
7
1 2 3 4 5 6 7 8 9 10
8
1 2 3 4 5 6 7 8 9 10
```

A questão agora é relacionar a saída de valores com o programa. Existem apenas 4 linhas que desempenham uma ou outra função, enquanto as restantes definem ou notificam. Essas 4 linhas serão as responsáveis pelos resultados (estas linhas estão assinaladas com comentários de {1} até {4}).

As linhas de escrita são as numeradas de {1} a {3}. A linha {1} escreve o valor apontado por J, que identifica um inteiro.

como foi definido na declaração do parâmetro do procedimento. Escreve-se então o valor de J, sendo a próxima acção de escrita efectuada na linha seguinte. Imediatamente a seguir à escrita de J, existe a já familiar linha {2} que escreve os números de 1 a 10, acabando com a passagem para o início da nova linha {3}.

As linhas de 10 números são óbvias, e são iguais umas às outras; é o número que as precede que muda. Esse número, tal como foi estabelecido, é o valor apontado pelo identificador do parâmetro. Este valor parece mudar, com incrementos de 1, entre 1 e 8. Um exame à linha comentada com {4} revela a solução para a mudança de valores de J.

Esta linha é, lembremos,

FOR I := 1 TO 8 DO B-PROCEDURE(J) {4}

que indica uma variação de I entre 1 e 8, sendo o identificador I usado como parâmetro do procedimento B-PROCEDURE. Quando um identificador surge entre parênteses na chamada a um procedimento, o tipo de identificador usado compara-se com o tipo esperado e, se forem iguais, o valor correntemente apontado pelo primeiro identificador passa ao segundo, para a utilização pelo procedimento.

Existe uma diferença entre a descrição formal do parâmetro (ou seja «J: INTEGER») e o valor actual do parâmetro em qualquer chamada ao procedimento (neste caso é o valor actual de I no programa principal)<sup>1</sup>. O que o programa principal passa ao procedimento não é o nome do identificador, mas o valor para o qual ele aponta.

<sup>1</sup> Ao identificador presente no cabeçalho do procedimento, neste caso J, dá-se o nome de «parâmetro mudado», e ao identificador que, na altura da chamada, se pretende passar a J, neste caso I, dá-se o nome de «parâmetro actual». (N. de T.)

## NOME E VALOR

Normalmente, o Pascal usa «passagens por valor» mais do que «passagens por nome»: o valor para o qual o identificador aponta (ao nível do bloco chamador) não é afectado pelo que lhe seja feito dentro do procedimento<sup>1</sup>.

Por exemplo, veja-se o seguinte programa C-PROGRAM:

```
PROGRAM C-PROGRAM (INPUT, OUTPUT);
TYPE
  VAR I : INTEGER;
  { 3 }
PROCEDURE C-PROCEDURE (J :
  INTEGER);
  VAR I : INTEGER;
  BEGIN
    J := J + 1;
    { 1 }
    WRITELN (OUTPUT, J);
    FOR C := 1 TO 10 DO
      WRITE (OUTPUT, I);
    END;
  { 2 }
BEGIN
  INICIO DO PROGRAMA
  PRINCIPAL
  FOR I := 1 TO 8 DO
    BEGIN
      C-PROCEDURE (I);
      WRITELN (OUTPUT, I);
    END
  END;
```

que produz a seguinte saída de resultados:

```
1
1 2 3 4 5 6 7 8 9 10 1
4
1 2 3 4 5 6 7 8 9 10 2
```

<sup>1</sup> Optou-se por «passagens por ...» e não pela tradução literal «chamadas por ...», na medida em que o que se faz é a passagem de um parâmetro e não a sua chamada. O termo «chamada» estaria mais correcto se estivesse em causa o procedimento, o que não é o caso.

```

9
1 2 3 4 5 6 7 8 9 10 3
16
1 2 3 4 5 6 7 8 9 10 4
25
1 2 3 4 5 6 7 8 9 10 5
36
1 2 3 4 5 6 7 8 9 10 6
49
1 2 3 4 5 6 7 8 9 10 7
64
1 2 3 4 5 6 7 8 9 10 8

```

Observa-se que, se bem que o valor apontado por I seja elevado ao quadrado (através da atribuição  $I := J * J$  em C-PROCEDURE, comentada com {1}), o conteúdo de I não se altera, dado que apenas o seu valor passa como parâmetro (vê-se no último número, depois de 10, em cada linha, o valor do identificador I).

Tentemos uma modificação final no programa:

```

PROGRAM C-PROGRAM (INPUT,OU
  TPUT);
  VAR I, K : INTEGER;
      { 1 }
  { 2 }
  PROCEDURE C-PROCEDURE (VAR
    J : INTEGER);
    VAR I : INTEGER;
    BEGIN
      J := J * J;
      WRITELN(OUTPUT, J);
      FOR I := 1 TO 10 DO
        WRITE(OUTPUT, I);
      END;
    { 3 }
  BEGIN { INICIO DO PROGRAMA
    PRINCIPAL }
    FOR K := 1 TO 8 DO
      { 4 }
      BEGIN
        I := K;
        { 5 }

```

```

C-PROCEDURE(I);
      { 6 }
      WRITELN(OUTPUT, I);
      { 7 }
    END
  END.

```

sendo a saída de resultados, neste caso, a seguinte:

```

1
1 2 3 4 5 6 7 8 9 10 1
4
1 2 3 4 5 6 7 8 9 10 4
9
1 2 3 4 5 6 7 8 9 10 9
16
1 2 3 4 5 6 7 8 9 10 16
25
1 2 3 4 5 6 7 8 9 10 25
36
1 2 3 4 5 6 7 8 9 10 36
49
1 2 3 4 5 6 7 8 9 10 49
64
1 2 3 4 5 6 7 8 9 10 64

```

Devemos agora estudar as alterações no programa. A primeira é o aparecimento de uma variável extra no programa principal (K, na linha {1}), sendo esta a variável empregada para controlar o ciclo FOR principal (linha {3}), não passando como parâmetro ao procedimento (linha {5}). Atribui-se ao identificador I o valor de K (linha {4}), sendo então passado o identificador I ao procedimento (linha {5}).

Na declaração do procedimento na linha comentada com {2} existe a alteração -VAR J : INTEGER-, uma instrução para o

tradutor passar o nome da variável assim como o seu valor<sup>1</sup>. O resto do procedimento é igual à versão anterior (ou seja, C-PROCEDURE), sendo então o resultado o mesmo, no que diz respeito ao procedimento. A diferença está no número que termina a sequência de 1 a 10.

Essa diferença vem do facto de que na linha

```
WRITELN (OUTPUT,I) (6)
```

o valor apontado por I foi alterado. A razão para esta alteração é o facto de que, ao chamar o procedimento, foi passado o nome de I (dada a mudança na declaração do parâmetro do procedimento). Tem-se então que, em vez de um ponteiro para o valor de I, se passou um ponteiro para o identificador I do procedimento. Assim, quaisquer alterações em I dentro do procedimento seriam «sentidas» por I fora do procedimento. Como o valor de I foi elevado ao quadrado, I também o foi.

Voltemos aos termos «local» e «global». Se o parâmetro II for definido como

```
NOVA-PROC(II: INTEGER)
```

o alcance do identificador II é o do procedimento NEW-PROC. Se a definição for

```
MAIS-NOVA-PROC(VAR II: INTEGER)
```

o alcance do identificador II é igual ao alcance do identificador actual que substitui II ao ser-lhe passado a II na altura da chamada (tal como I substituiu a J no programa D-PROGRAM).

<sup>1</sup> Na realidade o que passa é o endereço onde está localizada a variável, sendo uma passagem de parâmetros deste tipo designada por «passagem por referência de endereço». Como a alteração passa a ser reconhecida fora do procedimento, um parâmetro declarado sem «VAR» no cabeçalho do procedimento é designado «de entrada», e se tiver «VAR» é designado «de saída». (N. de T.)

## UMA OBSERVAÇÃO FUNCIONAL

As funções têm muito em comum com os procedimentos, com a vantagem de produzirem um valor (associado ao nome da função). Um exemplo comum do uso de funções é o cálculo de um factorial. É muito fácil escrever um programa para calcular o factorial de um número. Por exemplo,

```
PROGRAM FACTORIAL (INPUT, OUTPUT);
{ 2
VAR FAC, NUM, CONTADOR : INTEGER;
{ 3
BEGIN (PROGRAMA PRINCIPAL)

    FAC := 1;
    READLN (INPUT, NUM);
    IF NUM < 1 THEN
        FOR CONTADOR := 1 TO 100
        DO FAC := FAC * CONTADOR;
    WRITELN (OUTPUT, "FACTORIAL
    L := ", FAC);
END;
```

O número do qual se pretende calcular o factorial é NUM, sendo o resultado armazenado em FAC, usando-se o identificador COUNTER para contador do ciclo. Julgamos que o programa se explica a si próprio.

Seria bom poder fazer a atribuição

```
FAC := FACTORIAL(NUM);
```

Assim, e para encontrar o factorial, empregamos uma função que opera sobre o valor de NUM para dar o factorial. A função FACTORIAL pode ser codificada de diversas maneiras. Eis uma delas:

```
FUNCTION FACTORIAL (NUMERO :
: INTEGER) : INTEGER;
{ 2
VAR FAC, CONTADOR : INTEGER;
{ 3
{ 3
BEGIN
```



```

FAC:=1;
IF NUMERO<1 THEN
  FOR CONTADOR:=1 TO NUM
    DO FAC:=FAC*CONTADOR;
  FACTORIAL:=FAC / 1;
END;

```

A primeira diferença que se nota entre uma função e um procedimento é a atribuição de um «tipo» à função. Neste caso, esse tipo é INTEGER. Isto significa que se vai devolver um resultado de tipo inteiro. Podemos alterar a primeira linha, por exemplo, para

```

FUNCTION FACTORIAL (NUMBER:
  INTEGER):REAL1

```

para que o tipo da função passe a ser real. O valor devolvido ao programa é o resultante da atribuição da linha {1}.

A função para calcular o factorial utiliza um método iterativo, porque emprega um ciclo (existem outras formas de ciclo, com o emprego de WHILE-DO e REPEAT-UNTIL, das quais falaremos adiante). O método iterativo é exactamente o mesmo que o usado no programa vulgar, e até certo ponto o uso de uma função tem «fins estéticos».

Dizemos «fins estéticos» porque (para um valor, pelo menos) é desnecessário. Usar uma função torna, possivelmente, o programa mais fácil de entender, mas o seu uso não é integral. O método que a seguir indicamos para o cálculo de factoriais já não pode ser executado facilmente sem o uso de funções.

<sup>1</sup> A possibilidade de o fazer depende da implementação, porque mais adiante se faz FACTORIAL := FAC e, sendo FACTORIAL tipo real e FAC tipo inteiro, dar-se-ia «incompatibilidade de operandos». Isto as nos esquecemos de que não existem factoriais reais, pelo que fazer a função de tipo real tem interesse puramente académico, não fugindo sentido de outro modo.

## Funções recursivas

O factorial de 3 é igual a  $3*2*1$  e o factorial de 4 é  $4*3*2*1$ . Sendo  $N!$  a representação de «factorial de  $N$ », vê-se logo que

$$4! = 4*3!$$

e, que de modo geral, o factorial de qualquer número  $N$  pode ser definido como

$$N! = N*(N-1)!$$

Isto parece-me bastante claro, e em Pascal seria escrito como

```

FACTORIAL(NUMERO):=NUMERO*
  FACTORIAL(NUMERO-1)

```

pelo que a diferença é pequena.

Se o factorial de 1 é 1, qual é o factorial de zero? Por estranho que pareça, é 1 também. Por definição, o factorial de 1 é uma vez o factorial de zero. Portanto, se o factorial de 1 é 1, então o factorial de zero é também 1. Se o número for negativo, o factorial não está definido e será igual a 1<sup>1</sup>.

Assim, se o número for inferior ou igual a 1, o factorial é igual a 1, e se for superior será igual ao produto do próprio número pelo factorial do seu antecedente. A função Pascal fica então

```

FUNCTION FACTORIAL(NUMERO
  : INTEGER): INTEGER;
{ }
VAR FAC : INTEGER;
{ }
BEGIN
  IF NUMERO<1 THEN FAC:=1
  ELSE FAC:=NUMERO*FACTORI
    AL(NUMERO-1); { 1 }
  FACTORIAL:=FAC
END;

```

<sup>1</sup> Isto apenas se aplica à função escrita. Por definição, não existe o factorial de um número negativo, havendo erro de execução no programa se lho pedirmos. (N. do T.)

Foi por este motivo que dissemos que o uso de uma função era parte integrante deste método de cálculo. Na linha comentada com {1}, a função FACTORIAL faz referência a si própria (recursividade).

O tradutor, ao percorrer a definição da função, encontra o identificador FACTORIAL (na linha {1}), e descobre que FACTORIAL é uma função da qual ele conhece alguns pormenores, incluindo as naturezas dos parâmetros formais, que são comparadas com as dos parâmetros actuais, e a informação sobre o endereço do início do código objecto da função FACTORIAL (código esse que, obviamente, ainda não está completo).

Para explicar como funciona este mecanismo, temos, antes, de examinar o modo como se empregam parâmetros em funções (e procedimentos), o que nos leva à distinção entre chamadas por nome e chamadas por valor.

A figura 3.3(a) mostra como um identificador «aponta» para um valor. A figura 3.3(b) usa este simbolismo para mostrar a acção de uma chamada normal por valor — ou seja, apenas o valor é copiado entre identificadores. A figura 3.3(c) mostra uma chamada por nome — ou seja, copia-se o nome do identificador (o que implica que também o valor seja copiado).

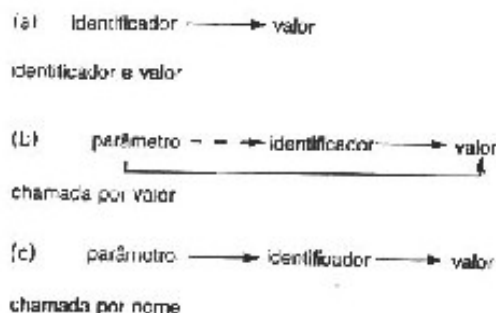


Figura 3.3. Nome e valor.

A figura 3.4 mostra como o tradutor faz face à situação de haver chamadas recursivas (quando o bloco chamador é o bloco

chamado), neste caso de funções. O tradutor comporta-se exactamente como se fosse chamada outra função qualquer, e uma função recursiva não toma mais tempo do computador do que se não o fizesse.

Quando se chama uma função, os pormenores da função, incluindo um ponteiro para o código objecto dessa função, são armazenados na pilha de rotinas.

Se a função faz uma chamada a outra função ou a um procedimento, os novos pormenores são também copiados para a pilha, mantendo os anteriores a sua presença. Os pormenores apenas são copiados após terem sido modificados os da função anterior, para terem em conta os valores correntes e o estado corrente de execução do código objecto da função.

Como sempre em Pascal, os ponteiros são uma chave de compreensão.

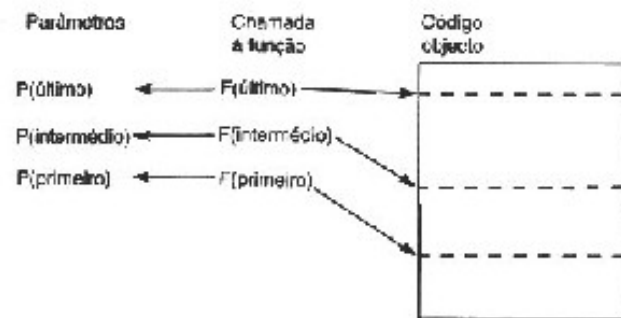


Figura 3.4. Chamadas recursivas à função F.

Existem três funções na pilha de rotinas da figura 3.4, onde todas as funções partilham o mesmo identificador (o que poderiam ser exemplos da função FACTORIAL). O estado de coisas é tal que a execução do programa está no início da execução da terceira chamada à função. A função imaginária da figura não é FACTORIAL, pois aqui as chamadas são feitas em pontos diferentes ao longo do código objecto.

A primeira chamada executou três quartos do código da função, a segunda chamada chegou até meio, e a terceira chamada ainda só começou a ser executada. Note-se que só é necessária uma vez a existência do código objecto da função, dado que cada chamada contém apontadores para o código objecto apropriado.

O alcance do parâmetro é, pois, local à função ou procedimento que o emprega; se for passado a outra rotina dentro da rotina chamadora, é como se tratasse de outro identificador.

## Os tipos em Pascal

O conhecimento adquire duas formas. Sabemos alguma coisa sobre um assunto, ou sabemos onde obter informações sobre ele.

Samuel Johnson em *Life of Johnson*, de Boswell

C. A. R. (Tony) Hoare colaborou bastante com Nicklaus Wirth no desenvolvimento do Pascal, e foram dele muitas das ideias sobre tipos de dados em Pascal.

Várias pessoas, entre as quais o professor David Barron, afirmaram que a descrição do Pascal como «uma linguagem para programação estruturada» dá uma ênfase indevida às estruturas de controle usadas no Pascal. Barron argumenta que são as capacidades de estruturação de dados do Pascal que a tornam uma linguagem tão poderosa. Concordamos com Barron no facto de que as facilidades em estruturar dados em Pascal são mais importantes do que as relativas a estruturas de controle, e por isso deixamos as estruturas de controle para o próximo capítulo.

Do nosso ponto de vista, a noção de tradutor é de suprema importância para a compreensão do Pascal. Por isso, julgamos que o modo como o Pascal utiliza blocos, procedimentos e funções é mais importante que as ideias (conceitos) sobre tipos de dados ou estruturas de controle, ao tentar compreender a linguagem.

A tendência para uma concentração em estruturas de controle, esquecendo a importância crucial dos tipos de dados numa linguagem de programação, é uma característica comum da maior parte dos comentários mais superficiais sobre «programação estruturada». Se bem que os meios sofisticados possíveis para controlar um programa sejam importantes aspectos de Pascal, os mecanismos de controle estruturados não constituem, por si sós, uma linguagem de programação estruturada.

## TIPOS BÁSICOS DE DADOS EM PASCAL

Nas matemáticas e em muitas outras disciplinas, dispor de uma boa notação, uma boa maneira de descrever o que se passa pode ter influência decisiva nos progressos dessa disciplina.

Argumentam os apoiantes do Pascal que este fornece essa influência na notação para programação mas, por outro lado, há utilizadores de outras linguagens (especialmente linguagens como FORTH e LOGO) que fazem declarações similares. Quer o FORTH quer o LOGO podem receber a classificação de linguagens de programação «estruturada», e em ambas as linguagens se verifica a existência de ideias bastante claras sobre distinção entre nomes de identificadores e os valores a eles associados.

Existem vários tipos básicos de dados em Pascal, e já encontramos a maior parte deles. Esses tipos são:

**Integer** (inteiro). Os valores de identificadores deste tipo pertencem a um subconjunto dos números e dependem da implementação. O maior inteiro disponível no sistema tem o nome (*standard*) de MAXINT.

**Boolean** (lógico). Os valores de identificadores deste tipo assumem os valores lógicos TRUE (verdadeiro) e FALSE (falso).

**Real** (real). Os valores de identificadores deste tipo pertencem a um subconjunto, que depende da implementação dos números reais. Por exemplo, temos 1, 100, 0.1, 5E-5 ou 87.35E+8<sup>1</sup>.

**Char** (carácter). Os valores de identificadores deste tipo são alfanuméricos que dependem da implementação<sup>2</sup>. São individualmente representados pelos próprios caracteres entre plicas (plica — o carácter representado por ').

Examinando a função para cálculo de um factorial, podemos obter uma vaga ideia do modo como estes tipos básicos servirão

<sup>1</sup> O símbolo «E» representa «potência de 10», sendo o ponto (.) igual à vírgula (,) para o computador. Assim, o que para o computador é 87.35E+8 para nós será  $87,35 \times 10^8$ . (N. do T.)

<sup>2</sup> Normalmente, o conjunto de caracteres usados é o ASCII. (N. do T.)

de «base de construção» na direcção de estruturas mais complexas.

Um problema que se nota nas funções de cálculo de factoriais do capítulo 3 é a possibilidade da existência de valores não válidos, que não são detectados (pois todos os valores negativos ficam com um factorial igual a 1). Existem maneiras de controlar dentro do programa esses valores (por exemplo, verificar se o valor é negativo); mais fácil, no entanto, é encarregar o tradutor e a execução do programa de efectuarem esta tarefa.

A nossa sugestão é que, durante a execução do programa, valores não válidos (como a utilização de um número negativo neste caso) produzam um erro. Dentro da tradução, portanto, deve existir um item que diga «para este cálculo apenas são admitidos números inteiros positivos». Isto não é conceptualmente diferente, para a tradução, da decisão de que certos números devem ser inteiros, onde portanto a introdução de um número real origina um erro.

Eis uma função que oferece auxílio:

