SEU PRIMEIRO PROGRAMA EM BASIC

Rodnay Zaks

SEU PRIMEIRO PROGRAMA EM BASIC

Rodnay Zaks

AO LIVRO TÉCNICO S/A
Indústria e Comércio

CIP-Brasil. Catalogação-na-fonte
Sindicato Nacional dos Editores de Livros, RJ.

Z25s

Zaks, Rodnay.
Seu primeiro programa em BASIC / Rodnay Zaks ; tradução de Eduardo Luiz Peixoto Fortuna. – Rio de Janeiro : Ao Livro Técnico, 1985.

(Série Computação)

Tradução de: Your firs BASIC program
Apêndice
ISBN 85-215-0233-8

1. BASIC (Linguagem de programação para computadores) I. Fortuna, Eduardo Luiz Peixoto II. Título III. Série

85-0034 CDD – 001.6424

Reimpressão – 1987

Impresso no Brasil / Printed in Brazil

SÉRIE COMPUTAÇÃO
EDUARDO FORTUNA

Copy-desk / Maria Nazaré Mattos de Rezende
Capa / Daniel Le Noury
Ilustrações / Einar Vinje
Diagramação e montagem / Gilberto Lobato Bastos
Produção / Basileu Santiago Ribeiro

ISBN 85-215-0233-8
(Edição original: ISBN 0-89588-092-X)

AO LIVRO TÉCNICO S/A – Indústria e Comércio

RIO DE JANEIRO
Rua Sá Freire, 36/40
São Cristóvão – CEP 20930
Tel.: (021) 580-4868

SÃO PAULO
Rua Vitória, 486/496 – 2º andar
Centro – CEP 01210
Tel.: (011) 221-9986

1

Falando em BASIC

2

A comunicação com o Seu Computador

3

Calculando com BASIC

4

A Memorização de Valores e o Uso de Variáveis

5
Escrevendo um Programa de Forma Clara

6
Tomando Decisões

7
Automatizando as Repetições

8
Criando um Programa

9

Estudo de Caso: A Conversão Métrica

Apêndice

10

O Próximo Passo

Prefácio

Centenas, talvez milhares de livros foram escritos sobre BASIC.

Por que mais um? Simplesmente porque a audiência mudou. No passado, usar linguagem de programação, tal como o BASIC, era privilégio de uns poucos, que tinham acesso aos computadores. Os programadores eram um pequeno grupo de elite. Não é mais o caso, os computadores pessoais fizeram do BASIC a mais acessível e a linguagem mais amplamente utilizada em computadores. Além do mais, a maioria dos usuários tem um pequeno ou nenhum preparo anterior. Eles usam os computadores para diversão, educação, negócios ou em suas profissões.

Este livro é endereçado a este novo grupo de usuários. Ele não só parece diferente, ele é diferente. Ele pretende atingir o iniciante e, portanto, pressupõe que o leitor não tenha nenhum conhecimento técnico anterior. Este livro é para todos — dos 8 aos 88 anos — que queiram aprender rapidamente, como se iniciar com o BASIC.

O autor acredita que todo o novo usuário de computador que queira aprender a escrever seus próprios programas em BASIC são jovens e entusiasmados ou, então, de "cuca-fresca". Eles querem um ensinamento simples e direto para aprender o BASIC. Este é o caminho deste livro: ele se propõe a fazer o aprendizado do BASIC de forma fácil e divertida.

Além disso, este livro tem por objetivo ensinar a você o essencial do BASIC em poucas horas. Você pode estar escrevendo seu 1.º programa BASIC em até mesmo 1 hora. Com um pouco mais de tempo, você saberá o bastante para começar a escrever programas úteis e significativos.

O tempo está passando... vamos começar.

O autor lhe deseja uma jornada agradável ao longo do mágico caminho do conhecimento.

Rodnay Zaks

Como ler este livro

Este é um livro instrutivo. Você deve ler cada capítulo em seqüência e entender cada um antes de continuar para o próximo. Providenciei exercícios no final de cada capítulo para ajudar você a testar seus novos conhecimentos. Faça tantos quantos puder. As respostas dos exercícios seleccionados serão encontradas no Apêndice "A", ao final do livro.

Se você tem um computador, tente todos os programas. Para realmente aprender e lembrar, pratique e experimente. Este livro lhe dará a habilidade e o conhecimento de que você precisa para começar — mas lembre-se, nada pode substituir a experiência.

A importância deste livro está em permitir que você comece programando, rápida e efetivamente, em BASIC. Para atingir esse objetivo e tornar simples o seu caminho, tive que simplificar; em conseqüência, este livro não descreve todos os aspectos e concepções do BASIC — apenas os mais importantes.

Espero que você entenda tudo rapidamente e, em pouco tempo, esteja escrevendo seu primeiro programa BASIC, "curtindo" e se divertindo com o poder de suas novas habilidades em programação.

O que você vai aprender

Capítulo um — Explica a linguagem do computador e apresenta você aos heróis do livro: o Computador, o Intérprete, o Programa, as Instruções e outros caracteres do elenco.

Capítulo dois — Mostra como se comunicar com seu computador usando os recursos do teclado e do visor. Você aprenderá a digitar seus primeiros programas BASIC e a executá-los.

Capítulo três — Mostra como realizar cálculos com o BASIC.

Capítulo quatro — Ajuda você a escrever programas que possam ser usados repetidamente. Além disso, você aprenderá a usar variáveis, correta e efetivamente.

Capítulo cinco — Mostra como fazer seus programas claros e legíveis.

Capítulo seis — Mostra como tomar decisões complexas, baseadas na lógica e nos valores.

Capítulo sete — Explica como repetir tarefas automaticamente, usando laços de programa (*loops*).

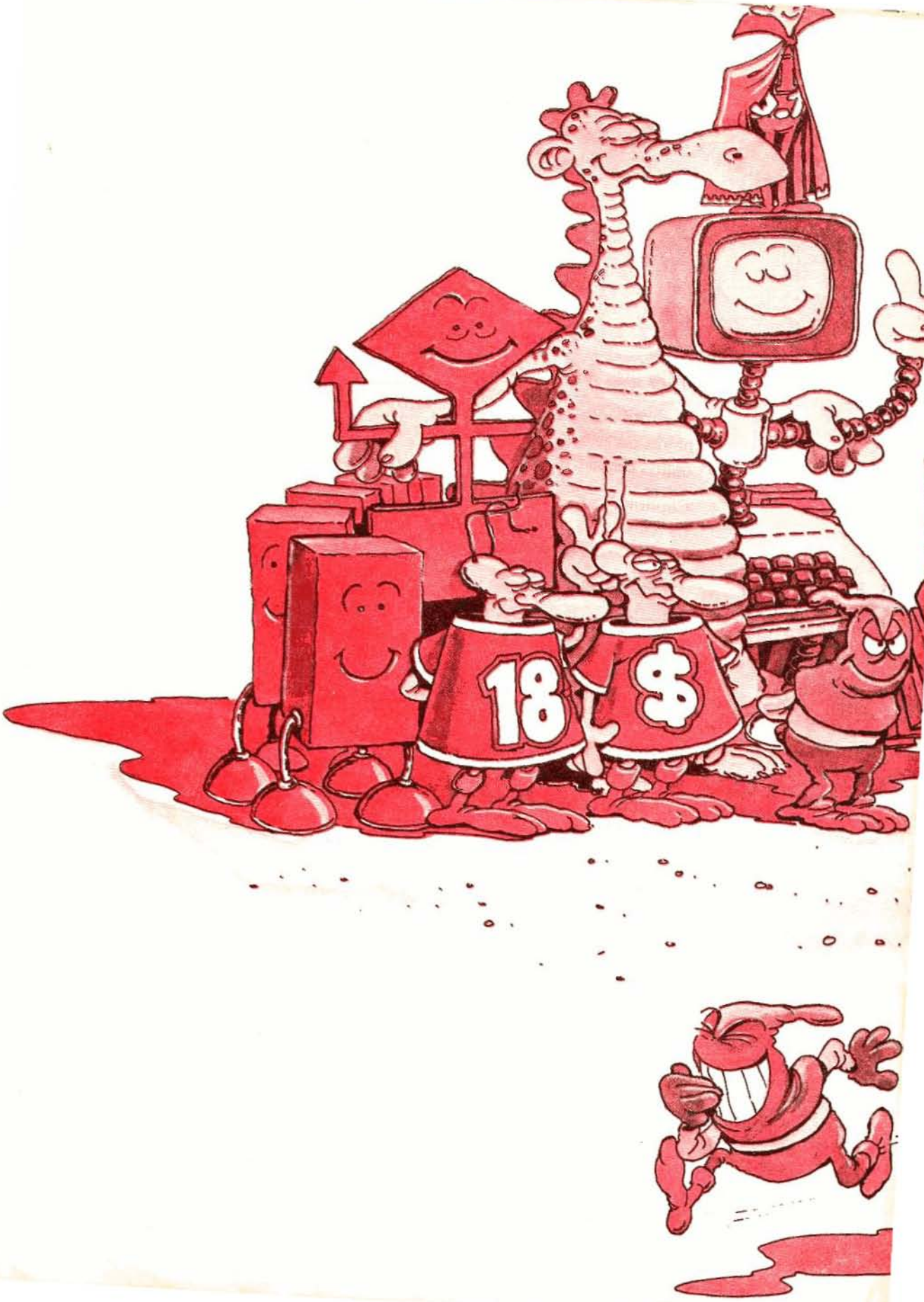
Capítulo oito — Mostra o método correto para projetar um programa: do algoritmo ao programa de execução documentado — incluindo o projeto do fluxograma.