```
PROGRAM USO_DE_MAXINT (INPUT, OUTPUT);
{
}
TYPE POSINTEIRO = 0..MAXINT;
{
}
VAR POSNUMERO : POSINTEIRO;
{
}
FUNCTION FACTORIAL (NUM : POSINTEIRO) : POSINTEIRO;
{
}
VAR FAC : POSINTEIRO;
BEGIN
  IF NUM = 0 THEN FAC := 1
  ELSE FAC := NUM * FACTORIAL (NUM - 1);
  FACTORIAL := FAC;
END;
{
}
BEGIN { PROGRAMA PRINCIPAL
  WRITELN (OUTPUT, 'QUAL É O
  NUMERO?');
```

```

READLN(INPUT, POSNUMERO);
C := 0;
POSNUMERO := FACTORIAL(POSNUMERO);
WRITELN(OUTPUT, 'O RESULTADO E ', POSNUMERO);
END.

```

A diferença principal surge na declaração de tipo na linha comentada com 1. Declara-se um novo tipo de identificador que admite valores inteiros, mas apenas aqueles que se encontrem entre zero e o valor máximo que o computador em questão admite.

A introdução de um número que não esteja dentro daquele subconjunto, na linha {3}, produz automaticamente um erro de execução. A variável POSNUMERO espera um valor inteiro positivo e, se o valor introduzido não for desse tipo, a execução não prossegue.

Note-se que a função FACTORIAL (LINHA {2}) está também definida como de tipo POSINTEIRO, se bem que, com as verificações feitas ao valor do parâmetro, um simples INTEGER já fosse suficientemente seguro.

Voltando ao programa EGYPT, descrito na Introdução, vemos que se dá aos nove valores desconhecidos os identificadores de «A» até «I», e que o seu tipo é INTEGER (podia ser POSINTEIRO). A declaração era:

```

VAR A,B,C,D,E,F,G,H,I : IN
TEGERS;

```

e no entanto mais adiante haverá a verificação

```

IF ABS(10*(A+30000*B+300000*
C+3000000*D+30000000*E+300000000*F+3000000000*G+30000000000*H+300000000000*I)-
1E-5) < 1E-5

```

o que é ridículo, dado que se está a comparar um valor inteiro (o resultado das operações entre parênteses) com um valor real (1E-5). A pessoa que escreveu o programa estava obviamente habituada a BASIC, onde as verificações de validade têm de ser

aproximadas. Em Pascal, contudo, são possíveis verificações exactas empregando-se inteiros.

Se bem que o tradutor não indique o erro informando «está a comparar um inteiro com um real. Cuidado!», a linha não faz sentido.

## TIPOS «ARRAY» (MATRIZ) E «SCALAR» (ESCALAR)<sup>1</sup>

Um dos tipos estruturados mais comumente usados em Pascal é tão popular que existe um construtor especial chamado *array* que poderia, por exemplo, reduzir a declaração

```

VAR A,B,C,D,E,F,G,H,I : IN
TEGERS

```

através da notação mais compacta e agradável

```

VAR ADIVINHAS : ARRAY [1..9]
OF INTEGER;

```

onde existe uma dualidade tal que ADIVINHAS[1] é equivalente ao «A» anterior, ADIVINHAS[2] é equivalente a «B», e assim por diante até ADIVINHAS[9], equivalente a «I».

Os valores armazenados na matriz devem estar entre zero e nove, pelo que podemos estar tentados a declarar

```

TYPE
DECINTEIRO = 0..9;
DECLARRAY = ARRAY [1..9]
OF DECINTEIRO;
VAR
ADIVINHAS : DECLARRAY;

```

que efectivamente declara ADIVINHAS como uma matriz de nove elementos, onde cada elemento é um inteiro decimal entre

<sup>1</sup> Matematicamente, define-se «escalar» como uma quantidade que não necessita de mais do que o seu valor para ser definida, em oposição ao «vetor», que necessita ainda de uma direcção. (N. do T.)

zero e nove. Repare-se na sequência de declarações de tipos.

A sequência seguinte não é válida, e iria confundir o tradutor.

```
TYPE < SEQUENCIA INVALIDA >
?
DECARRAY = ARRAY [1..9]
OF DECINTEIRO,
DECINTEIRO = 0..9;
```

A confusão advém do facto de que DECINTEIRO deve ser declarado ao sistema antes de ser utilizado, neste caso auxiliando a declaração de DECARRAY. Não deve ser colocado na ordem devida antes de poder funcionar correctamente. Por «ordem devida» entende-se «nunca fazer e inesperado».

As matrizes podem ter mais do que uma dimensão, por exemplo:

```
VAR
  JOGO_DE_DAMAS = ARRAY [1..8, 1..8] OF PECAS;
```

onde o tipo PECAS é definido como

```
TYPE
  PECAS = (PRETA, BRANCA, VAZIO);
```

ou seja, o valor de um identificador do tipo PECAS só pode ser PRETA, BRANCA ou VAZIO; no jogo das damas, a casa conterá uma peça branca, uma preta ou estará vazia.

O tipo PECAS é o exemplo de um tipo escalar, e sobre este tipo definem-se as seguintes operações:

SUCC(PRETA) que é BRANCA  
PRED(VAZIO) que é BRANCA  
ORD(PRETA) que é zero  
ORD(VAZIO) que é 2

ou seja, os índices dos itens de um tipo escalar estão numa dada

ordem<sup>1</sup>. Ao primeiro item dá-se o índice zero (neste caso a PRETA) e existem funções para devolver o elemento anterior ou o seguinte de um elemento dado (SUCCessor — dá o seguinte; PREDecessor — dá o anterior)<sup>2</sup>.

O tipo básico BOOLEAN é também um exemplo de um tipo escalar, que pode ser considerado como definido por

```
TYPE
  BOOLEAN = (FALSE, TRUE);
```

### O «set», um tipo não estruturado

O tipo *set* (conjunto)<sup>3</sup> define os limites de valores de um conjunto, onde (ao contrário do tipo escalar, que até certo ponto é parecido com ele) não existe uma ordem particular dos elementos do *set*. Usa-se, muitas vezes, um tipo escalar para delimitar a variação dos elementos de um *set*, por exemplo,

```
TYPE
  MESES = (JAN, FEV, MAR,
  ABR, MAI, JUN, JUL, AGO, S
  ET, OUT, NOV);
  ESTACAO = SET OF MESES;
  { }
  VAR
    PRIMAVERA, VERAO, OUTONO
    , INVERNO : ESTACAO;
  MES : MESES;
```

<sup>1</sup> Se aplicada ao código ASCII, ORD dá-nos, por exemplo, ORD('h')=98, tendo-se que a sua inversa é — geralmente — CHR, onde CHR(98)='h'. (N. do T.)

<sup>2</sup> Tal como ocorre para ORD e CHR, se aplicarmos PRED e SUCC ao código ASCII temos, por exemplo, PRED('C')='B' e SUCC('C')='D'. (N. do T.)

<sup>3</sup> Ao longo destas referências a *set* preferi manter o original, na medida que julgo que este é um dos (muitos) termos que perderia simplicidade e clareza com a tradução. (N. do T.)



Para verificar se um valor específico de MES está dentro dos limites de PRIMAVERA, podia escrever-se a linha

```
IF MES IN PRIMAVERA THEN
  WRITE(OUTPUT, MES, ' E
  PRIMAVERA ');
```

No programa EGYPT existia uma longa sucessão de testes IN semelhantes, onde o ser era escrito em extensão na altura do teste, por exemplo:

```
IF IN (E, C, D, F, F, G, H, I);
```

o que realmente significa «o valor de A é igual a algum dos valores de B até I».

Um teste mais fácil seria a definição de

```
TYPE
  DECINTeiros = (0..9);
  SETNUMERICO = SET OF DECINTeiros;
  C;
  VAR
    SET_DE_REFERENCIA : SETNUMERICO;
    A, B, C, D, E, F, G, H, I : DECINTeiros;
  { QUALQUER OUTRAS DECLARAÇÕES NECESSÁRIAS }
```

e, dentro do programa, a definição de SET\_DE\_REFERENCIA

```
SET_DE_REFERENCIA := (0..9)
```

A variável SET\_DE\_REFERENCIA contém os dígitos de zero a nove, que são os necessários. Assim, na altura das comparações, escrever-se-ia

```
IF SET_DE_REFERENCIA = (A, B,
C, D, E, F, G, H, I) THEN WRITE(
OUTPUT, A, B, C, D, E, F, G, H, I)
```

em vez da série de testes extremamente complexa do programa. Os diversos operadores relacionais conhecidos funcionam com set do seguinte modo:

- = igualdade entre sets
- <> desigualdade (diferença) entre sets
- > contém
- < está contido em

e se alguns dos valores em [A,B,C,D,E,F,G,H,I,3] estiver duplicado, este conjunto não será igual a SET\_DE\_REFERENCIA. Assim, é sempre verdade que

```
SET_DE_REFERENCIA >= (A, B, C,
D, E, F, G, H, I, 3)
```

Outras operações sobre set são

- + união entre sets (conjunção)
- \* interseção de sets
- diferença entre sets (disjunção) ou seja, A-B dá um set formado pelos elementos de A que não o são de B

Por exemplo, o resultado da diferença entre sets

```
SET_DE_REFERENCIA - (A, B, C, D,
E, F, G, H, I, 3)
```

é o conjunto de números não usados na multiplicação e assim uma medida da sobreposição dos elementos individuais de A,B,C,D,E,F,G,H,I,3. Note-se que SET\_DE\_REFERENCIA é o «set universal» para este universo de valores, pelo que é sempre verdade que

```
SET_DE_REFERENCIA >= (A, B, C, D,
E, F, G, H, I, 3) - (A, B, C, D, E, F,
G, H, I, 3)
```

dado que o set de A até 3 é um subconjunto (ou melhor, um «sub-set») de SET\_DE\_REFERENCIA.

### «Records» e estruturas

O tipo *record* (registo) é o mais flexível dos tipos de dados do Pascal. Um «tipo *record*» é a base para uma estrutura que pode ter as mais variadas características internas. A estrutura pode combinar, por exemplo, elementos reais, inteiros, escalares e do tipo *set*.

Eis um exemplo popular (7):

```
TYPE
( 1 )
DATA = RECORD
  DIA: 1..31;
  MES: JAN, FEB, MAR, APR, MAY,
  JUN, JUL, AGO, SET, OCT, NOV, D
  EZ;
  END; INTEGER;
END;

( 2 )
TRABALHO_DE_CASA = RECORD
  ASSUNTO: (MATEMATICA, FISICA,
  GEOGRAFIA, INOLES, PORTUGUES);
  DATA_DADO: DATA;
  DATA_DE_EXECUCAO: DATA;
  CLASSIFICACAO: 0..20;
  SATISFACAO: (NENHUMA, POUCA,
  ACEITAVEL, ESQUECIDO);
  CASTIGO: BOOLEAN;
END;

( 3 )
VAR
( 4 )
TRABALHO_DIARIO: ARRAY [1..5]
  OF TRABALHO_DE_CASA;
```

Para atribuir a informação de que o décimo quinto item do `TRABALHO_DE_CASA` originou punição, escrevemos

```
TRABALHO_DIARIO(15).CASTIGO
:= TRUE;
```

e o motivo foi

```
TRABALHO_DIARIO(15).SATISF
ACAO := ESQUECIDO;
```

donde que

```
TRABALHO_DIARIO(15).CLASSI
FICACAO := 0;
```

Na realidade, esta informação podia ter sido introduzida de modo mais compacto, fazendo

```
WITH TRABALHO_DIARIO(15)
DO BEGIN
  CASTIGO := TRUE;
  SATISFACAO := ESQUECIDO;
  CLASSIFICACAO := 0;
END;
```

As possibilidades de uso das estruturas à base de *records* para simplificar cálculos (processamento, de um modo geral) são, assim, bastante amplas. Existem métodos mais complexos de utilização de *records* (chamados «variantes») que permitem novas formas de manipulação de dados, mas não os estudaremos aqui.

### «POINTERS» E LISTAS

O Pascal, como já vimos, faz uso extensivo de *pointers* (ponteiros) na tradução e execução de programas. Não deverá, por isso, surpreender a descoberta de que o Pascal apresenta a facilidade explícita da utilização de *pointers*. Será, contudo, uma surpresa saber que a ordem do tradutor é interrompida quando se consideram *pointers*.

<sup>7</sup> Ponteiro. Opto por deixar a palavra inglesa na medida em que a semelhança do que ocorre com *set*, *array*, *record*, e outras, é a usualmente usada quando o assunto tratado é Pascal. (N. do T.)

Começemos por ver um exemplo de uma declaração de um tipo «**POINTER**»:

```

TYPE
{
}
POINTER = ^LIST_ITEM;
{
}
LIST_ITEM =
RECORD
  CONTEUDO: INTEGER;
  ITEM_POINTER: POINTER
END;
VAR
{
}
PTR1, PTR2: POINTER;
ININT: INTEGER;

```

O primeiro ponto a notar é que o *pointer* é declarado como um *pointer* para um item chamado **LIST\_ITEM**: IMPORTANTE. **LIST\_ITEM** ainda não foi declarado. As estruturas que usam *pointers* não usam locação «estática», tal como as variáveis vulgares, dado que (por usar *pointers*) a armazenagem desses itens chega a crescer para além do permitido. As variáveis que usam os mecanismos de *pointers* (ou seja, variáveis «dinâmicas») usam um mecanismo chamado «pilha»<sup>1</sup> (*heap*) de armazenagem.

Usando uma forma de «declaração *forward*», o tradutor é avisado de que, quando o identificador **LIST\_ITEM** é declarado, devem colocar-se na *heap* exemplares desse tipo de variável. A seta (^), ou acento circunflexo, não é então apenas um modo de definir que uma variável é de tipo **POINTER**, mas também que é uma mensagem para o sistema, para que este se prepare para a variável dinâmica (no caso presente, **LIST\_ITEM**).

A variável **LIST\_ITEM** é do tipo *record*, com duas subestruturas: **CONTENT**, que é um inteiro, e **ITEM\_POINTER**, que é do tipo **POINTER**, e assim um ponteiro para outro elemento **LIST\_ITEM**. As variáveis, cujo nome genérico é **LIST\_ITEM**, formam aquilo a que se chama uma «lista ligada».

<sup>1</sup> Não confundir com *stack*, que tem também «pilha» por tradução. (N. do T.)

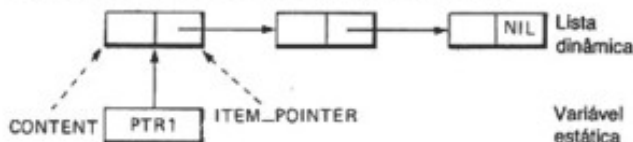
Nas declarações de variáveis não existem declarações de um identificador do tipo **LIST\_ITEM**, dado só existirem declarações de variáveis tipo **POINTER** e **INTEGER**. Só através dos *pointers* teremos acesso à lista ligada (ver figura 4.1).

Para começar a criar uma lista ligada, precisamos de inicializar a lista. Para começar o trabalho devemos ter bases firmes, as duas variáveis estáticas **PTR1** e **PTR2**. Se bem que estejam definidos como *pointers*, a posição corrente do identificador (sua localização em memória) não varia, o que apenas acontece com o seu valor (tal como com todas as variáveis estáticas).

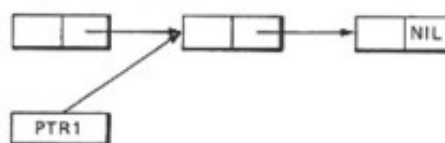
Começamos o processo fazendo<sup>1</sup>

```
PTR1 := NIL;
```

O *pointer* é deste modo inicializado com um valor que não aponta para lado nenhum. Lêem-se os números armazenados na variável estática **ININT** lendo-os um a um. A sua sequência é



(a) Apontador para o fim da lista



(b) Apontador para depois de se fazer **PTR1 := PTR1^ITEM\_POINTER**

Figura 4.1 Uma lista ligada.

<sup>1</sup> A constante **NIL** é muito importante no Pascal, dado que não tem tipo fixo, valendo sempre zero. Pode ser atribuído a um inteiro, e vale 0, ou a um real, e vale 0.0, ou a um carácter e vale o carácter 0 (NUL) do código ASCII, ou ainda a um *record*, valendo um *record* vazio. (N. do T.)

```

BEGIN
  < >
  NEW(PTR2); {RESERVA ESPA
CO PARA A VARIÁVEL DIN
AMICA}
  < >
  PTR2↑.CONTEUDO:=ININT;
  {O CONTEUDO DO ELEMEN
TO DE LIST ITEM PARA
O QUAL PTR2 APONTA APO
RA E FEITO IGUAL AO NO
VO VALOR}
  PTR2↑.ITEM_POINTER:=PTR1
;
  {ITEM_LIST APONTA PAR
A O ÚLTIMO ITEM,
INICIALMENTE NÃO EXIST
ENTE}
  < >
  PTR1:=PTR2; {O LOCAL EM
QUE O NOVO ELEMENTO LI
ST_ITEM É ARMAZENADO
E GUARDADO EM PTR1, PA
RA SER USADO NO ITEM_P
OINTER SEGUINTE}
  < >
END

```

e a sequência continua até se esgotar a fonte de dados.

Para andar para trás na sequência podemos usar *pointers*, fazendo

```

WHILE PTR1<>NIL DO
  BEGIN
    WRITELN(OUTPUT,PTR1↑.C
ONTEUDO); {ÚLTIMO CONTEUD
O}
    PTR1:=PTR1↑.ITEM_POINT
ER {POINTER SEGUINTE}
  END

```

Como se vê, a flexibilidade deste tipo de armazenagem de dados terá grande utilidade em aplicações como processamentos de listas. O uso de «pilhas» (*heap*) de armazenagem para variáveis dinâmicas requer eventualmente um grande espaço de memória do computador, pelo que é melhor usá-las com cuidado em microcomputadores.

## A organização do Pascal

Agora mais do que nunca, a Humanidade está perante uma encruzilhada. Uma via conduz ao desespero e total falta de esperança. A outra, à total extinção.

Rezemos para que haja sabedoria para escolher acertadamente.

*My speech to the Graduates*, de Woody Allen

Falamos agora de decisões, de como podemos fazer os nossos programas adaptar-se a circunstâncias que variem.

A maior parte dos programas analisados até agora usam métodos para controlar o padrão dos acontecimentos, o mais comum dos quais é o ciclo. No programa EGYPT, por exemplo, existem diversos ciclos, cada um dentro do outro. Existem outros modos de controlar o fluxo do programa, alguns deles menos confusos.

### O PROGRAMA «STARS»

Para ilustrar esta discussão, estudaremos o programa «Stars», um jogo favorito desde os tempos dos grandes computadores e BASIC interactivos.

Neste programa, o computador escolhe um número aleatório entre 1 e 100, e o utilizador procura adivinhar esse número.

As pistas são fornecidas por asteriscos (\*). Um asterisco significa que o número introduzido está longe do escolhido pelo computador, e sete asteriscos significam que está bastante perto. O jogador dispõe de sete tentativas.

Qualquer programa deveria fornecer instruções sobre o seu uso, mas tais instruções, neste caso, tornariam mais difícil ver a estrutura do programa. Não existem, assim, instruções mas — à parte essa omissão — a forma do texto Pascal é bastante idêntico ao original em BASIC.

```

PROGRAM GOTO_1 (INPUT, OUTPUT);
(* *)
LABEL
100, 110, 120, 130, 140, 150,
160, 170, 180;
CONST N = 100;
MAXIMUM = 7;
VAR C_NUMBER, ADIVINHA, DI
FERENCA, CONTADOR : INTEGE
R;
(* *)
BEGIN
C_NUMBER := RANDOM(R);
(* DEPENDE DA IMPLEMENTAÇÃO *)
CONTADOR := 1;
(* *)
100 : READ(ADIVINHA);
IF ADIVINHA = C_NUMBER T
HEN GOTO 170;
(* *)
DIFERENCA := ABS(C_NUMBER
R-ADIVINHA);
IF DIFERENCA = 64 THEN
GOTO 160;
IF DIFERENCA = 32 THEN
GOTO 150;
IF DIFERENCA = 16 THEN
GOTO 140;
IF DIFERENCA = 8 THEN
GOTO 130;
IF DIFERENCA = 4 THEN
GOTO 120;
IF DIFERENCA = 2 THEN
GOTO 110;
(* *)
WRITE(' ');
100 : WRITE(' ');
120 : WRITE(' ');
130 : WRITE(' ');
140 : WRITE(' ');
150 : WRITE(' ');
160 : WRITELN('#');
(* *)
CONTADOR := CONTADOR + 1;
IF CONTADOR <= MAXIMUM
THEN GOTO 100;
(* *)
WRITELN('O NUMERO ERA ',
NUMBER);

```

```

GOTO 180;
(* *)
170 : WRITELN('CORRETO
EM ', CONTADOR, ' TENTATIVAS
');
180 : (* LINHA VAZIA, DA
DO CICLE 'END' NAO PODE TER
ETIQUETA *)
END.

```

Este programa está escrito de uma maneira bastante confusa, mas a sua versão original em BASIC foi sobejamente copiada. Essa versão em BASIC (original) tinha quase tantos GOTO quanto esta versão em Pascal, à parte a linha

```

IF CONTADOR <= MAXIMUM T
HEN GOTO 100;

```

havendo no programa em BASIC um ciclo a começar imediatamente antes da linha número 100. Esta forma de ciclo e a saída do ciclo com

```

IF ADIVINHA = C_NUMBER TH
EN GOTO 170;

```

não são permitidas por muitas versões de BASIC, e nestas versões teria de usar-se mais um GOTO.

Vamos ser nós o tradutor Pascal, para ver como o programa é tratado.

Depois da declaração dos ficheiros *standard* INPUT e OUTPUT (que são admitidos por omissão em leituras — READ — e escritas — WRITE), a segunda linha declara certas etiquetas (*labels*), que são constantes e não vulgares identificadores. Quando existe um GOTO dentro do programa, o tradutor atribui um valor a um ponteiro para onde está a informação sobre a etiqueta declarada. Esta etiqueta aponta então para o local no código objecto onde está situada a linha pretendida.

Se as etiquetas não estiverem declaradas, poderá haver um salto para uma etiqueta não existente. A declaração das etiquetas torna este erro menos provável, uma vez que as etiquetas vão ser verificadas pelo tradutor. Também, sem a declaração prévia das

etiquetas, o tradutor necessita de duas passagens pelo programa para «atar pontas soltas», tal como saltos para a frente (isto é, saltos para etiquetas que ainda não foram encontradas). Este é o motivo por que muitos assembladores (tal como o do BASIC BBC) necessitam de duas passagens pelo programa fonte em *assembler*.

As outras declarações de identificadores, constantes e variáveis, estão numa forma *standard*, pelo que não apresentam problemas. Na verdade, problemático é saber como foi construído o programa. Este é o ponto onde os GOTOs, geram confusão.

#### «STARS.1»: O PROGRAMA

A primeira linha do programa — depois de BEGIN — depende da implementação, dado que o gerador de números aleatórios não é Pascal *standard*. A forma empregada é razoavelmente *standard* e produz um número inteiro entre 1 e o valor de A (que neste caso é uma constante, 100). Este é o número gerado pelo computador que temos de adivinhar.

Posto a 1 o valor de CONTADOR, o processamento começa com uma tentativa, através da leitura da variável ADIVINHA. Se a tentativa coincidir com o número do computador, o comando passa para a instrução que se encontra na linha com a etiqueta 170. Uma pesquisa ao longo do programa para procurar essa linha conduz-nos próximo do fim do programa.

A instrução é

```
170: WRITELN ('CORRECTO EM',CONTADOR,'TENTATIVAS');
```

Assim, por exemplo, se o valor corrente de CONTADOR for 5, o computador escreverá

```
CORRECTO EM 5 TENTATIVAS
```

O identificador CONTADOR indica o número de tentativas efectuadas. Após a instrução 170, existe uma instrução «vazia» (etiqueta 180), e o programa termina com «END.». Evidentemente, se não acertamos no número, segue-se outra acção, e o programa não prossegue pela etiqueta 170.

Calculada a diferença entre o número a adivinhar e a tentativa efectuada, fica esse valor em DIFERENCA, após o que uma complicada série de comparações e GOTOs fixa o número de asteriscos a escrever. Num dos extremos, se a diferença for maior ou igual a 64, existe um salto para 160. A instrução correspondente a esta etiqueta escreve um asterisco e muda de linha.

Se a diferença for inferior a 64, mas maior ou igual a 32, o salto é para 150, onde se escreve um asterisco sem mudar de linha. Após a execução desta instrução, o comando passa para a linha seguinte, onde se escreve um asterisco com mudança de linha (etiqueta 160).

À medida que a diferença vai diminuindo, o número de asteriscos escritos vai aumentando (note-se que as diferenças são potências de 2, nomeadamente 1,2,4,8,16,32 e 64). Se a diferença for inferior a dois — sendo, portanto, 1 — escrevem-se sete asteriscos, começando com a instrução na linha «etiquetada» com 110.

Depois de ser escrita a quantidade pertinente de asteriscos, a variável CONTADOR é incrementada de uma unidade, e faz-se uma verificação para comparar CONTADOR com o número máximo de tentativas permitidas. Se esse máximo (referenciado pela constante MAXIMUM) for excedido, será executada a instrução

```
WRITELN('O NUMERO ERA ',  
O_NUMBER);
```

e o comando passa para a instrução seguinte, que por sua vez o passa para a linha etiquetada com 180. Esta linha está vazia, isto é, apenas contém «180:», mas é necessária uma vez que a linha final, «END.», não é executável, e portanto não pode ter uma etiqueta associada. Os comentários estão presentes apenas para expandir a linha.



Percorrendo este programa, verificam-se as acções tomadas, dado que o programa não é de imediato evidente.

### A primeira modificação

Grande parte da complicação do programa anterior vem da escrita dos asteriscos. Talvez seja possível sugerir um modo de melhorar o programa, usando talvez um ciclo...

Se usássemos um ciclo, teríamos de inventar uma função que, para uma diferença de 67, por exemplo, produzisse o valor 1 (para 1 asterisco), pelo que o ciclo seria, talvez,

```
FOR I := 1 TO FUN(DIFERENCA)
  DO WRITE('*');
  Writeln(' ');
```

considerando a função FUN como

```
FUNCTION FUN(D : INTEGER)
  INTEGER;
  VAR FUNTEMP : INTEGER;
  BEGIN
    D := 127 DIV D;
    FUNTEMP := 0;
    REPEAT
      FUNTEMP := FUNTEMP +
1)    D := D DIV 2
    UNTIL D = 0;
    FUN := FUNTEMP
  END;
```

Esta função altera o valor da diferença (D) para a divisão inteira de 127 pelo valor original da diferença

127 DIV D	FUNTEMP
127 DIV 64 = 1	1
127 DIV 32 = 3	2

127 DIV 16 = 7	3
127 DIV 8 = 15	4
127 DIV 4 = 31	5
127 DIV 2 = 63	6
127 DIV 1 = 127	7

O ciclo REPEAT/UNTIL pega no resultado dessa divisão inteira e, por meio de divisões sucessivas (agora por 2), estabelece o valor de FUNTEMP, posteriormente a devolver através do nome da função, FUN. Isto parece, realmente, tão complexo como o sistema do programa «Stars\_1».

Podemos substituir estas duas rotinas de escrita de asteriscos (a de «Stars\_1» e a versão que usa FUN e um ciclo FOR) por algo muito mais simples:

```
IF DIFERENCA <> 0 THEN
  BEGIN
    DIFERENCA := 127 DIV DIF
  ERENCA;
    REPEAT
      WRITE('*');
      DIFERENCA := DIFERENCA
    DIV 2
    UNTIL DIFERENCA = 0;
    Writeln(' ');
  END;
```

o que facilmente se colocaria dentro de um procedimento, do seguinte modo:

```
PROCEDURE STAR_UNTIL(D : IN
TEGER);
  BEGIN
    IF D <> 0 THEN
      BEGIN
        REPEAT
          WRITE('*');
          D := D DIV 2
        UNTIL D = 0;
        Writeln(' ');
      END
    END;
```

Um método bastante mais «limpo», e o processo que pode sempre substituir um ciclo REPEAT/UNTIL, é o uso de procedimentos recursivos, ou seja,

```
PROCEDURE STAR_RECURSIVE (D
: INTEGER);
BEGIN
  IF D <= 0 THEN
    BEGIN
      WRITELN('');
      STAR_RECURSIVE(D + 1)
    END
  ELSE WRITELN(D);
END;
```

e, apesar de o procedimento recursivo não ser de execução mais rápida e não gerar menos código, tem uma certa coerência que nos agrada particularmente.

O procedimento chama-se passando como parâmetro o resultado de 127 DIV DIFERENÇA, por exemplo,

```
IF DIFERENÇA <> 0 THEN
  STAR_RECURSIVE(127 DIV DIFERENÇA);
```

Repare-se na verificação para garantir que não existem divisões por zero (a comparação IF DIFERENÇA <> 0 THEN...).

Em versões de BASIC vulgares (o que não é o caso do BASIC BBC ou do SuperBASIC do QL), a recursividade não faz sentido dado não existirem prontamente disponíveis estas facilidades. Valerá a pena lembrar que qualquer noção codificada como um ciclo pode ser codificada de modo recursivo, e vice-versa, o que for codificado recursivamente pode sempre ser codificado iterativamente como um ciclo (se bem que, por vezes, com problemas).

O aparente poder da recursividade apresenta certa redução. É útil manter a noção das proporções; em algumas linguagens, em algumas implementações de tradutores de linguagens e para algumas aplicações, a recursividade é um desperdício total. Para outros fins, a recursividade será ideal. Dado que o Pascal está desenhado como uma linguagem estruturada recursivamente, ele pode ser altamente eficiente.

Lembremo-nos, no entanto, que o compilador de Pascal deve garantir que existam sempre exemplos de rotinas num arranjo recursivo, pelo que após um certo período de tempo o sistema não dispõe de espaço em memória para arrumar os pormenores acerca de nova chamada ao procedimento. Em versões anteriores de Pascal Apple (uma variante do UCSD) os resultados de uma falta de espaço deste tipo eram catastróficos e sem explicação: o sistema ficava completamente transtornado<sup>1</sup> sem dizer onde e porquê a falha ocorrera.

## ATÉ («UNTIL») ACABAR

No programa «Stars», executa-se o processo até se atingir o máximo de tentativas ou se acertar no número. Assim, o programa vê-se melhor como,

```
CONTADOR := 0;
{ }
REPEAT
  READ (ADIVINH);
  DIFERENÇA := ABS (ADIVINH
  - C_NUMBER);
{ }
  IF DIFERENÇA <= 0 THEN
    STAR_RECURSIVE(127 DIV
    DIFERENÇA);
  CONTADOR := CONTADOR + 1
{ }
  UNTIL DIFERENÇA = 0 OR CON
  TADOR = MAXIMUM;
{ }
  IF DIFERENÇA = 0 THEN
    WRITELN('CORRECTO EM ', C
    ONTADOR, ' TENTATIVAS');
  ELSE
    WRITELN('O NUMERO ERA ',
    C_NUMBER);
```

<sup>1</sup> Empregar o termo «transtornado» não é totalmente descabido de realidade, na medida em que existe um compilador de Pascal (Mti) que, perante certos erros de compilação, indica o erro número 999, que significa *completely lost*, ou seja «completamente perdido (baralhado)». (N. de T.)

o que é mais fácil de compreender do que o complicado ciclo de GOTO na versão «Stars — I». O que acontece é que se põe a zero o contador de nome CONTADOR e depois se repete (REPEAT) uma sequência de instruções. O READLN é a tentativa para a variável ADIVINHA e o procedimento STAR\_RECURSIVE encarga-se de escrever a quantidade adequada de asteriscos, caso tentativa não esteja correcta.

O valor do contador é incrementado de uma unidade, o que termina a referida sequência de instruções a repetir. A sequência é repetida (REPEAT) até que (UNTIL) se verifique uma dada condição, neste caso o número a adivinhar (DIFERENCA=0) ou ao atingir-se o número máximo de tentativas (CONTADOR=MAXIMUM).

Quando o ciclo estiver completo, testa-se qual dos dois motivos possíveis o terminou: se (IF) a tentativa estava correcta, então (THEN) somos informados do número de tentativas empregadas, se não (ELSE) dizem-nos qual o número que devíamos ter indicado.

## ENQUANTO («WHILE») FOR VERDADE

Outro modo de encarar o processo de adivinhação do número é considerá-lo sob a forma de «enquanto a diferença não for zero e a quantidade de tentativas não exceder o máximo, escreve asteriscos e volta a tentar». Este método de controle é, neste caso, um pouco menos útil que o UNTIL, dado que, sendo exercido antes de efectuada a acção, são necessárias mais instruções antes do ciclo WHILE. No que diz respeito ao ciclo UNTIL, a acção precede a comparação, pelo que, independentemente do que aconteça, é sempre efectuada pelo menos uma vez.

Contudo, talvez o assunto não seja assim tão linear, eis uma estrutura de controle usando WHILE:

```
READ(ADIVINHA);
DIFERENCA := ABS(ADIVINHA -
  O.NUMERO);
CONTADOR := 1;
{ }
```

```
WHILE DIFERENCA <> 0 AND O
  CONTADOR <= MAXIMUM DO
  BEGIN
    STAR_RECURSIVE(127 DIV
      DIFERENCA);
    CONTADOR := CONTADOR +
  1
  END;
{ }
IF DIFERENCA = 0 THEN
  WRITELN('CORRECTO EM ', O
  CONTADOR, ' TENTATIVAS')
ELSE
  WRITELN('O NUMERO ERA ', O
  NUMBER)
```

e, na realidade, à parte o uso extra de um READLN e o cálculo extra de DIFERENCA, podemos considerar «mais arrumada» esta solução. A verificação de «DIFERENCA igual a zero» simplifica o corpo do bloco de acções do ciclo WHILE; uma vez que já não é necessária a verificação extra de DIFERENCA ser zero (dado que 127 DIV 0 falharia, ocasionando um erro de *overflow*<sup>1</sup>).

Outro ponto interessante a notar são as comparações:

```
UNTIL DIFERENCA = 0 OR CON
  TADOR = MAXIMUM

WHILE DIFERENCA <> 0 AND O
  CONTADOR <= MAXIMUM DO
```

Note-se que a comparação lógica que vem no fim do bloco (ou seja, a comparação feita em UNTIL...) é o complemento lógico da comparação que vem no início do bloco (ou seja, a comparação feita em WHILE...). As acções REPEAT/UNTIL e WHILE/DO são, ao que parece, bastante parecidas, mas têm esta característica interessante que as faz parecer, até certo ponto, «opostas» uma à outra.

<sup>1</sup> Ultrapassar o valor máximo que o computador pode suportar. Dependendo da implementação, este erro (de execução) pode ou não ser dado a conhecer ao utilizador. No caso do MT/1 Pascal, o valor em erro é substituído pela constante MAXINT, sem aviso nenhum. (N. da T.)

## CONTROLE RECURSIVO

O ciclo pode ser «composto» com o uso de chamadas recursivas:

```

PROCEDURE STAR_PROC (CONTADOR
: INTEGER);
{
  VAR ADIVINHA, DIFERENCA
  : INTEGER;
{
  BEGIN
  {
    READ (ADIVINHA);
    DIFERENCA := ABS (ADIVINHA - C_NUMBER);
  {
    IF DIFERENCA <= 0 AND
    CONTADOR <= MAXIMUM
    THEN
      BEGIN
        STAR_RECURSIVE (127
        DIV DIFERENCA);
        STAR_PROC (CONTADOR
        + 1);
      END
    ELSE
      IF DIFERENCA = 0 THEN
        WRITELN ('CORRECTO EM ' || C
        ONTADOR, ' TENTATIVAS');
      ELSE
        WRITELN ('O NUMERO
        ERA ' || C_NUMBER);
      {
    END;
  }

```

onde a extensão do procedimento fica a dever-se principalmente ao uso «generoso» de espaços. O tradutor Pascal ignora linhas em branco e comentários quando forma o código objecto, pelo que o único efeito de grandes espaçamentos é exclusivamente o aumento do código fonte (e da sua clareza).

Existem, circunscritos ao procedimento STAR\_PROC, três identificadores, a saber: o parâmetro CONTADOR, e os inteiros ADIVINHA e DIFERENCA declarados.

Uma vez que são locais, o tradutor esquece-os ao completar o procedimento.

As duas variáveis C\_NUMBER e MAXIMUM englobam todo o procedimento, pelo que as alterações efectuadas sobre elas

permanecem após cada execução do procedimento. Os valores de variáveis globais alteradas mantêm-se e não são esquecidos (embora neste caso não haja alterações).

Quando se compara a estrutura do método de controle de ciclos por procedimentos recursivos com os métodos que usam WHILE ou UNTIL vemos que as parecenças são maiores em relação ao WHILE, se bem que seja possível competir com o ciclo UNTIL.

Não se pretendia competir com o ciclo WHILE quando se escreveu o procedimento recursivo. A parecença surgiu mais devido (como já notámos) à maior simplicidade do bloco WHILE.

Quando se ideou a estrutura recursiva de controle, a simplicidade era um imperativo, pelo que o método recursivo acabou por ficar parecido com o ciclo WHILE. Dadas outras circunstâncias, talvez ficasse parecido com um ciclo UNTIL. A razão principal porque um WHILE é mais simples que um UNTIL, nesta aplicação em particular, é a necessidade de verificar se DIFERENCA é nulo antes de chamar o procedimento STAR\_RECURSIVE. Se sete asteriscos significassem «longe», não haveria necessidade de efectuar esta verificação.

## A CONSTRUÇÃO CASE

Se com a função FUN se modifica DIFERENCA para um número entre 1 e 7, pensar-se-ia que era exequível empregar uma construção de programa com a seguinte forma:

```

CASE FUN(DIFERENCA) OF
  1 : { LINHA FICTICIA };
  2 : UM_ASTERISCO;
  3 : DOIS_ASTERISCOS;
  4 : TRES_ASTERISCOS;
  5 : QUATRO_ASTERISCOS;
  6 : CINCO_ASTERISCOS;
  7 : SEIS_ASTERISCOS;
  8 : SETE_ASTERISCOS;
END; { FIM DA CONSTRUÇÃO DA

```

onde, por exemplo, as definições dos procedimentos podiam ser

```
PROCEDURE UM_asterisco;
BEGIN
  WRITE('1*');
END;

PROCEDURE DOIS_asteriscos;
BEGIN
  WRITE('2**');
  UM_asterisco;
END;

.....