Capítulo nove — Ajuda você a aplicar todas estas concepções no estudo de um caso prático.

Capítulo dez — Ajuda você a examinar o próximo passo para a especialização em programação.

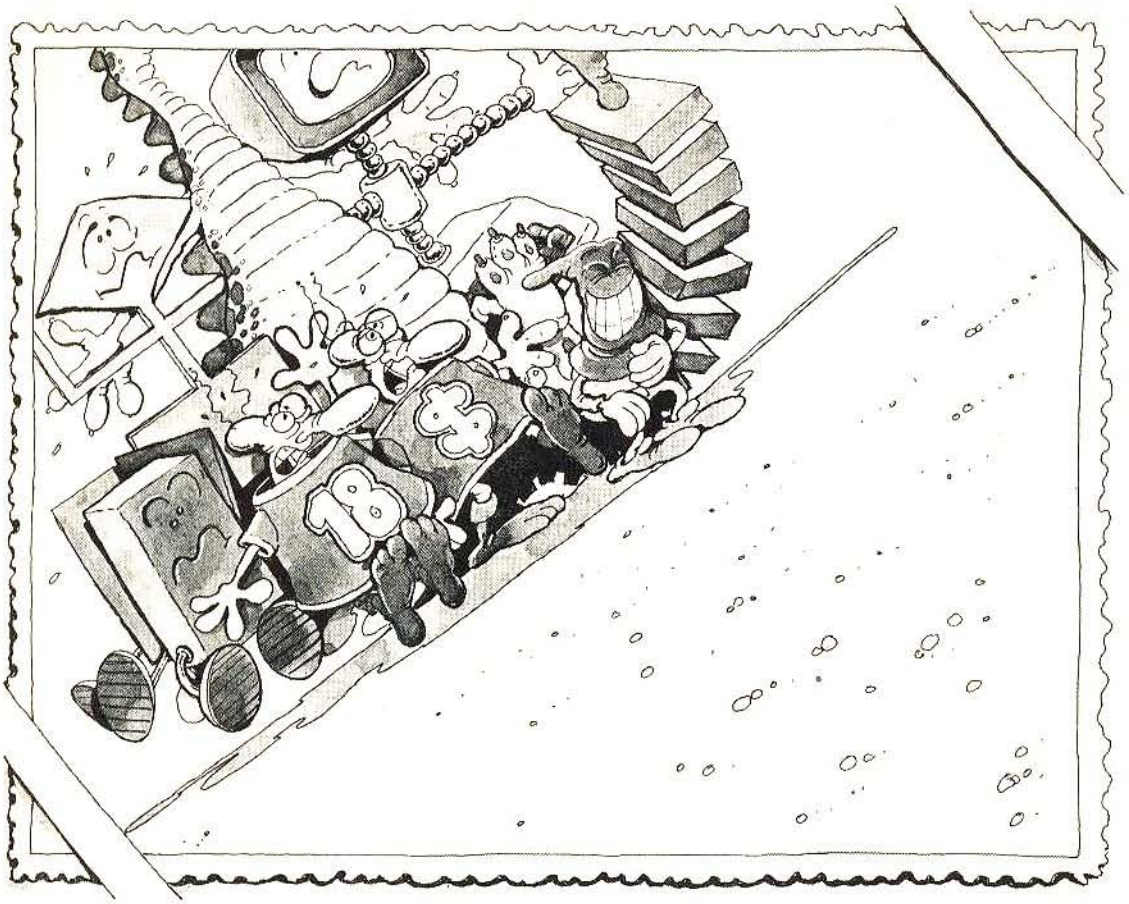
Apêndices A, B e C: Oferecem as respostas para os exercícios selecionados, uma lista de palavras comuns reservadas e um glossário.