PROCEDURE SETE_asteriscos;
BEGIN
  WRITE('7*****');
  SEIS_asteriscos;
END;
```

Se, por algum azar, o valor produzido por FUN(DIFERENCA) for superior a 7, em Pascal *standard* temos um erro. Em Pascal Hisoft e diversos outros, esta «contra» é resolvido através da incorporação de uma nova linha. Em Pascal Hisoft existe uma outra instrução de nome ELSE<sup>1</sup>:

```
CASE FUN(DIFERENCA) OF
  0 : LINHA_FICTICIA 1;
  1 : UM_asterisco;
  2 : DOIS_asteriscos;
  3 : TRES_asteriscos;
  4 : QUATRO_asteriscos;
  5 : CINCO_asteriscos;
  6 : SEIS_asteriscos;
```

<sup>1</sup> Em MIT4 Pascal o nome é idêntico. Tendo-se que cada bloco executável associado a um dos valores possíveis da variável de selecção do CASE é um «ramo», o bloco associado ao ELSE toma o nome de *default branch* (ramo por defeito) e cobre todas as outras possibilidades da variável de selecção não enumeradas.

```
7 : SETE_asteriscos;
< >
ELSE WRITELN('ERRO: NO VALOR
  A INTRODUCIDO'); CNAD E STR
  NDOARD;
< >
END; (FIM DA CONSTRUCAO DA
  EEE)
```

O CASE é uma versão compactada de uma extensa sequência de IFs:

```
D := FUN(DIFERENCA);
IF D = 0 THEN < > ELSE
IF D = 1 THEN UM_asterisco
ELSE
IF D = 2 THEN DOIS_aster
iscos ELSE
IF D = 3 THEN TRES_aste
ricos ELSE
IF D = 4 THEN QUATRO_A
steriscos ELSE
IF D = 5 THEN CINCO_A
steriscos ELSE
IF D = 6 THEN SEIS_A
steriscos ELSE
IF D = 7 THEN SETE_
asteriscos ELSE
  WRITELN('ERRO: NO V
  ALOR INTRODUCIDO');
```

Isto mostra porque é que não gostamos do uso de CASE como estrutura de controle, dado que parece demasiado complexo e incontrolado. Existe normalmente um modo melhor.

Em alguns casos, os outros tipos escalares, a construção CASE ajuda à compreensão. Por exemplo

```
CASE POR_PRIORIDADE_DO_LUIDO
OF
< >
VERMELHO : ACCAO_1;
VERDE : ACCAO_2;
AZUL : ACCAO_3;
< >
END;
```

De modo geral, o uso do CASE parece demasiado intrincado.

## O PROGRAMA STARS\_2

Apresentamos agora uma versão melhorada do programa, que tem em conta as considerações anteriores e mostra como consegue ajudar a estruturação dos dados, embora em menor escala.

Existem ligeiras modificações sobre o programa anterior (por exemplo o novo tipo JOGO\_INT) mas, essencialmente, não há nada de novo. Não daremos explicações sobre o programa, deixando ao cuidado do leitor descobrir o que se passa.

```
PROGRAM STARS_2 (OUTPUT, INF
UT
)
CONST A = 100;
MAX = 2;
TYPE JOGO_INT = 0..A;
VAR C_NUMBER : JOGO_INT;
[-----]
PROCEDURE STARS (D : JOGO_I
NT);
BEGIN
  IF D > 0 THEN
    BEGIN
      WRITE('4');
      STARS(D DIV 2);
    END
  ELSE WRITELN(' ')
  END;
[-----]
PROCEDURE JOGO (CONTADOR :
JOGO_INT);
VAR ADIVINHHA, DIU : JOGO
_INT;
BEGIN
  READ(ADIVINHHA);
  DIU := ABS(ADIVINHHA -
C_NUMBER);
  IF DIU < 0) AND (CONT
ADOR < MAX) THEN
    BEGIN
      STARS(12 * DIU DIV 2);
    END
  ELSE
    IF DIU = 0 THEN
      WRITELN('CORRECTO
EM ', CONTADOR, ' TENTATIVAS
')
    ELSE
      WRITELN('O NUMERO'
```

```
ERR C_NUMBER);
END;
[-----]
BEGIN & PROGRAMA PRINCIPAL
)
(
)
C_NUMBER := RANDOM(A);
JOGO(1);
(
)
END.
```

## A MAIS PODEROSA CONSTRUÇÃO

O procedimento, seguido de perto pela função, é a construção mais poderosa do Pascal.

Para progredir na programação de qualquer linguagem, devemos adquirir um raciocínio sistemático, que dê ênfase à divisão em blocos do programa. Pensar em termos de procedimentos e funções ajuda a produzir programas de compreensão simples. Normalmente, programas de compreensão fácil são também programas de fácil correcção (os erros tendem a ser pequenos) e fácil modificação.

Voltando atrás no livro, notar-se-á que ele começou com o conceito de tradutor de Pascal, e depois passou para procedimentos. Foi apenas depois de estudados os tipos de dados que o texto começou a examinar as estruturas de comando *standard*. Tal como já foi discutido antes, as estruturas de comando *standard* em Pascal constituem a característica menos importante da linguagem.

A característica mais importante do Pascal são o programa e o modo como o código fonte se transforma no código objecto correspondente. Uma vez compreendido o programa e a sua tradução, podemos passar para o estudo das secções que formam um programa. Em qualquer programa concebido com sensatez, as principais secções são as rotinas (procedimentos e funções).

Muitos programadores aprenderam já há anos a importância da rotina, numa linguagem chamada FORTRAN IV. O uso de rotinas mostrava-se tão importante em FORTRAN IV por esta linguagem ser, em muitos aspectos, uma linguagem primitiva,

com estruturas de comando mais primitivas que muitas versões de BASIC. Para assegurar que os programas em FORTRAN IV tivessem menos hipóteses de conter erros, e para ajudar a isolar esses erros quando ocorressem, tornavam-se os programas em FORTRAN IV um conjunto de rotinas.

O Pascal tem um conjunto de estruturas de comando mais rico e substancialmente mais útil que o FORTRAN IV, pelo que muitos que escrevem sobre Pascal tendem a começar pelas estruturas de comando. Vindos do FORTRAN IV (que não é o mesmo que FORTRAN 77), a nossa reacção inicial é sempre começar pelo topo (ou seja, pelo tradutor) e deslocar-nos depois para o fundo (ou seja, as estruturas de comando).

Será interessante notar que muitas descrições de Pascal começam pelo fim, as estruturas de comando, e muitas vezes não atingem o começo, o tradutor.

Um dos conceitos em voga em programação é conhecido como programação «estruturada» (também conhecida como programação *top down*): a definição de programação *top down* implica que o programador proceda à definição global do que pretende fazer antes de entrar em pormenores.

Devido ao modo como funciona o tradutor de Pascal (que requer informação sobre os pormenores antes de poder usar a informação), deve existir uma boa quantidade de «meditação e trabalho no papel» antes de efectivamente se escrever o programa em Pascal. Este é, como em FORTH, traduzido do fim para o princípio e requer, também como o FORTH, bastante trabalho prévio.

Muitas pessoas que escrevem sobre Pascal tendem a ser *top down* apenas superficialmente e são na realidade *bottom up* (inverso de *top down*), o que se observa na ordem pela qual apresentamos os diagramas sintácticos no capítulo 6.

## A sintaxe do Pascal

A ordem e a simplificação são os primeiros passos em direcção do domínio de um assunto — o verdadeiro inimigo é o desconhecido.

*A Montanha Mágica*, de Thomas Mann

O funcionamento do tradutor do Pascal é crucial para uma compreensão do funcionamento e raciocínio do Pascal, enquanto linguagem. Este capítulo permite-nos estudar com mais pormenor a diferença de significados entre «sintaxe» e «semântica» e a confusão que isto pode causar. Neste ponto, será útil a consulta à secção do capítulo 2, «Análise de um programa em Pascal».

Numa linguagem vulgar, a «sintaxe» liga-se ao modo como as palavras se juntam para formar frases, e a «semântica» liga-se ao estudo dos significados das palavras e das frases. A sintaxe não pode ser confundida com a semântica, dado que ambas existem independentemente uma da outra em muitos aspectos. Todos sabemos que aquilo que «dizemos» não é necessariamente aquilo que «queremos dizer».

Em linguagem de computador, a sintaxe está relacionada com as regras da linguagem e com as formas permitidas pela linguagem. A semântica de uma linguagem de programação depende daquilo que a sintaxe dessa linguagem permite, e não existe independência. A «rigidez» da semântica de uma linguagem de programação contrasta com a «fluidez» da semântica das linguagens vulgares.

Aqui é onde pode surgir confusão. O nosso conhecimento sobre o que dizem os significados é aquele que a linguagem normal (ou seja, a semântica da nossa linguagem natural) nos dá. Quando se aplica este conhecimento à semântica de uma linguagem de programação, surgem provavelmente ambiguidades nos significados de estruturas de programas, dado que isto está na nossa natureza.



Para ilustrar este ponto, isolámos duas «confusões», se bem que nenhuma delas o seja, realmente, no que toca ao tradutor da linguagem. A primeira confusão é produzida pela nossa habilidade para construir múltiplos significados a partir de quase nada, na nossa linguagem natural; a segunda confusão está relacionada com a falta de certeza dos que concebem as linguagens quanto ao que devem fazer.

### CONFUSÃO 1: IF THEN IF THEN ELSE

Examinemos esta porção de texto de programa:

```
IF condição-1 THEN IF condição-2 THEN acção-1
ELSE acção-2;
```

Tentemos estabelecer o seu significado. Aplicando o nosso sentido comum (o da linguagem natural) em concepções de significado, vemos que podemos dar duas interpretações. Qualquer delas pode ser a pretendida:

```
IF condição_1
THEN
  BEGIN
    IF condição_2 THEN
      acção_1 ELSE
      acção_2
  END;
```

```
IF condição_1
THEN
  BEGIN
    IF condição_2 THEN
      acção_1
    END
  ELSE acção_2;
```

ou seja, o IF.SP. tanto se associa ao primeiro THEN como ao segundo THEN.

Vemos o conflito na interpretação (que pode ficar bem confusa) dado estamos a aplicar conceitos de semântica da linguagem natural da semântica de uma linguagem de computador. Para uma linguagem de computador, a semântica é dada pela sintaxe, visto que é esta que decide o que o tradutor deve aceitar como correcto. No que diz respeito ao tradutor, só existe uma interpretação aceitável; se fossem permitidas várias interpretações, a nossa confiança na linguagem ficaria um tanto ou quanto abalada.

A sintaxe geral da instrução IF em Formalismo Allan (AF) — veja-se apêndice C para mais detalhes — é:

```
[instrução-IF] —>
<IF> [condição booleana]
<THEN> [instrução]
{*<ELSE> [instrução]*}
```

cujas interpretação significa que uma instrução IF é composta pela palavra IF seguida de uma condição booleana (por exemplo, DIFERENÇA>0). A condição é sempre seguida pela palavra THEN e uma ou mais instruções, havendo em opção a palavra ELSE com as instruções associadas.

Os parênteses rectos [] rodeiam um item da linguagem, os sinais <> rodeiam texto de programa sob a forma como na realidade aparece, e o conjunto {\*} rodeia uma sequência à escolha (ver no capítulo 2: «Análise de um programa em Pascal» para um breve exame do AF; valerá a pena comparar com o diagrama sintáctico da instrução IF).

A instrução IF é, na realidade, um exemplo de uma instrução, pelo que podemos incluí-lo na sequência após THEN, ou seja:

```
[instrução-IF] —>
<IF> [condição booleana]
<THEN> [instrução-IF] ; uma possibilidade :
{*<ELSE> [instruções]*}
```

O que está entre : ; é um comentário, não fazendo portanto

parte da descrição sintáctica. Esse arranjo pode ser expandido (recursivamente) substituindo a porção instrução - IF pela sua própria definição, tendo-se então

```
[instrução - IF] —>
<IF> [condição - booleana]
<THEN>
  <IF> [condição - booleana]
  <THEN> [instrução]
  [* <ELSE> [instrução] *]
  [* <ELSE> [instrução] *]
```

vendo-se que a sequência produzida por uma instrução IF imediatamente a seguir a um THEN é indicada por

IF THEN IF THEN ELSE ELSE

sendo os ELSE facultativos. No entanto, se omitirmos apenas um deles, voltamos à «confusão» original. Examinando a forma AP do IF composto, vemos que o tradutor assume a ordem

```
[instrução - IF] —>
<IF> [condição - booleana]
<THEN>
  <IF> [condição - booleana]
  <THEN> [instrução]
<ELSE> [instrução]
```

dado que o tradutor espera que o ELSE se refira ao IF imediatamente anterior.

Observa-se uma equivalência importante no Pascal, usando AP do seguinte modo:

[acção - X] = <BEGIN> [acção - X] <END>

ou seja, colocar qualquer instrução Pascal entre as palavras reservadas BEGIN e END não tem qualquer efeito sobre a

instrução, servindo para delimitar o início e o fim de um bloco de instruções a efectuar assim, e se o ELSE devesse referir-se ao primeiro IF, ou seja

```
[instrução - IF] —>
<IF> [condição - booleana]
<THEN> [instrução - IF - sem - ELSE]
<ELSE> [instrução] : outra possibilidade;
```

já só é necessária uma pequena modificação para produzir

```
[instrução - IF] —>
<IF> [condição - booleana]
<THEN> <BEGIN> [instrução - IF - sem - ELSE]
<END>
<ELSE> [instrução]
```

o que, expandido, fica

```
[instrução - IF] —>
<IF> [condição - booleana]
<THEN>
  <BEGIN>
    <IF> [condição - booleana]
    <THEN> [instrução]
  <END>
  <ELSE> [instrução]
```

Estas manipulações reflectem-se, como se vê, nos programas correspondentes em Pascal (usando condição\_1, condição\_2, acção\_1 e acção\_2).

Chamamos «ELSE pendente» a este problema, e ocorre em muitas outras linguagens. Isto é particularmente verdade naquelas linguagens cujo estilo veio, em parte, de uma linguagem anterior chamada ALGOL 60, dado que tendem a usar as mesmas estruturas de comando.

O problema do ALGOL 60 não existe para uma linguagem

com um nome parecido, o ALGOL 68. A semântica do ALGOL 68 não está aberta a essa dúvida, do ponto de vista da semântica natural, porque todas as construções (tais como o IF) são delimitadas fisicamente. Por exemplo, um IF termina com FI.

### Confusão 1: alguns comentários

Se bem que a confusão 1 não seja, na realidade, uma «confusão» no sentido mais estrito, a sua existência indica que se pode melhorar e tornar mais «fechado» o modo como o Pascal está estruturado.

A «complexa» linguagem ALGOL 68 era na verdade muito simples na sua concepção, e a complexidade aparente derivava do facto de que a sua simplicidade era produto de um pesado investimento em tradutores complicados. Já dissemos que, quanto mais simples nos parecer um sistema (dados sistemas de potências equivalentes), tanto maior será o investimento necessário no *software* requerido para o suportar.

Para conceber um sistema simples de usar e que, mesmo assim, constitua uma potente ferramenta de programação, requer-se uma quantidade muito grande de *software* especificamente desenvolvido.

O Pascal assume duas coisas neste ponto: 1) o utilizador é competente e 2) o *software* para executar o problema deve ser tão simples quanto possível. A combinação destes dois pressupostos significa que o tradutor é tão simples quanto possível, e que o utilizador deve aprender mais sobre a estrutura do tradutor do que o necessário em outras linguagens. Se bem que seja raro estudar o tradutor, a sua importância faz-se sentir na necessidade da presença dos diagramas sintácticos em qualquer texto Pascal.

Nota-se que, no desenvolvimento da linguagem Modula-2, Wirth se aproximou mais do tipo de sintaxe do ALGOL 68 (ver o apêndice A).

### CONFUSÃO 2: ATÉ QUE «DISTÂNCIA» USAR FORWARD?

Em alguns casos é difícil estabelecer a ordem precisa em que os procedimentos devem ser declarados ao tradutor, de tal modo que nenhum seja usado antes de ser declarado. Isto é particularmente verdadeiro quando existem procedimentos que chamam procedimentos que por sua vez chamam outros procedimentos, e assim por diante. Noutros casos, esta ordem não pode ser estabelecida. Considere-se um procedimento A que em dado ponto contém uma chamada ao procedimento B. Dentro de B existe uma chamada a C, fazendo C uma chamada a A. Qual dos três procedimentos deve estar primeiro? A solução é simples. Declarar-se todos os três como o complemento FORWARD, e a ordem deixa de importar:

```
[declaração -forward] —>  
<PROCEDURE> [identificador] [parâmetros] <;>  
<FORWARD> <;>
```

Depois, quando se definir o conjunto de instruções do procedimento, o formato correcto será:

```
[Definição -do -procedimento -após -FORWARD] —>  
<PROCEDURE> [identificador] <;>  
[bloco -normal -do -procedimento] <;>
```

não se repetindo os parâmetros na segunda menção no identificador do procedimento.

Isto constitui base de confusão, reclamando algumas pessoas que na segunda referência ao procedimento deveriam existir os parâmetros que eventualmente existissem na primeira. Esta confusão deve-se, principalmente, à especificação original de Wirth: o uso do FORWARD assemelha-se muito a uma reflexão tardia. Vê-se isto claramente nos diagramas sintácticos que constam da segunda edição do livro de Wirth: *Pascal: User*

*Manual and Report*<sup>1</sup>, e ainda porque FORWARD não surge no *Report* (relatório) (Ver apêndice G).

A única menção feita no manual restringe-se a este exemplo (e, por outro lado, só surge como um de quatro comentários no final de um capítulo sobre procedimentos e funções).

```
PROCEDURE MIX : TI; FORWARD
;
PROCEDURE MIX : TI;
BEGIN
  ...
END;
PROCEDURE W; 2 OS parâmetros
DE SÃO ESTOS (PARÂMETROS)
BEGIN
  F(b);
END;
BEGIN
  P(a);
  C(b);
END;
```

Pessoalmente, não cremos de modo algum na clareza do mecanismo destes dois procedimentos, dado que não acontece nada, além do que o computador eventualmente fica com a memória cheia, após um ciclo sem fim de chamadas de procedimento. Parece muito um gesto de abandono da parte de Wirth e do co-autor, Kathleen Jensen.

Se definirmos

[identificação-de-procedimento] —>  
<PROCEDURE> [identificador]

a sintaxe de um procedimento define-se no manual do utilizador (*User Manual*), como

[declaração-de-procedimento] —>  
[identificação-de-procedimento] [parâmetros] <:>  
[bloco] <:>

<sup>1</sup> *Pascal: Guia do Utilizador e Relatório*. (N. do T.)

não havendo menções a FORWARD; e, de facto, não contém tal declaração. Obviamente, FORWARD está fora do esquema original e, se bem que se discuta normalmente em textos de Pascal a declaração FORWARD, os diagramas sintácticos normalmente fornecidos com esses textos não referem a existência de FORWARD.

Certas pessoas argumentavam que, no processo de definição do Pascal *standard*, na segunda referência ao procedimento, os parâmetros deviam ser repetidos. Este ponto de vista perdia, dada a natureza dos identificadores de procedimentos em Pascal.

Na primeira menção ao identificador do procedimento (aquele que contém FORWARD) o nome do identificador e certas características (por exemplo as dos parâmetros) são armazenadas junto com um ponteiro (por ajustar) para o código objecto (ainda não gerado). Ao encontrar o identificador pela segunda vez, não é necessário repetir os parâmetros, dado que a informação sobre eles já está em memória junto com o ponteiro (se bem que o ponto de vista era que a repetição ajudaria à segurança).

A segunda menção à identificação do procedimento serve para produzir o código objecto inerente a esse procedimento, de maneira que basta ajustar o ponteiro antes referido para o início deste segmento de código (similar aos «vectores de execução» em FORTH).

## Confusão 2: alguns comentários

A ideia original de Wirth de ter um tradutor de um só passo (uma só passagem pelo texto), se bem que impressionantemente económica em execução para fins *standard*, gera estas confusões por poder produzir mais problemas, alguns dos quais não foram previstos.

Na BS 6193, trata-se FORWARD como um exemplo especial de uma directiva, sendo FORWARD a única directiva requerida. E muitos sistemas de Pascal existem outras directivas para procedimentos, tais como EXTERNAL (para um procedimento

de uma livraria Pascal)<sup>1</sup>, ou FORTRAN (para um procedimento de uma livraria FORTRAN).

É um reflexo das origens do Pascal que, nas especificações originais da linguagem, não existam menções ao uso de livrarias de programas. Cada programa era como uma entidade independente, isolada dos outros programas: à parte os identificadores *standard*, não existia maneira de usar procedimentos contidos em livrarias especializadas<sup>2</sup>. Este problema tornou-se visível quando se implementou o sistema Pascal UCSD (variante Apple Pascal) no capítulo sete.

Como um fã de FORTH, posso ver que muitos problemas da «passagem única» do Pascal estão presentes em FORTH, e a solução em Pascal tem muito em comum com a solução em FORTH. No entanto o FORTH tem a vantagem de ser totalmente transparente para o utilizador informado. Talvez haja algo a dizer quanto a uma menor estreiteza de espírito da parte do utilizador e desenhistas de linguagens de programação.

<sup>1</sup> Uma livraria é um modo especial de organizar programas (rotinas). Começa com um índice dos programas que contém, seguido desses programas. Quando após a compilação do programa chamador se lhe junta o módulo «livraria», para encontrar um procedimento, faz-se uma pesquisa do índice e, se o procedimento existir, é copiado. Os procedimentos e/ou funções que constam da livraria deverão estar previamente compilados.

<sup>2</sup> Não é obrigatório que seja uma livraria em M+/+ Pascal, pode-se incluir um procedimento externo (declarado com EXTERNAL) num módulo feito pelo utilizador, que não obedece aos requisitos específicos de uma livraria.

## Programação em Pascal

No fim de contas, e chegando ao cerne da questão, quantas pessoas falam a mesma linguagem, mesmo quando falam a mesma língua?

*The Lion of Bonz-Jachin and Jachin-Boaz*, de Russell Hoban

Este capítulo é dedicado a um único programa: um programa em Pascal Apple para desenhar «flocos de neve».

Empregámos a variante Pascal Apple do UCSD uma vez que mostra como temos de utilizar métodos *ad hoc* para chegar a procedimentos de livraria, a fim de ultrapassar as deficiências na especificação original do Pascal. A forma da linguagem no Pascal Apple é suficientemente diferente do Pascal «vulgar» nos aspectos que o tornam esolarecedor, sem ser excessivamente diferente da versão *standard*.

### A CURVA DO FLOCO DE NEVE: ROTINAS

Para desenhar a curva do floco de neve procedemos do seguinte modo:

1. Começamos com um triângulo equilátero (a ORDEM=0).
2. Dividimos cada lado em três secções. No terço intermédio, construímos um triângulo equilátero, dirigido para fora do triângulo original (com ORDEM=ORDEM+1).
3. Se o valor de ORDEM não for suficientemente grande, repetimos a acção 2.

Os três primeiros flocos de neve estão representados nas figuras 7.1 a 7.3. Podemos ver a acção 2, em conjunção com a acção 3, quer como recursiva quer como iterativa (isto é, quer usando rotinas recursivas quer usando GOTO). Escolhemos a abordagem recursiva.

O Pascal Apple (tal como versões de Pascal UCSD, e também o Pascal Hiosfi) desenha usando rotinas de «tartaruga» gráfica, um método muito recomendável. Para desenharmos uma linha, especificamos a direcção, ponto de início e comprimento da linha a desenharmos. Sobre a teoria da «tartaruga» gráfica escrevemos já uma obra, *Introducing LOGO*.

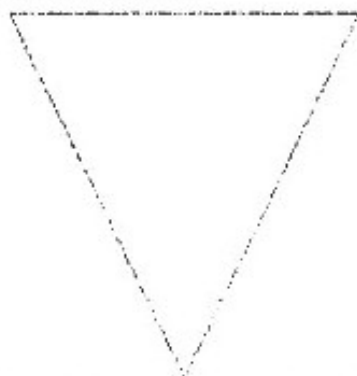


Figura 7.1. Floco de neve de ordem de ordem 0.



Figura 7.2. Floco de neve de ordem de ordem 1.

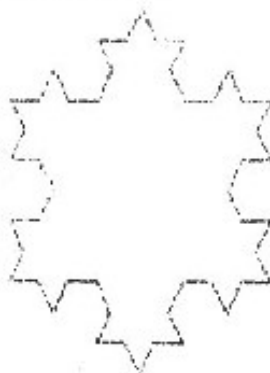


Figura 7.3. Floco de neve de ordem de 2.

O floco de neve baseia-se num triângulo equilátero. Para desenharmos um triângulo equilátero em gráficos «tartaruga», esta move-se para a frente uma distância fixa e roda 120 graus; a tartaruga mover-se-á novamente a mesma distância e rodará ainda 120 graus, para repetir estes dois passos uma segunda e uma terceira vez.

Isto sugere um procedimento para delinear o triângulo:

```
PROCEDURE FLOCO_DE_NEVE (LADO : REAL; CINCOPLETOS)
BEGIN
  DESENHA_LADO (LADO, 120);
  DESENHA_LADO (LADO, 120);
  DESENHA_LADO (LADO, 120);
END;
```

que dá conta do recado. Ainda não sabemos desenharmos um lado, mas já começámos. O comprimento do lado do floco de neve a desenharmos é dado pelo identificador real LADO, e esta informação é passada ao procedimento DESENHA\_LADO.

Com o procedimento DESENHA\_LADO há um problema: não existe maneira de decidir qual o tipo de floco a desenharmos (ou seja, a ordem do floco de neve). Temos assim de considerar mais um parâmetro: a ordem da curva. Esta informação deve ser passada a DESENHA\_LADO:

```
PROCEDURE FLOCO_DE_NEVE (ORDEM : INTEGER; LADO : REAL)
BEGIN
  DESENHA_LADO (ORDEM, LADO, 120);
  DESENHA_LADO (ORDEM, LADO, 120);
  DESENHA_LADO (ORDEM, LADO, 120);
END;
```

Vamos agora definir o conteúdo de DESENHA\_LADO.\*

Quando estamos a desenharmos um lado, ele pode ser uma linha

recta, ou seja, ter ordem zero, ou pode ter um triângulo no meio, e possivelmente triângulos sobre triângulos (ordem superior a zero). Esta definição parece ser simples.

```
PROCEDURE DESENHA_LADO (ORD
EM: INTEGER; LADO: REAL);
BEGIN
  IF ORD = 0
  THEN DESENHA_TRI (ORD
EM, LADO)
  ELSE MOVE (ROUND (LADO));
  TURN (ANGULO);
END;
```

que, basicamente, diz que, se ORDEM for diferente de zero, se desenha a linha triangular, se não é desenhada uma linha recta. Em qualquer dos casos faz-se uma viragem no fim da linha.

A rotina implícita da «tartaruga» gráfica MOVE, ao fazer MOVE(DIST), move a tartaruga de DIST unidades para a frente, onde DIST deve ser uma quantidade inteira, e é por isso que se faz ROUND(LADO). A função ROUND(X) arredonda o valor de X para o inteiro mais próximo<sup>1</sup>.

A rotina TURN(GRAUS) muda a direcção da tartaruga fazendo-a rodar uma quantidade inteira de graus dada por GRAUS e variando entre zero e 359 no sentido anti-horário, correspondendo os zero graus à direcção «em frente para a direita».

A linha triangular desenha um terço de um lado, e roda -60° (ou seja, rotação horária); traça uma linha e a volta no final é de 120°; a linha seguinte tem uma volta de -60°; a última linha não tem volta, e a ordem é decrementada de uma unidade. Juntando tudo, definimos DRAWTRI, que irá ser

```
PROCEDURE DESENHA_TPI (ORDE
M: INTEGER; LADO: REAL);
BEGIN
```

```
  DESENHA_LADO (ORD = 1,
-60, LADO/3);
  DESENHA_LADO (ORD = 2,
120, LADO/3);
  DESENHA_LADO (ORD = 3,
-60, LADO/3);
  DESENHA_LADO (ORD = 1,
0, LADO/3);
END;
```

O motivo pelo qual se fez a variável LADO de tipo real é que as sucessivas divisões por três, usando uma variável inteira, originariam demasiados erros de arredondamento. Por exemplo, 17/9 seria 2 (o inteiro mais próximo), mas 17 DIV 9 é 1; mesmo que façamos (17 DIV 3) DIV 3 a resposta é 5 DIV 3, ou seja 1.

Como se pode notar, DESENHA\_LADO chama DRAWTRI, que por seu lado chama DESENHA\_LADO. Há a necessidade de, pelo menos, uma declaração FORWARD.

## CURVA «FLOCO DE NEVE»: O PROGRAMA

O floco de neve tem de ser desenhado algures no *écran* de vídeo, e o *écran* do Apple II, em Pascal, tem 280 unidades de largura por 192 unidades de altura. Temos de tomar decisões sobre: onde colocar a tartaruga no início, qual o tamanho do lado inicial do triângulo (ou seja, o lado do triângulo de ordem zero) e qual a orientação inicial da tartaruga.

Se ALTURA for a altura do *écran*, e LARGURA a sua largura, o ponto inicial para a tartaruga deve ser a meio do *écran* (na horizontal) e a um oitavo do fundo do *écran* (na vertical). A origem dos gráficos *turtle* situa-se no canto inferior esquerdo do *écran*.

A tartaruga deverá estar orientada na direcção 30 (ou seja, rodada de 30° para a direita).

A sequência de que necessitamos é:

```
INITTURTLE;
MOVETO (LARGURA DIV 2, ALTU
RA DIV 8);
TURNTO (30);
```

<sup>1</sup> Os arredondamentos são diferentes para números positivos e negativos. (N. do T.)



que inicializa a «tartaruga» gráfica para se poder começar a desenhar, move a tartaruga para metade do *Screen* na horizontal e um oitavo na vertical, e roda a tartaruga para a direcção correcta.

A outra informação necessária é o lado inicial do triângulo. Dado que desejamos que a forma seja simétrica, usamos, convenientemente, um pouco de trigonometria:

$$LADO\_INICIAL := (ALTURA * 9 / 16) * (SQRT(3) / 2),$$

(note-se que  $SQRT(3)/2$  é igual ao co-seno de  $30^\circ$ ). Podemos, obviamente, simplificar esta expressão.

Perguntar-se-á o porquê de todas estas contorções, porque não definir, pura e simplesmente, os valores numericamente? A razão de não usar atribuições deve-se à facilidade de posteriores alterações, para compatibilidade com outros sistemas dispondo da «tartaruga» gráfica.

Eis, portanto, o programa:

```
PROGRAM DEMONSTRALAD_DO_FL
ODO_DE_NEVE;

USES TURTLEGRAPHICS; (* LI
VRARIA DE ROTINAS PARA GRA
FICOS *)

CONST LARGURA = 280;
      ALTURA = 192;
      ORDEN_MAXIMA = 6;
VAR ORDEN : INTEGER;
    LADO_INICIAL : REAL;

PROCEDURE DESENHA_LADO(ORDEN : INTEGER; LADO : REAL);
FORWARD(LADO);
PROCEDURE DESENHA_TRI(ORDEN : INTEGER; LADO : REAL);
FORWARD(LADO);
PROCEDURE FLDO_DE_NEVE(ORDEN : INTEGER; LADO : REAL);
BEGIN
  DESENHA_LADO(ORDEN, LADO);
  DESENHA_LADO(ORDEN, LADO);
```

```
  LADO := DESENHA_LADO(ORDEN, LADO);
END;
PROCEDURE DESENHA_TRI(ORDEN : INTEGER);
BEGIN
  IF ORDEN > 0
  THEN DESENHA_TRI(ORDEN - 1, LADO)
  ELSE MOVE(ROUND(LADO * COS(30)), ROUND(LADO * SIN(30)));
  TURN(ANGULO);
END;
PROCEDURE DESENHA_TRI(ORDEN : INTEGER);
BEGIN
  DESENHA_LADO(ORDEN - 1, LADO);
  DESENHA_LADO(ORDEN - 1, LADO);
  DESENHA_LADO(ORDEN - 1, LADO);
  DESENHA_LADO(ORDEN - 1, LADO);
  DESENHA_LADO(ORDEN - 1, LADO);
END;
BEGIN (* PROGRAMA PRINCIPAL *)
  REPEAT (* ENTRADA DA ORDEN *)
  UNTIL ORDEN < 0;
  REPEAT (* ENTRADA DA LARGURA *)
  UNTIL LARGURA < 0;
  REPEAT (* ENTRADA DA ALTURA *)
  UNTIL ALTURA < 0;
  INIT(TURTLE);
  MOVETO(LARGURA DIV 2, ALTURA DIV 2);
  TURN(30);
  PENCOLOR(BLACK);
  LADO_INICIAL := (ALTURA * 9 / 16) * (SQRT(3) / 2);
  FLDO_DE_NEVE(ORDEN, LADO_INICIAL);
  READLN(ORDEN); (* PERMITE QUE A TIGURA PERMANECA NO ECRAN *)
  TEXTMODE (* VOLTAR AO MODO TEXTO *)
END;
```

Já analisámos os procedimentos que, à parte as duas declarações FORWARD, não têm nada de excepcional. A seguir ao identificador do programa (note-se a ausência de parâmetros), surge uma novidade do Pascal: a declaração USES.

Enquanto alguns sistemas Pascal, para utilizar procedimentos de livreria, usam a declaração EXTERNAL, em Pascal Apple essa declaração toma a forma USES, dizendo que o programa usa rotinas de uma livreria específica (neste caso, de TURLE-GRAPHICS).

As rotinas usadas (por exemplo MOVETO ou INITTURTLE) podiam, noutros sistemas, ser introduzidas por

```
PROCEDURE MOVETO(X, Y : IN  
  TREAL); EXTERNAL;  
PROCEDURE INITTURTLE; EXTE  
  RNAL;
```

dependendo do sistema.

O resto do programa é bastante claro, e as pequenas diferenças (tais como «o que é INITTURTLE?») não são importantes. Este é o motivo por que não tratámos com grande minúcia uma implementação particular. As versões de Pascal *standard* variam entre si, bem como as de Pascal UCSD, mas o que é importante são as noções que levaram à criação da linguagem.

O único modo de ver qual o significado exacto de qualquer instrução Pascal numa implementação particular é examinar a documentação sobre essa mesma implementação.

## Capítulo 8

# Perspectivas do Pascal

As coisas são sempre melhores no início

*Cartas Provinciais, de Blaise Pascal*

Muitas das versões de Pascal para microcomputadores revelam-se bastante completas: por exemplo, a maior parte das versões de CP/M, tal como MT + Pascal, as diferentes versões de Pascal UCSD e as menos divulgadas, tais como as da Hisoft.

A maior parte das diferenças, normalmente, provém do uso de ficheiros para entrada (*input*) e saída (*output*) de dados.

Por exemplo, veja-se o desprezo de parâmetros do programa para ficheiros no UCSD e em muitas outras Pascal de microcomputadores, bem como a inclusão de um novo tipo de ficheiro (INTERACTIVE) no Pascal UCSD.

Em algumas versões mais simples (muitas vezes chamadas «mini» Pascal) existem diferenças radicais; por exemplo, para poupar escrita e espaço, PROCEDURE passa a PROC, FUNCTION passa a FUNC e assim por diante. É normal que o «mini» Pascal não permita a formação de novos tipos de dados, e o tipo *pointer* não esteja implementado.

É possível, no Pascal *standard*, ter uma declaração de *procedure* como a seguinte:

```
PROCEDURE Q ( PROCEDURE F; PROCEDURE G );
```

onde se passam como parâmetros da *procedure* Q as duas *procedures* F e G: o corpo da *procedure* Q poderia então ser:

```
PROCEDURE Q ( PROCEDURE F; PROCEDURE G );  
BEGIN  
  F;  
  G;  
END;
```

Em Pascal UCSD isto não é possível, embora existam maneiras de contornar esta restrição.

O Pascal teve grande impacto e um computador (o *Apple III*) foi desenhado como uma «máquina Pascal» (ver o apêndice D). Com as suas influentes ideias de estruturação (compartilhadas em muitas outras linguagens), o Pascal foi visto como um modo de projectar sistemas complexos que funcionassem. Acidentalmente, no entanto é C a linguagem mais usada para desenvolver sistemas deste tipo<sup>1</sup>; funciona de um ponto de vista totalmente diferente.

O Pascal não se mostrou totalmente satisfatório, por algumas das razões já mencionadas, e, de facto, duas das pessoas responsáveis pelo seu aparecimento e divulgação (Wirth e Hoare) vieram posteriormente a desenvolver outras linguagens (Modula-2 e Occam). Dado que o Modula-2 é explicitamente uma das versões de Pascal de Wirth, damos as linhas gerais desta linguagem no apêndice A.

Para os seus apoiantes e admiradores, uma das características inovadoras do Pascal é a atitude de autocrítica em relação a si próprio. J. Welsh, W. Sneideringer e C. A. R. Hoare (escrevendo para a publicação *Software — Practice and Experience*, 1977) dizem: «Se pretendemos que futuros projectos de linguagens, e mesmo futuros utilizadores, beneficiem inteiramente das vantagens introduzidas pelo Pascal, é essencial que os seus defeitos, bem como as suas virtudes, sejam cuidadosamente identificados e catalogados. A natureza minuciosa, por vezes em alto grau, das críticas aqui feitas deve considerar-se como a crença em que o Pascal é actualmente a melhor linguagem no domínio público, para fins de programação de sistemas e implementação de *software*.»

Na secção final, os três autores salientam o facto de que as vantagens de uma linguagem de alto nível poderiam ser combi-

nadas com a alta eficiência, numa «revelação que merece o título de descoberta».

Salientam que um dos resultados alcançados pela revelação do Pascal é que as linguagens são actualmente julgadas segundo padrões muito mais elevados que antigamente: o Pascal mostrou que se deve esperar mais de uma linguagem. Infelizmente, e devido às nossas expectativas crescentes, as desvantagens do Pascal tornaram-se exigentes.

Wirth escreveu um artigo intitulado «Uma contribuição da linguagem de programação Pascal» (de *IEEE Transactions on Software Engineering*, Junho de 1975), em que explica que o Pascal se destinava a ser uma linguagem que acentuasse a construção sistemática de programas, e que o Pascal era de mais fácil compreensão e mais simples para a detecção de erros por parte do compilador. Na altura em que o artigo foi escrito, o Pascal tinha cinco anos, e Wirth notou que ele já tinha demonstrado os seus méritos no tocante à facilidade de programação, fácil ajuste à verificação formal de programas, fáceis e eficientes possibilidades de implementação e portabilidade<sup>1</sup>.

Esses cinco anos também revelaram problemas. Os dois principais isolados por Wirth diziam respeito à definição de ficheiros e uso de *records* de tipo *variant* — julgo que o conceito deste tipo de registos comporta demasiados problemas, e parece que Wirth está de acordo com isto.

O Pascal ganhou ascendência na aplicação ao ensino da informática, precisamente a aplicação para que o Pascal inicialmente se destinava. Para programação *standard* de natureza pouco ousada, com aplicações *standard* em perspectiva, o Pascal tem muito a recomendar a si próprio; não terá, contudo, o sucesso que em tempos se pensou.

Um pensamento de David Banon em 1980: «Quando se pretende determinar quanto se usa determinada linguagem, é

<sup>1</sup> Nomeadamente, hoje em dia, C é muito usado para desenhar sistemas tais como compiladores, sistemas operativos, etc. (N. do T.)

<sup>1</sup> Entende-se por portabilidade de uma linguagem a sua maior ou menor capacidade de mudar de computador. Uma linguagem é tanto mais portátil quanto maior for o número de computadores que a suportam. (N. do T.)

vantajoso dispor de uma boa linguagem mas mais importante ainda é ter uma linguagem estável e altamente disponível. Podemos assim chegar à situação em que toda a gente conheça Pascal e todos o possam usar. Se queremos que o Pascal se torne uma linguagem universal, devemos negar a nós próprios a indulgência nas suas alterações (de Pascal — *A Linguagem e Sua Implementação*).

Gostávamos de saber o que ele pensa agora.

#### Apêndice A

## Modula-2: Um descendente do Pascal?

A linguagem de programação Pascal afectou várias linguagens, na sua forma e na sua natureza. Uma destas linguagens é o Modula-2.

Foi Nicklaus Wirth quem desenvolveu o Modula-2, tal como aconteceu com o Pascal, e a referência chave ao Modula-2 é o livro (manual) de Wirth intitulado *Programming in Modula-2*, 1983.

Uma linguagem chamada «Modula» emergiu dos problemas inerentes à multiprogramação, e a linguagem foi oficialmente anunciada ao Mundo em 1977 num artigo de Wirth intitulado «Modula: uma linguagem para programação modular» (*Software — Prática e Experiência*).

O Pascal foi concebido como uma linguagem de fins genéricos e, como vimos, ganhou vasta aceitação, enquanto a linguagem de programação Modula emergiu de experiências em multiprogramação, pelo que se concentrou nessa aplicação em particular. Em 1975 já se havia implementado, experimentalmente, o Modula.

Por «multiprogramação» entendem-se várias coisas, mas normalmente a interpretação refere-se a um sistema com o qual funcionam mais que um programa ao mesmo tempo, num computador que possa partilhar recursos, ou um programa de computador que execute simultaneamente mais que uma instrução<sup>1</sup>. A diferença entre os dois casos é apenas uma questão de generalidade, dado que o primeiro é apenas uma versão mais geral do segundo. No primeiro caso, o sistema é, efectivamente,

<sup>1</sup> Normalmente, numa situação de «vários programas a funcionar ao mesmo tempo», também se usa o termo de «multiprocessamento». (N. do T.)

um programa de computador que executa várias tarefas (ou seja, executa várias tarefas de utilizador simultaneamente)<sup>1</sup>.

## DE MODULA PARA MODULA-2 PARA MODULA

A linguagem Modula foi inventada em 1975, se bem que apenas tenha sido formalmente anunciada em 1977, ao iniciar-se um projecto de pesquisa para construir um sistema de computador integrado: máquina (*hardware*) e programa (*software*). Decidiu-se que o sistema (mais tarde conhecido por *Litik*) deveria ser programado numa única linguagem de alto nível (isto é, linguagem que, com uma instrução, obriga o computador a desencadear várias acções).

Isto significa, portanto, que não iria haver distinção entre a linguagem de alto nível usada na construção do sistema (por exemplo C ou BCPL) e a linguagem de baixo nível, de controlo, usada como «intermediária» entre o *hardware* do computador e eventuais dispositivos periféricos (normalmente linguagem *assembler* ou código máquina).

Já existe pelo menos uma linguagem adequada a esta tarefa, o FORTH, mas Wirth e os seus colaboradores consideraram o FORTH demasiado difícil para ser aceite pela generalidade das pessoas. Desenvolveram a nova linguagem, Modula-2, destinando-a a aplicações globais, combinando todos os aspectos do Pascal com extensões para incluir os conceitos de modularidade e multiprogramação. A sintaxe da nova linguagem estava mais perto da sintaxe do Modula do que da sintaxe do Pascal, pelo que a nova linguagem se chamou Modula 2. A estrutura desta nova linguagem ficou também mais próxima da estrutura do ALGOL 68, se bem que com muitas diferenças. Como, no entanto, se tornou mais fácil dizer simplesmente Modula — dado que o

Modula original estava ultrapassado — Wirth e outros tendem a referir-se a «Modula» quando na realidade querem dizer «Modula-2».

## ADIÇÕES AO PASCAL

A linguagem Modula contém várias adições ao Pascal, e Wirth indica os itens principais:

1. O conceito de «módulo»: um módulo consiste num conjunto de declarações e numa sequência de instruções, tudo isto iniciado com a palavra **MODULE** e terminado com **END**. Existe a facilidade de separar um módulo em duas partes: a «parte das definições» e a «parte de implementação». Aquelles termos que dependem da máquina usada podem ser guardados em módulos específicos, pelo que o uso destes termos pode ser confinado e isolado.
2. Uso de uma sintaxe mais sistemática: ajuda no processo de aprendizagem. Com particular melhoria em relação ao Pascal, todas as construções que começam com uma palavra reservada terminam com outra palavra reservada (ou seja, cada construção é devidamente «delimitada»). Como acontece com o ponto anterior, também este parece compor um pouco os erros originais do Pascal, erros esses que não estavam presentes no ALGOL 68. O uso da delimitação, e o uso de módulos com itens específicos da máquina, são particularidades *standard* no ALGOL 68, se bem que tenham nomes diferentes.
3. Conceito de «processo»: a linguagem Modula foi inicialmente desenhada para ser implementada sobre um computador convencional, de um só processador. Para uma verdadeira multiprogramação são necessários vários processadores, mas o Modula oferece algumas facilidades básicas que permitem a especificação de uma «quase multiprogramação». Também é oferecida «concorrência» para dispositivos periféricos. Põem-se em execução processos concorrentes por meio de

<sup>1</sup> Ao *software* encarregado de gerir o funcionamento simultâneo de diversos programas chama-se «*kernel*», «núcleo do multiprocessamento» ou simplesmente «*Kernel*». (N. do T.)

«co-rotinas», que são processos executados simultaneamente por um só processador. Neste sentido, multiprogramação significa a existência de vários processos a «competirem» para os recursos da máquina disponíveis (tal como acontece com muitos sistemas interactivos multi-utilizador). Esta facilidade esteve sempre disponível no ALGOL 68, de um modo menor, com o uso de «cláusulas co-laterais».

4. Facilidades de «baixo nível», que permitem quebrar as regras, algo rígidas, de consistência; são chamadas «baixo nível» dado serem particularidades próximas da verdadeira natureza do computador. Assim como acontece com quaisquer possibilidades de manipular itens da máquina (por exemplo, PEEK e POKE em BASIC), existem muitos riscos. O ALGOL 68 conteve sempre facilidades deste tipo.
5. O «tipo procedimento», permitindo a atribuição dinâmica de procedimentos a variáveis; outra vantagem já existente em ALGOL 68.

## A SINTAXE E OS MÓDULOS DE MODULA

Qualquer linguagem de programação é um conjunto infinito de frases, onde essas frases serão «bem formadas» (gramaticalmente correctas). A sintaxe da linguagem determina os critérios para determinar aquilo que se entende por «bem formado». Em Modula, as frases bem formadas denominam-se «unidades de compilação», e cada unidade é uma sequência finita de símbolos de um vocabulário finito.

O vocabulário de Modula consiste em identificadores, números, cadeias, operadores e delimitadores. Cada um desses elementos, composto por uma sequência de caracteres, recebe o nome de «símbolo léxico». Existe uma distinção em Modula entre um símbolo (digamos, uma palavra chave) e um carácter, uma sucessão dos quais pode formar um símbolo.

A inovação mais importante do Modula em comparação com o

Pascal é o módulo<sup>1</sup>. Um módulo consiste num conjunto de declarações e uma sequência de instruções. Os módulos estão compreendidos por

```
MODULE nome — do — módulo
.....
END nome — do — módulo
```

O cabeçalho do módulo contém o identificador desse módulo (que aqui é «nome — do — módulo») e possivelmente uma lista de elementos a serem IMPORTados ou EXPORTados. As «listas de importação» especificam os identificadores ou objectos declarados fora do módulo, mas nele usados (o que portanto devem ser «importados»). O segundo caso, «exportação», é uma lista que especifica os identificadores ou objectos declarados dentro do módulo e usados fora dele. Como Wirth diz, «um módulo é uma parede fechada à volta dos objectos a ele adjacentes, cuja transparência é controlada estritamente pelo programador».

Não é possível dar linhas gerais em poucas palavras das hipóteses de Modula, mas pode dizer-se que as instruções estruturadas incluem IF, CASE, REPEAT, WHILE, FOR e WITH (todas terminadas com um símbolo terminador específico; ver o ponto 2).

Para dar um exemplo de Modula, e mostrar como difere do Pascal, eis um exemplo de Wirth (*Programming in Modula-2*):

```
MODULE Power;
  FROM InOut IMPORT ReadCard, WriteString, WriteLn;
  FROM RealInOut IMPORT ReadReal, Done, WriteReal;
```

<sup>1</sup> Em Pascal 86 também existe o conceito de módulo, ao qual já fizemos referência em nota anterior. A estrutura de um módulo é rigorosamente a mesma de um programa, substituindo-se 'PROGRAM' por 'MODULE' e 'END' por 'MODEND.' Usa-se um módulo, por exemplo, quando o espaço de memória disponível para uma compilação não é suficiente por se ter um programa demasiado grande, «partindo-se» o programa em bocados e sendo cada bocado um módulo. Após as eventuais compilações, os módulos são ligados usando um linker, num processo de «linguagem» (passe o portuguêsismo). (N. do T.)