Se você está pronto, vamos abrir o Álbum de Família e eu lhe apresentarei os heróis deste livro.


18
$

Conheça nossos Heróis

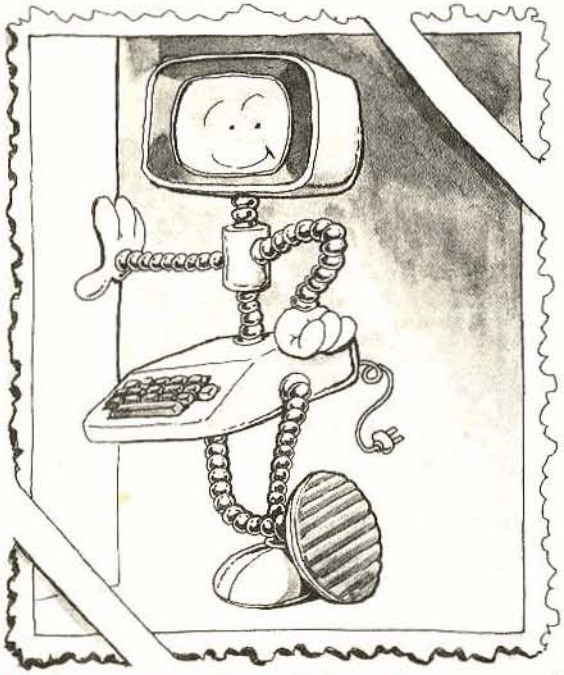
Apresentando (no sentido dos ponteiros do relógio).
Dino, o programador, o Intérprete BASIC abraçado com seu amigo o Computador, a Cobra Programa, o Travesso Bug, duas variáveis, Instruções de Programa, prontas para andar para os seus lugares assinalados, e o indispensável fluxograma, onde Dino descansa. Oba, nosso Bug parece estar pronto para alguma travessura!



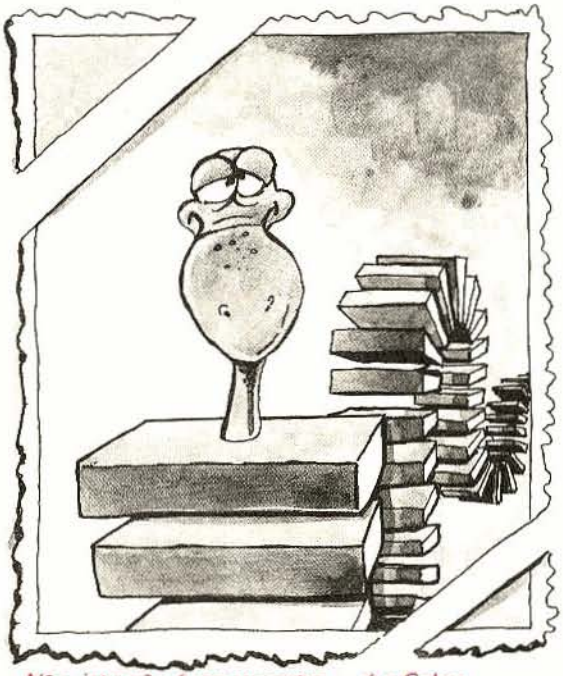
Desculpe...
Nosso travesso Bug
continua brincando.
É demais para
uma fotografia de grupo...