```

VAR i: CARDINAL; x, z: REAL;
BEGIN
  WriteString("x = "); ReadReal(x);
  WHILE Done DO
    WriteString("^i = "); ReadCard(i);
    z := 1.0;
    WHILE i > 0 DO
      (* z*x^i = x0^i0 *)
      z := z*x; i := i-1
    END;
    WriteReal(z, 16); WriteLn;
    WriteString("x = "); ReadReal(x)
  END;
  WriteLn
END Power

```

que deve ser comparado com os programas de potenciação do capítulo 2.

A importância do Modula reside na multiprogramação e nas técnicas cada vez mais popularizadas de empregar processores concorrentes usando «janelas». Com as suas ideias sobre listas de importação e exportação, parecem existir pontos em comum entre Modula e linguagens com base «objecto», como Smalltalk-80.

## Os diagramas sintácticos do Pascal

*Introdução.* O tradutor julga indispensável esta nota prévia. Ao longo dos diagramas sintácticos aparecerá a palavra «instrução», do inglês *statement*. No entanto, uma instrução não é obrigatoriamente aquilo que se retira do sentido estrito da palavra. O manual de Wirth, *Pascal — User Manual and Report*, refere duas possibilidades: o *statement* simples, caso em que se tem uma instrução (WRITE, READ, etc.) e o *compound statement* (instrução composta), caso em que se tem uma sequência de acções a tomar, compreendidas entre BEGIN e END.

Os modos de funcionamento do Pascal estão muito bem determinados, o que se verifica ao usar os diagramas sintácticos. Estes diagramas mostram como se formam construções legais em Pascal — isto é, mostram quais são as formas sintácticas permitidas em Pascal. Quando nos interrogamos sobre o motivo por que algumas instruções são terminadas com «;» e outras não, os diagramas sintácticos explicam-no<sup>1</sup>.

A ordem pela qual se apresentam os diagramas é importante. Neste livro, os diagramas sintácticos começam pelo de um programa, e terminam com os de inteiros sem sinal e caracteres. Esta ordem é, em certos pontos, inversa da convencional, que começa com o diagrama de um identificador e termina com o de um programa.

<sup>1</sup> Não é completamente certo. Os diagramas sintácticos confirmam uma cívica como: põhò «;» ou não?, mas não há lá NADA que explique o *porquê*. No caso de um IF...THEN...ELSE... o bloco dependente do IF não pode terminar com «;» uma vez que o «;» é o terminador Pascal de instruções pelo que o ELSE ficava sem IF, que havia sido terminado pelo «;» mal colocado. (N. do T.)

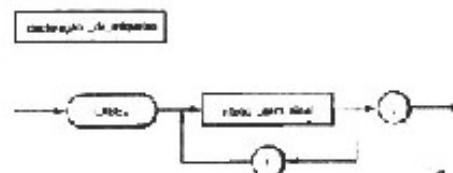
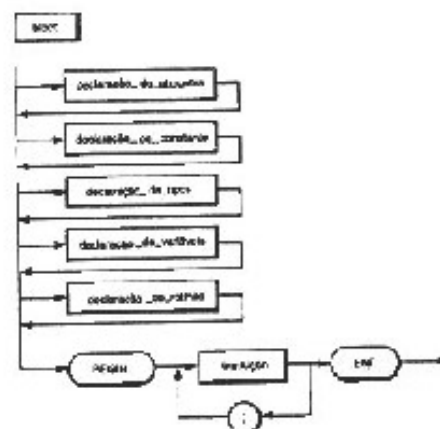
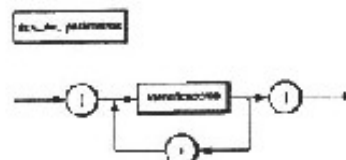
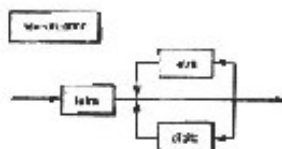


A minha ordem foi escolhida para ter uma estrutura *top down*; a maior parte dos outros conjuntos de diagramas começam ao contrário, tendo uma abordagem *bottom up* dos conceitos. A ordem de definição dos conceitos deriva da definição básica tanto quanto possível do «programa». Os meus diagramas baseiam-se no BS *standard* Pascal, mas o modo como se definem certos conceitos não é usual, e escolhi-mo-lo para ajudar à compreensão.

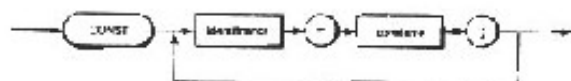
Nos diagramas, elementos dentro de círculos ou elipses alongadas devem aparecer tal como se vêem; os elementos dentro de retângulos são conceitos de Pascal, existindo normalmente outra definição para eles; as linhas mostram a sequência dos elementos na definição.

Por exemplo, na definição de «programa» a sequência é uma «linha recta» de um elemento para outro, enquanto na definição de «identificador» a sequência é: primeiro, uma letra, e depois (à escolha) uma letra ou um dígito, repetindo-se esta opção, se for caso disso.

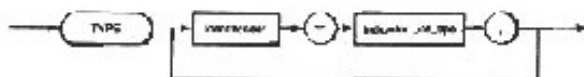
Não damos aqui o significado de um elemento: a preocupação é a sintaxe e não a semântica destes diagramas.



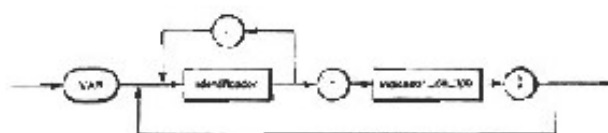
identificacao\_de\_versao



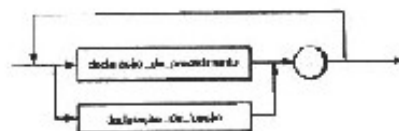
conexao\_de\_tipo



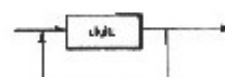
dados\_cole\_de\_criacao



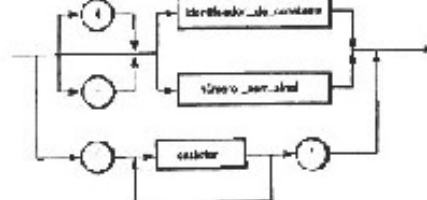
dados\_cole\_de\_criacao



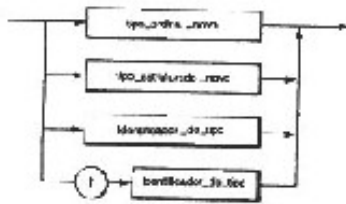
dados\_cole\_de\_criacao



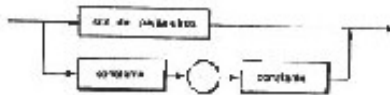
dados\_cole\_de\_criacao



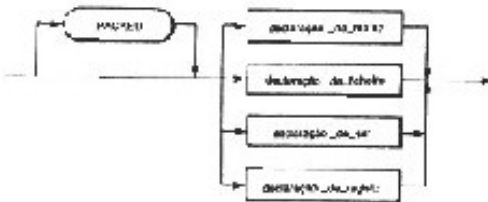
relatorio\_de\_tpa



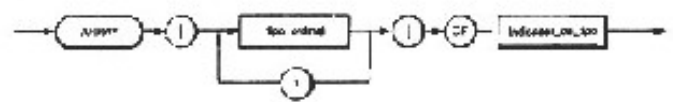
tpa\_ordem\_novo



tpa\_ordem\_novo: tpa



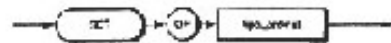
descontrole\_de\_tpa



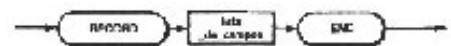
descontrole\_de\_tpa: tpa

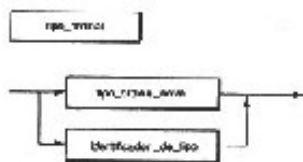


descontrole\_de\_tpa

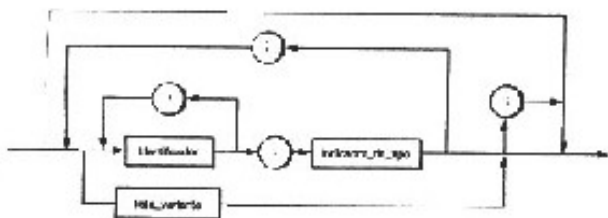


recuperação\_de\_tpa

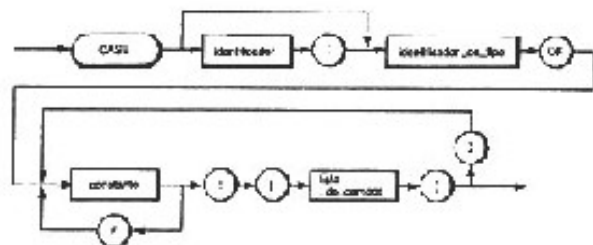




**lista\_de\_campos**



**lista\_variavel**



**definicao\_do\_procedimento**



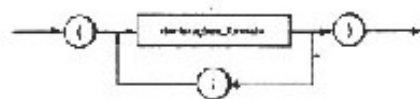
**integracao\_procedimento**

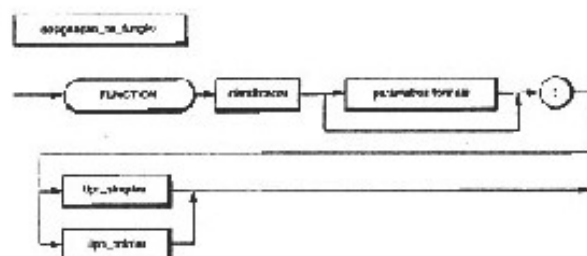
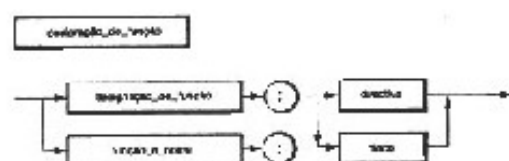
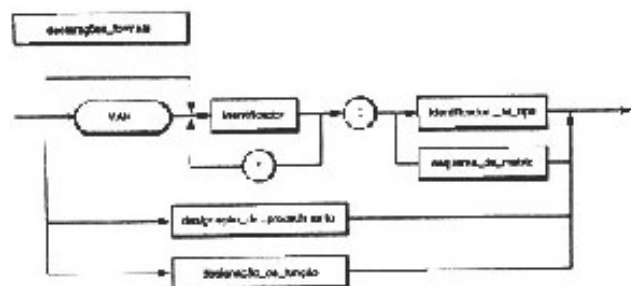


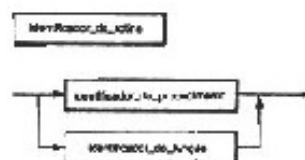
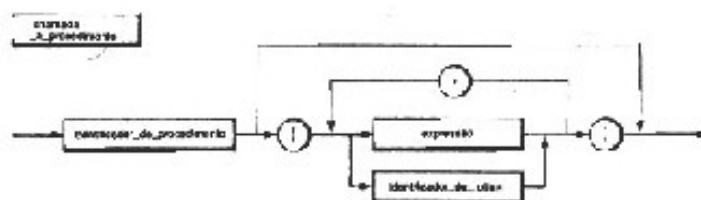
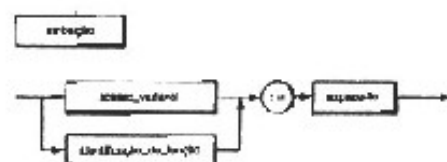
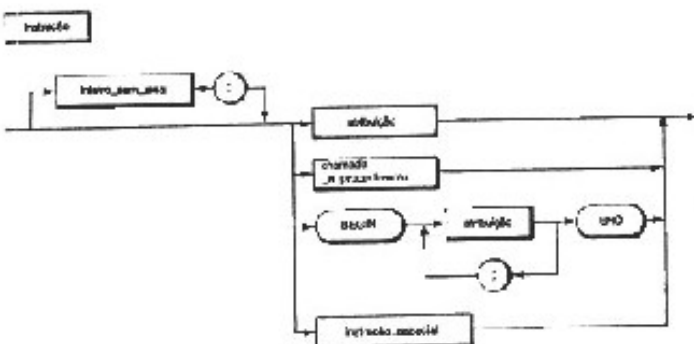
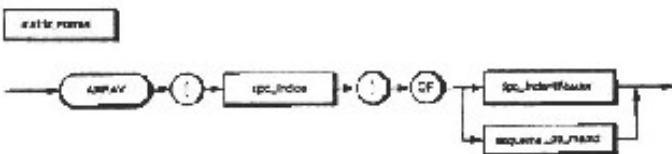
**global**



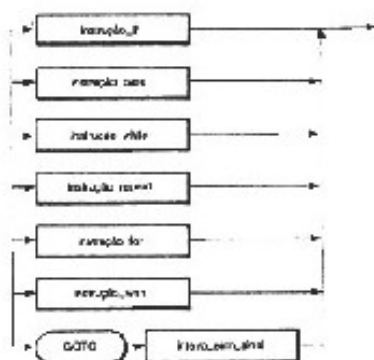
**parametros**







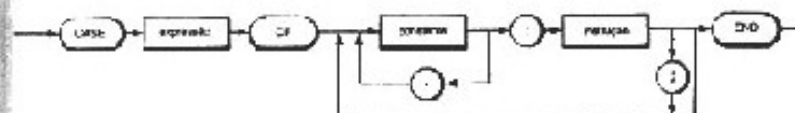
# instrução: repetir



# instrução: if



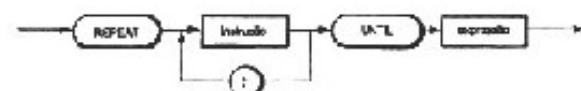
# instrução: case



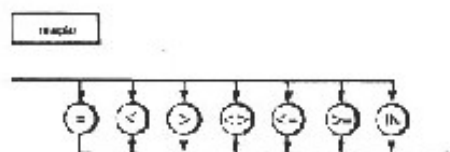
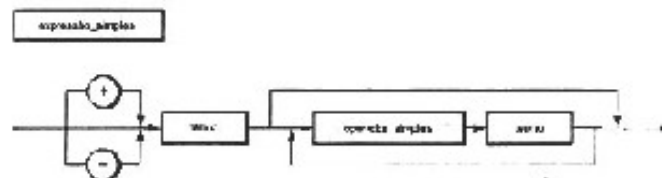
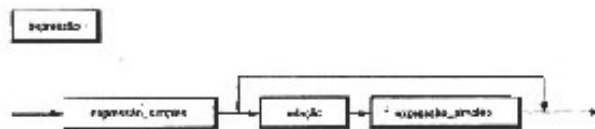
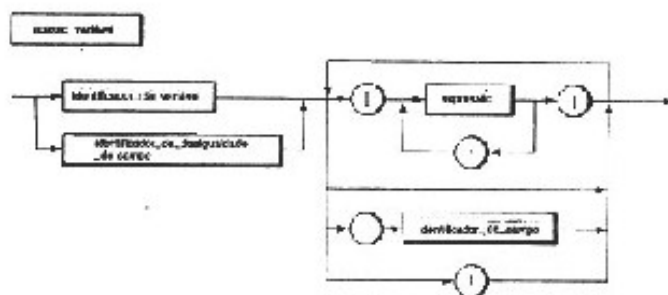
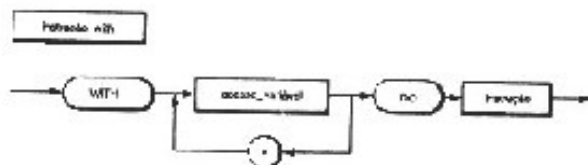
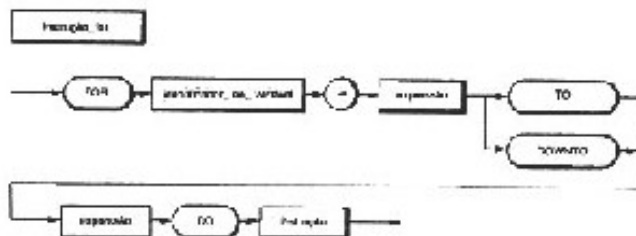
# instrução: while



# instrução: repeat

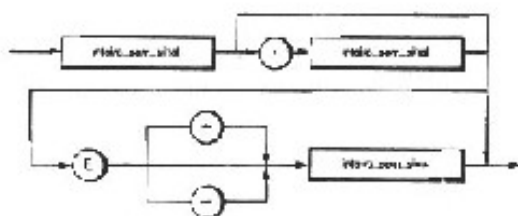




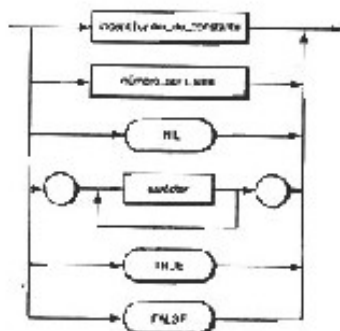




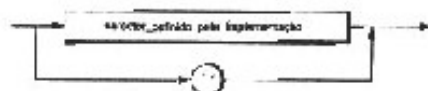
16449\_001\_001



16449\_001\_001



16449\_001\_001



## Descrição da sintaxe em AF

O Formalismo Allan (AF) foi idealizado para fornecer um modo de descrição da sintaxe no meio da escrita, em vez de diagramas sintáticos. O Formalismo Allan tem muitas semelhanças com o BNF (Backus-Naur Form), com algumas ligeiras extensões no uso de delimitadores (como sejam os parênteses).

Esses delimitadores, em AF, são:

[ ] O elemento assim envolvido é um conceito da linguagem, e tem estabelecida uma definição (ou o seu significado é imediatamente óbvio). Em diagramas sintáticos, o seu equivalente é um retângulo.

< > O elemento assim envolvido é uma palavra ou um símbolo, aparecendo no programa tal como for mostrado. Em diagramas sintáticos, o seu equivalente é um círculo ou uma elipse alongada.

{ } O elemento ou elementos assim envolvidos podem ocorrer zero ou mais vezes. Em diagramas sintáticos o seu equivalente é um ciclo (para trás) facultativo.

[\*\*] O elemento ou elementos que apareçam assim delimitados podem ocorrer zero ou uma vez. Em diagramas sintáticos, o seu equivalente é um caminho (para a frente) facultativo.

() Quando delimitados assim, os elementos são tratados como uma entidade do programa.

Existem ainda dois operadores binários:

/ Indica a possibilidade de uma escolha entre elementos de ambos os lados da barra.

—> Indica que o elemento à esquerda da seta aponta para a definição que está à direita da seta.

[programa] —>  
 <PROGRAM> [identificador] [lista-de-parâmetros] <;>  
 [bloco] <...>  
 [identificador] —>  
 [letra] { [letra] / [dígito] }  
 [lista-de-parâmetros] —>  
 <(> [identificador] { <,> [identificador] } <)>  
 [bloco]  
 { [declaração-de-etiquetas] { [declaração-de-constantes] }  
 { [declaração-de-tipos] } { [declaração-de-variáveis] }  
 { [declaração-de-procedimentos] }  
 <BEGIN> [instrução] { <;> [instrução] } <END>  
 [declaração-de-etiquetas] —>  
 <LABEL> [inteiro-sem-sinal] { <,> [inteiro-sem-sinal] }  
 <;>  
 [declaração-de-constantes] —>  
 <CONST> [identificador] <=> [constante] <;>  
 { [identificador] <=> [constante] <;> }  
 [declaração-de-tipos] —>  
 <TYPE> [identificador] <=> [indicador-de-tipo] <;>  
 [identificador] <=> [indicador-de-tipo] <;>  
 [declaração-de-variáveis] —>  
 <VAR> [identificador] { <,> [identificador] } <;>  
 [indicador-de-tipo] <;>  
 { [identificador] } <,> [identificador] { <,> [indicador-de-tipo] }  
 <;>  
 [declaração-de-procedimentos] —>  
 { [declaração-de-procedimento] / [declaração-de-função] }  
 <;>  
 { { [declaração-de-procedimento] / [declaração-de-função] } <;> }

[inteiro-sem-sinal] —>  
 - [dígito] { [dígito] }

[constante] —>  
 { { \* <+> / <-> \* } ([identificador-de-constante] /  
 [número-sem-sinal]) /  
 (<'> [carácter] { [carácter] } <'>) }

[indicador-de-tipo] —>  
 (tipo-ordinal-novo / [tipo-estruturado-novo] /  
 [identificador-de-tipo]  
 / (<'> [identificador-de-tipo] )

[tipo-ordinal-novo] —>  
 [lista-de-parâmetros] / ([constante] <...> [constante])

[tipo-estruturado-novo] —>  
 { \* <PACKED> \* } ([declaração-de-matriz] /  
 [declaração-de-ficheiro] /  
 [declaração-de-set] / [declaração-de-registo] )

[declaração-de-matriz] —>  
 <ARRAY> <[> [tipo-ordinal] { <,> [tipo-ordinal] }  
 <]>  
 <OF> [tipo-ordinário]

[declaração-de-ficheiro] —>  
 <FILE> <OF> [indicador-de-tipo]

[declaração-de-set] —>  
 <SET> <OF> [tipo-ordinal]

[declaração-de-registo] —>  
 <RECORD> [lista-de-campos] <END>

[tipo-ordinal] —>  
 [tipo-ordinal-novo] / [identificador-de-tipo]

[lista-de-campos] —>  
 { \*([indicador] {<,> [indicador]} <,> [indicador-de-tipo]  
 { <,> [identificador] {<,> [identificador]} <,>  
 [indicador-de-tipo]}) /  
 [lista-de-variantes] { \* <,> \* } \* }

[lista-de-variantes] —>  
 [CASE] { \* [identificador] <,> \* } [identificador-de-tipo]  
 <OF>  
 [constante] { <,> [constante] } <,> <(> [lista-de-campos]  
 <)>  
 { <,> [constante] { <,> [constante] } <,> <(> [lista-de-  
 campos] <)> }

[declaração-de-procedimento] —>  
 ([designação-de-procedimento] <,> ([directiva] / [bloco]) /  
 ([procedimento-e-nome] <,> [bloco])

[designação-de-procedimento] —>  
 <PROCEDURE> [identificador] { \* [parâmetros-formais]  
 \* }

[directiva]  
 <FORWARD> [instrução-dependente-da-  
 implementação]

[parâmetros-formais] —>  
 <(> [declarações-formais] { <,> [declarações-formais]}  
 <)>

[procedimento-e-nome] —>  
 <PROCEDURE> [nome-de-procedimento-já-declarado]

[declarações-formais] —>  
 { { \* <VAR> \* } [identificador] { <,> [identificador] } <,>  
 ([identificador-de-tipo] / [esquema-de-matriz]) /  
 [designação-de-procedimento] / [designação-de-rotina]

[declaração-de-função] —>  
 ([designação-de-função] <,> ([directiva] / [bloco]) /  
 ([função-e-nome] <,> [bloco])

[designação-de-função] —>  
 <FUNCTION> [identificador] { \* [parâmetros-formais] \* }  
 <(>  
 ([tipo-simples] / [tipo-pointer])

[esquema-de-matriz] —>  
 [matriz-compacta] / [matriz-normal]  
 [matriz-compacta] —>  
 <PACKED> <ARRAY> <(> [tipo-índice] <(> <OF>  
 [identificador-de-tipo]

[matriz-normal] —>  
 <ARRAY> <(> [tipo-índice] <(> <OF>  
 [identificador-de-tipo] / [esquema-de-matriz])

[tipo-índice] —>  
 [identificador] <...> [identificador] <,> [tipo-ordinal]

[instrução] —>  
 { \* [Início-sem-sinal] <,> \* }  
 [atribuição] / [chamada-a-procedimento] / [instrução-  
 especial] /  
 <BEGIN> [instrução] { <,> [instrução] } <END>

[instrução] —>  
 ([acesso-variável] / [identificador-de-função]  
 <=> [expressão])

[chamada-a-procedimento] —>  
 [identificador-de-procedimento]  
 { \* <(> ([expressão] / [identificação-de-rotina])  
 { <,> ([expressão] / [identificador-de-rotina]) } <,> \* }  
 [identificador-de-rotina] —>

[identificador-de-procedimento] / [identificador-de-função]

[instrução-especial] —>

[instrução-if] / [instrução-case] / [instrução-while]  
[instrução-repeat] / [instrução-for] / [instrução-with] /  
( <GOTO> [inteiro-sem-sinal] )

[instrução-if] —>

<IF> [expressão] <THEN> [instrução] { \* IELSE>  
[instrução] \* }

[instrução-case] —>

<CASE> [expressão] <OF>  
[constante] { <,> [constante] } <:> [instrução]  
{ <:> [constante] } <,> [constante] { <:> [instrução] }  
{ \* <:> \* } <END>

[instrução-while] —>

<WHILE> [expressão] <DO> [instrução]

[instrução-repeat] —>

<REPEAT> [instrução] { <:> [instrução] } <UNTIL>  
[expressão]

[instrução-for] —>

<FOR> [identificador-de-variável] <:=> [expressão]  
( <TO> / <DOWNTO> ) [expressão] <DO> [instrução]

[instrução-with]

<WITH> [acesso-de-variável] { <,> [acesso-de-variável] }  
<DO> [instrução]

[acesso-variável] —>

( [identificador-variável] / [identificador-de-designador-de-campo] )

{ \* (<'> / (<.> [identificador-de-campo])  
( <[> [expressão] { <,> [expressão] } <]> ) )  
{ <'> / (<.> [identificador-de-campo]) /  
( <[> [expressão] { <,> [expressão] } <]> { \* }

[expressão] —>

[expressão-simples] \* [factor] [expressão-simples]\*

[expressão-simples] —>

{ \* <+> / <-> \* } [termo] \* [operador-simples] [termo]  
{ [operador-simples] [termo] } \* }

[relação] —>

<=> / <<> / <>> / <<>> / <<=> / <>=> / <IN>

[termo] —>

[factor] { \* [operador-complexo] [factor]  
{ [operador-complexo] [factor] } \* }

[operador-simples] R>

<+> / <-> / <OR>

[factor] —>

[constante-sem-sinal] / [acesso-variável] /  
[identificador-de-limite] /  
[identificador-de-função]  
{ \* <(> ([expressão] / [identificador-de-rotina])  
{ <,> ([expressão] / [identificador-de-rotina]) } <)> ) /  
( <(> [expressão] <)> ) / ( <NOT> [factor] ) /  
( <[> { \* [expressão] { \* <.> [expressão] \* }  
{ <,> [expressão] { \* <.> [expressão] \* } } \* } <]> )

[operador-complexo] —>

<\*> / </> / <DIV> / <MOD> / <AND>

[número-sem-sinal] —>

[inteiro-sem-sinal] { \* <.> [inteiro-sem-sinal] \* }

$[* <E> \{ * <+> / <->^* \} \{ \text{inteiro} - \text{sem} - \text{ sinal} \}^* ]$   
 $[ \text{constante} - \text{sem} - \text{ sinal} ] \rightarrow$   
 $[ \text{identificador} - \text{de} - \text{constante} ] / [ \text{número} - \text{sem} - \text{ sinal} ] /$   
 $<NIL> /$   
 $< > [ \text{carácter} ] \{ [ \text{carácter} ] \} <^* >$   
 $[ \text{carácter} > \rightarrow$   
 $[ \text{carácter} - \text{definido} - \text{pela} - \text{implementação} ] / <^* >$

## Pascal-P e a Máquina Código-P

Nas linhas gerais originalmente traçadas por Nicklaus Wirth, quanto à forma de um tradutor de Pascal (um compilador), o programa original para traduzir código fonte em Pascal para código objecto estava escrito em Pascal. Isto não é tão impossível quanto pode parecer.

### PASCAL A PARTIR DE PASCAL

O tradutor original estava escrito numa versão muito simples de Pascal (digamos Pascal.0) e empregou-se esta versão para idealizar um tradutor para a versão completa de Pascal (digamos Pascal.1). Não existe, contudo, um tradutor para o Pascal.0 foi escrito em código máquina do próprio computador, mas como o Pascal.0 era uma forma de linguagem muito simples, o código máquina era muito mais fácil de escrever do que sucedia com a versão mais completa (ou seja, o Pascal.1). Esta técnica, também conhecida por *boot-strapping*, também tinha sido usada com a linguagem BCPL (a partir da qual se desenvolveu a linguagem C).

Muitas das complexidades da programação foram acunodadas ao nível da tradução para Pascal.0 de Pascal.1 e não estavam presentes na tradução em código máquina do Pascal.0. Outra grande vantagem deste sistema era o facto de que, para implementar um sistema Pascal patente num novo tipo de computador, bastava escrever um tradutor para Pascal.0 no código máquina do novo computador.

Este facto salientou grandemente a portabilidade da nova linguagem Pascal, dado que o trabalho necessário para transferir



Pascal para uma nova máquina ficava bastante simplificado.

No início, Wirth considerou a hipótese de escrever o tradutor do Pascal.1 noutra linguagem de alto nível, tal como o FORTRAN IV, que foi escolhido principalmente devido a ser a linguagem compilada mais disponível em computadores, e a ideia era que um tradutor de Pascal escrito em FORTRAN seria facilmente implementável na maior parte dos computadores.

Viu-se, contudo, que as implementações de FORTRAN tendiam a diferir muito entre si, apesar de o FORTRAN ser considerado supostamente uma linguagem *standard*, pelo que o tradutor não seria tão portátil como se esperava. Descobriu-se então que o tradutor de Pascal era difícil de escrever, uma vez que os tipos de dados e as disposições em FORTRAN eram bastante pobres. O Pascal tinha-os extremamente potentes, portanto porque não usar Pascal?

Uma vez tomada a decisão de traduzir pelo processo código máquina  $\rightarrow$  Pascal.0  $\rightarrow$  Pascal.1, o processo ficou, de um modo geral, simplificado, e em alguns casos chegaram a surgir níveis mais elevados de Pascal.

## COMPILADORES E INTERPRETADORES

Usámos, ao longo do livro, o termo «tradutor», dado ser um termo mais geral, que engloba «compiladores» e «interpretadores». Um outro motivo para usar este termo genérico assenta no facto de muitos dos «compiladores» de Pascal para microcomputador serem mais bem descritos como «interpretadores de compiladores».

Um compilador traduz o programa completo em código fonte para código objecto antes de o programa ser executado; um interpretador traduz cada linha do programa para código objecto, esquecendo esse código quando o comando passa para nova linha do programa.

Em teoria, o Pascal é uma linguagem compilada (mais rápida) e o BASIC é uma linguagem interpretada (pelo que é mais lenta). Existem compiladores de BASIC, e o Pascal nem sempre é tão

compilado como parece. A escrita com o tradutor é mais simples e há maior economia em termos de armazenagem se for necessário manter em memória o código fonte.

Torna-se muito difícil encontrar um interpretador «puro», dado ser sempre necessário armazenar informação que não deve ser esquecida. Uma linguagem sem variáveis é muito restritiva, mas, se existem variáveis, esta informação deve ser guardada em posições de memória. Existirão ponteiros para o código objecto apropriado.

De modo análogo, é difícil encontrar um compilador «puro», ou seja, uma tradução total do programa em código fonte para um programa em código máquina, sem modificações na execução. Os compiladores «mais puros» serão talvez os compiladores optimizadores de FORTRAN IV.

Uma linguagem que permita a criação de estruturas de dados sem fim e/ou ofereça facilidades de processamento de listas, por exemplo, o uso de ponteiros (*pointers*) em Pascal, necessita de variar a armazenagem consoante o estado do programa e das suas entradas. Então, na sua operação, um compilador de Pascal necessita de aspectos de um interpretador.

## O SISTEMA PASCAL-P

A tentativa de estabelecer se o tradutor de Pascal é um compilador ou um interpretador é complicada por um subconjunto do Pascal *standard* conhecido como «Pascal-P».

O compilador de Pascal-P é portátil para o subconjunto, que não está escrito em Pascal.0, mas acré um programa em «código objecto» para um computador hipotético. Este computador hipotético (a «máquina código-P» ou «máquina-P») é um computador funcionando à base de uma pilha (o Pascal, por exemplo, usa uma pilha para ter conhecimento das chamadas a rotinas).

Em vez de compilar Pascal.0 para código máquina, o Código-P é traduzido para o código máquina do computador onde o programa vai correr: o código-P substitui o Pascal.0. Existem

muitas maneiras de o realizar, mas os pormenores são demasiado complicados para valer a pena estudá-los.

O Pascal-P não contém as seguintes vantagens, presentes em Pascal *standard*:

- A. Procedimentos/funções como parâmetros
- B. Instruções GOTO que provoquem a saída do corpo de procedimentos ou funções
- C. Todos os tipos de ficheiros que não sejam: ficheiros de caracteres pré-definidos (do tipo «texto»)
- D. Todas as características relacionadas com «compactação»
- E. O procedimento *standard* DISPOSE, substituído em Pascal-P por MARK e RELEASE.

O exemplo mais importante de um sistema código-P é o Pascal UCSD, e deste o exemplo mais importante é o Pascal Apple. É interessante ver como as restrições em Pascal Apple correspondem a estas alterações.

As restrições A, B e E ainda se aplicam em Pascal Apple, mas em relação à restrição C existem bastantes alterações — algumas mais restritivas que em relação à versão *standard*, outras mais acomodativas. Existem em Pascal Apple possibilidades de compactação (restrição D), se bem que não seja esse o caso de procedimentos *standard* como PACK e UNPAC.

A razão por que o Pascal Apple é o exemplo mais importante do Pascal-P não é o Pascal-P em si, mas a máquina Apple-P. Não só o Pascal Apple foi implementado usando o sistema código-P, como este sistema foi ainda usado para implementar outras linguagens como o FORTRAN 77. No *Apple II*, a «carta da linguagem» contém, efectivamente, a máquina-P. («Carta» é uma placa de circuito impresso com uma extremidade sob a forma de ficha, encaixável posteriormente numa máquina.)

No *Apple*, o sistema funciona tomando o programa fonte em Pascal, compilando-o para código-P (o manual de referência ao Pascal Apple — *Apple Pascal Reference* — contém um bom estudo da forma do código-P). Após esta «compilação», o programa Pascal foi transformado num programa em código-P.

Na altura da execução, este programa é interpretado por um tradutor para código máquina linha a linha. Este é o motivo que faz o Pascal Apple ser muitas vezes tão rápido ou mais lento que o BASIC do *Apple*. A sequência é: *O Pascal é compilado para código-P e interpretado para código-máquina.*

Foi por isto que dissemos que, muitas vezes, as versões de Pascal de microcomputador são interpretadores de compiladores.

O facto de a máquina-P ter sido usada com tanto sucesso pela Apple para implementar linguagens diferentes do Pascal dá mais crédito ao desenho original do Pascal.

## Símbolos especiais do Pascal

Os símbolos especiais, em Pascal, usam em duas categorias principais:

[símbolo-especial] —>  
[palavra-reservada] / [símbolo]

onde se tem que

[palavra-reservada] —>

<AND> / <ARRAY> / <BEGIN> / <CASE> /  
<CONST> / <DIV> / <DO> / <DOWNTO> /  
<ELSE> / <END> / <FILE> / <FOR> /  
<FUNCTION> / <GOTO> / <IF> / <IN> /  
<LABEL> / <MOD> / <NIL> / <NOT> /  
<OF> / <OR> / <PACKED> / <PROCEDURE> /  
<PROGRAM> / <RECORD> / <REPEAT> /  
<SET> / <THEN> / <TO> / <TYPE> /  
<UNTIL> / <VAR> / <WHILE> / <WITH> /

e

[símbolo] —>

<+> / <-> / <\*> / </> / <=> / <<>> /  
<>> / <|> / <|> / <|> / <|> / <|> /  
<|> / <|> / <|> / <|> / <|> / <|> /  
<|> / <|> / <|> / <|> / <|> / <|> /

Dentro do conjunto [símbolo] podem existir confusões. Por exemplo, <<>> representa o símbolo de programa <>, que significa «diferente de».

Os símbolos especiais são «marcas» que têm significados especiais, servindo para delimitar as unidades sintáticas do Pascal. São «reservas» no Pascal, e não podem por isso servir para fins diferentes daqueles a que se destinam. As palavras reservadas não são procedimentos ou funções, e não «fazem» nada nesse aspecto.

Existem outras palavras «reservadas» em Pascal, podendo ser inseridas numa categoria [identificador-standard].

[identificador-standard] —>

<BOOLEAN> / <CHAR> / <FALSE> /  
<INTEGER> / <REAL> / <STRING> /  
<TRUE>

havendo ainda outra categoria, a [directiva]

[directiva] —>

IFORWARD> / [directiva-da-implementação]

o que significa que os sistemas Pascal deverão possuir (para serem *standard*) a directiva FORWARD, podendo existir outras directivas, que dependerão da implementação.

No Pascal UCSD do Apple existem outros identificadores «reservados»:

[identificador-adicional-de-Pascal-Apple] —>

<EXTERNAL> / <SEGMENT> / <SEPARATE> /  
<UNIT> / <USES> / <INTERFACE> /  
<IMPLEMENTATION>

e, para pormenores específicos sobre estes identificadores, consulte-se a documentação do Apple.

[símbolo-reservado-standard] —>

[palavra-reservada] / [símbolo] /  
[identificador-standard] / [directiva]

e a modificação para Pascal UCSD da Apple, por exemplo, é

```
[símbolo — reservado — do — Pascal — Apple] —>  
[símbolo — reservado — standard] ;  
[identificador — adicional — do — Pascal — Apple]
```

Os símbolos reservados não devem ser confundidos com procedimentos e funções *standard*, sobre os quais se deve consultar o apêndice F.

## Apêndice F

# Procedimentos e funções «standard»

## PROCEDIMENTOS PARA MANIPULAÇÃO DE FICHEIROS

### PUT(F)

Existe, inerente a qualquer ficheiro F, uma variável conhecida como janela<sup>1</sup> e representada por F\*. Esta variável está a apontar à posição corrente do ficheiro: ao fazer uma chamada a PUT, fazendo PUT(F), o valor corrente de F\* é colocado no ficheiro, e o ponteiro passa para a posição seguinte (ver também WRITE). Isto será verdadeiro apenas se a função EOF(F) tiver um valor verdadeiro (TRUE).

### GET(F)

Avança a janela F\* para o próximo componente do ficheiro, e atribui o seu conteúdo a F\*. Se esse conteúdo não existir (fim de ficheiro) então EOF(F) é posto a TRUE (valor lógico 1). Ver também READ.

### RESET(F)

Coloca-se a janela na primeira posição do ficheiro (colocando EOF(F) a TRUE se o ficheiro está vazio, e a FALSE se tiver algum conteúdo), para efeitos de leitura.

### REWRITE(F)

Torna o ficheiro F um ficheiro vazio, para efeitos de escrita.

<sup>1</sup> Para concretizar o que é uma janela, imagine-se uma lente ampliadora fixando um filme. De cada vez só se vê um fotograma. O filme é o ficheiro, e a lente é a janela de visualização. (N. do T.)

### **PAGE(F)**

Instrui o dispositivo de saída para passar para o início de nova página antes de imprimir a próxima linha do ficheiro de texto F.

### **READ(F,CH)**

Lê o componente seguinte do ficheiro F, colocando-o em CH. Omitindo F, é assumido o ficheiro *standard INPUT* e, se houver uma sucessão de identificadores a seguir a CH, existirá uma tentativa de ler essa quantidade de valores. Se CH for uma variável de tipo CHAR, então READ(F,CH) é o mesmo que GET(F); CH:=F;

### **READLN(F,CH)**

Tem um efeito similar a READ(F,CH), com a diferença de que, para um ficheiro de texto, existe o movimento para a linha seguinte.

### **WRITE(F,CH)**

Escreve o valor corrente de CH no ficheiro F. Omitindo F é assumido o ficheiro *standard OUTPUT*. WRITE(F,CH) é o mesmo que F:=CH; PUT(F);

### **WRITEIN(F,CH)**

Se F for um ficheiro de texto, provoca mudança de linha.

## **PROCEDIMENTOS DE MANIPULAÇÃO DE DADOS DINÂMICOS**

### **NEW(P)**

Ajusta o ponteiro P de modo a indicar a posição seguinte de memória disponível na pilha («heap»).

### **DISPOSE(P)**

Indica que o espaço de armazenagem ocupado pela variável P\* já não é necessário (não é assim em Pascal-P/Pascal UCSD, onde

tem de haver uma aproximação, com o uso dos procedimentos MARK e RELEASE).

### **MARK(I)**

(Apenas Pascal-P.) Marca o ponteiro presente como topo do limite da pilha, e guarda o valor no inteiro I.

### **RELEASE(I)**

(Apenas Pascal-P.) Repõe no ponteiro o topo da pilha indicado pelo inteiro I.

## **PROCEDIMENTOS DE TRANSFERÊNCIA DE DADOS (NÃO EM PASCAL-P)**

Suponhamos que há

```
VAR A : ARRAY[M..N] OF T;  
Z : PACKED ARRAY[U..V] OF T
```

onde um *packed array* é uma matriz economicamente armazenada, com novos problemas no acesso a componentes individuais nossos.

Como ajuda neste uso económico de espaço de armazenagem, mas tendo um acesso razoável, é possível expandir o *packed array* para dentro de um *array* normal (com um nome diferente) e, quando terminarem os cálculos e modificações, compactar o *array* novamente.

### **PACK(A,I,Z)**

efectua a sequência

```
FOR J := U TO V DO
```

```
  Z(J) := A[J-U+1];
```

### **UNPACK(Z,A,I)**

efectua a sequência

```
FOR J := U TO V DO
```

```
  A[J-U+1] := Z(J);
```

## FUNÇÕES ARITMÉTICAS

### **ABS(X)**

Calcular o valor absoluto de X. O tipo do resultado é o mesmo do X, que deve ser inteiro ou real.

### **SQR(X)**

Calcula o quadrado de X ( $X^2$ ). O tipo do resultado é o mesmo do X, que deve ser inteiro ou real.

Para as funções seguintes, o tipo do argumento X pode ser inteiro ou real, mas o tipo do resultado é sempre real.

### **SIN(X)**

Senó de X (normalmente, em radianos)

### **COS(X)**

Co-senó de X (normalmente, em radianos)

### **ARCTAN(X)**

Arco-tangente (inverso de tangente)

### **EXP(X)**

e (base dos logaritmos naturais) elevado a X

### **LN(X)**

Logaritmo natural de X

### **SQRT(X)**

Raiz quadrada de X

## PREDICADOS (FUNÇÕES BOOLEANAS)

### **ODD(X)**

X deve ser de tipo inteiro: o resultado é TRUE se X for ímpar e FALSE se for par.

### **EOLN(F)**

Devolve o valor TRUE quando, ao ler o ficheiro de texto F, se atingir o fim da linha corrente; noutra situação devolve o valor FALSE.

### **EOF(F)**

Devolve o valor TRUE quando ao ler o ficheiro F, se atingir o fim do ficheiro (End Of File); noutra situação, devolve o valor FALSE.

## FUNÇÕES DE TRANSFERÊNCIA

### **TRUNC(X)**

X deve ser do tipo real; o resultado é o maior inteiro menor ou igual a X se X maior ou igual a zero, ou o menor inteiro maior ou igual a X se X menor ou igual a zero.

### **ROUND(X)**

X deve ser do tipo real; o resultado do tipo é inteiro, é o valor arredondado de X.

### **ORD(X)**

Devolve o ordinal do parâmetro X no conjunto de valores dado pelo tipo de X.

### **CHR(X)**

X deve ser de tipo inteiro, e o resultado é o carácter cujo ordinal é X. (Em ASCII, por exemplo,  $\text{CHR}(66) = 'B'$ , — *N. do T.*)

## OUTRAS FUNÇÕES «STANDARD»

### **SUCC(X)**

X pode ser de um qualquer tipo escalar (à excepção de REAL), e o resultado é o valor seguinte de X, se existir.

### **PRED(X)**

X pode ser de um qualquer tipo escalar (à excepção de REAL), e o resultado é o valor anterior a X, se existir.

## Bibliografia

Os livros seguintes destinam-se a auxiliar aqueles que gostam dos aspectos técnicos a obter leitura de base sobre a teoria de Pascal. O melhor modo de aprender a programar bem é pensar naquilo que se está a fazer.

Estes livros ajudam a pensar e a entender o Pascal. Todos eles requerem algumas bases técnicas.

Barran, P.W. (ed). *Pascal: The Language and its Implementation* (Wiley, 1981). Uma excelente série de documentos sobre diversos aspectos do Pascal. Vale a pena ler.

Boules, K. *Problem Solving Using Pascal* (Springer-Verlag, 1977). A melhor introdução ao Pascal UCSD (foi ele quem o inventou). Tem uma estrutura mais *top down* que a maioria.

Jensen, K. e Wirth N.: *Pascal: User Manual, and Report* (2.ª ed. Springer-Verlag, 1974, 1978). A declaração do desenhista; de leitura essencial.

Rohr, J.S.: *Writing Pascal Programs* (Cambridge UP, 1983). Mostra como são construídos programas em Pascal, fingindo do domínio das ideias puras e mostrando algumas aplicações.

## Biblioteca Verbo de Informática

## 1.º volume

**Jogos Dinâmicos para o ZX Spectrum**  
de Tim Hartnell

Simultaneamente com programas de diversão e de criatividade, esta obra oferece a possibilidade de um perfeito conhecimento e do uso consciente do computador. Inclui jogos de «arcada» e de tabuleiro e programas de aventuras e sugestões para melhorar e desenvolver a programação.

## 2.º volume

**Aprofundar o Basic do Spectrum**  
de Mike Lord

Simultaneamente obra de carácter didáctico muito sério e meio de diversão para o amador de informática, esta obra interessa a um largo leque de utilizadores do *Spectrum*, desde principiantes a programadores experientes. É fonte de referência imprescindível quanto ao *software* desta máquina.



### 3.º volume

#### **O Domínio do Código Máquina do Spectrum** de *Toni Baker*

Este livro satisfaz a ambição — e a necessidade — de todos os programadores que utilizam o *Spectrum*: dominar directamente a linguagem que o coração da máquina conhece, o que representa um salto qualitativo quanto a versatilidade de programação, velocidade de execução e domínio do grialismo.

### 4.º volume

#### **As Melhores Rotinas para o ZX Spectrum** de *Jahni Hardinen e Andrew Hewson*

40 rotinas em código máquina para o *Spectrum*. Obra destinada a quem quer tirar o melhor partido do seu minicomputador: utilizando a linguagem máquina do *Spectrum*, multiplicam-se espantosamente as suas capacidades.

Além disso, trata-se de um instrumento de trabalho de inextinguível utilidade.

### 5.º volume

#### **Os 20 Melhores Programas para o Spectrum** de *Andrew Hewson*

Estes programas, além da sua utilidade prática, representam uma perfeita fonte de referência das técnicas de programação mais difundidas e ensinam a arte de programar através de exemplos perfeitos.

### 6.º volume

#### **Guia Avançado para o Spectrum** de *Mike James*

Introdução prática às características mais avançadas do *Spectrum*, tanto no *hardware* como no *software*.

Oferece ao leitor a exploração das possibilidades mais sofisticadas deste microcomputador.

### 7.º volume

#### **57 Rotinas em Basic para o Spectrum**

Autêntico instrumento de trabalho para quem se dedica a criar os seus próprios programas e ambiciona libertar-se da resolução dos múltiplos problemas práticos rotineiros que tornam fastidiosa aquela fascinante actividade.

### 8.º volume

#### **Astronomia no ZX Spectrum**

Obra dinâmica e cheia de interesse, que introduz o leitor com simplicidade nos vários campos da astronomia mas se dirige especialmente a quem possui o *Spectrum* e deseja expandir os seus conhecimentos informáticos a outros campos.

Próximo volume:

### **O Spectrum por dentro**

O interesse pelo *Spectrum* em si, como máquina – isto é, o *hardware* –, impôs-se naturalmente a inúmeros entusiastas da microinformática, desejosos, por curiosidade e por necessidade, de conhecerem efectivamente o funcionamento e o *design* do mais popular dos computadores.

Esta obra permite aos iniciados curiosos manusear projectos até agora «secreto», com perfeita compreensão do que acontece dentro do computador.