Este é o Intérprete BASIC. Ele vive na memória de seu computador. Seu trabalho é traduzir suas instruções para o computador. Ajudará você em tudo que puder.

Nunca esqueça este rosto. Este é o Bug. Ele infernizará sua vida. Faça o possível para mantê-lo longe de seus programas.



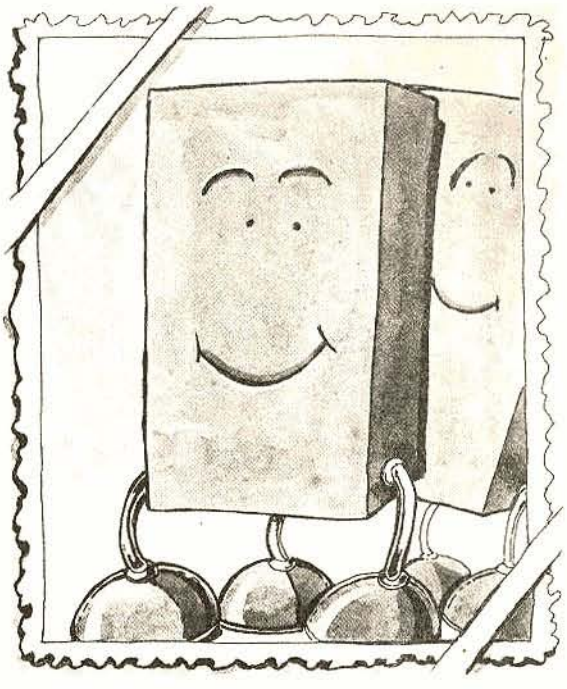
Este é seu amigo o Computador — a seu comando.



Não, isto não é um monstro – é a Cobra Programa. Ela é feita de instruções. Você aprenderá a montá-la. Ela é muito mansa quando você a conhece. Mantenha o Bug longe dela.

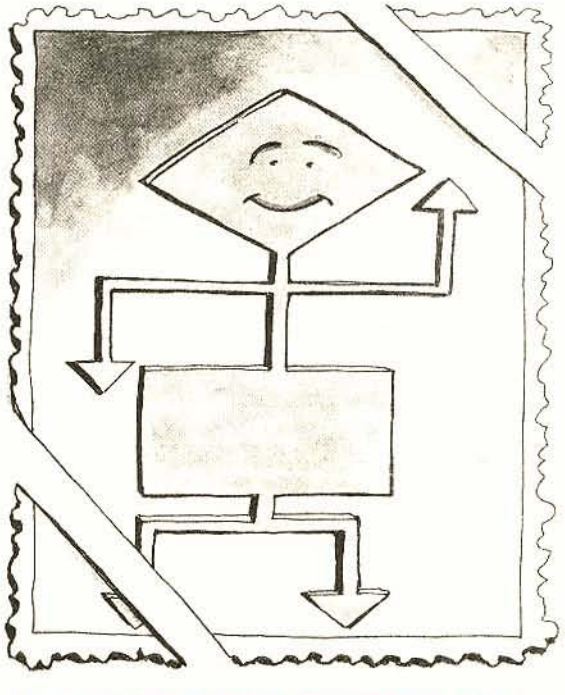
ÁLBUM DE FAMÍLIA

Este é Dino. Ele é amigável. Apesar de não ter tido uma educação formal, ele lhe mostrará como é simples escrever programas BASIC.



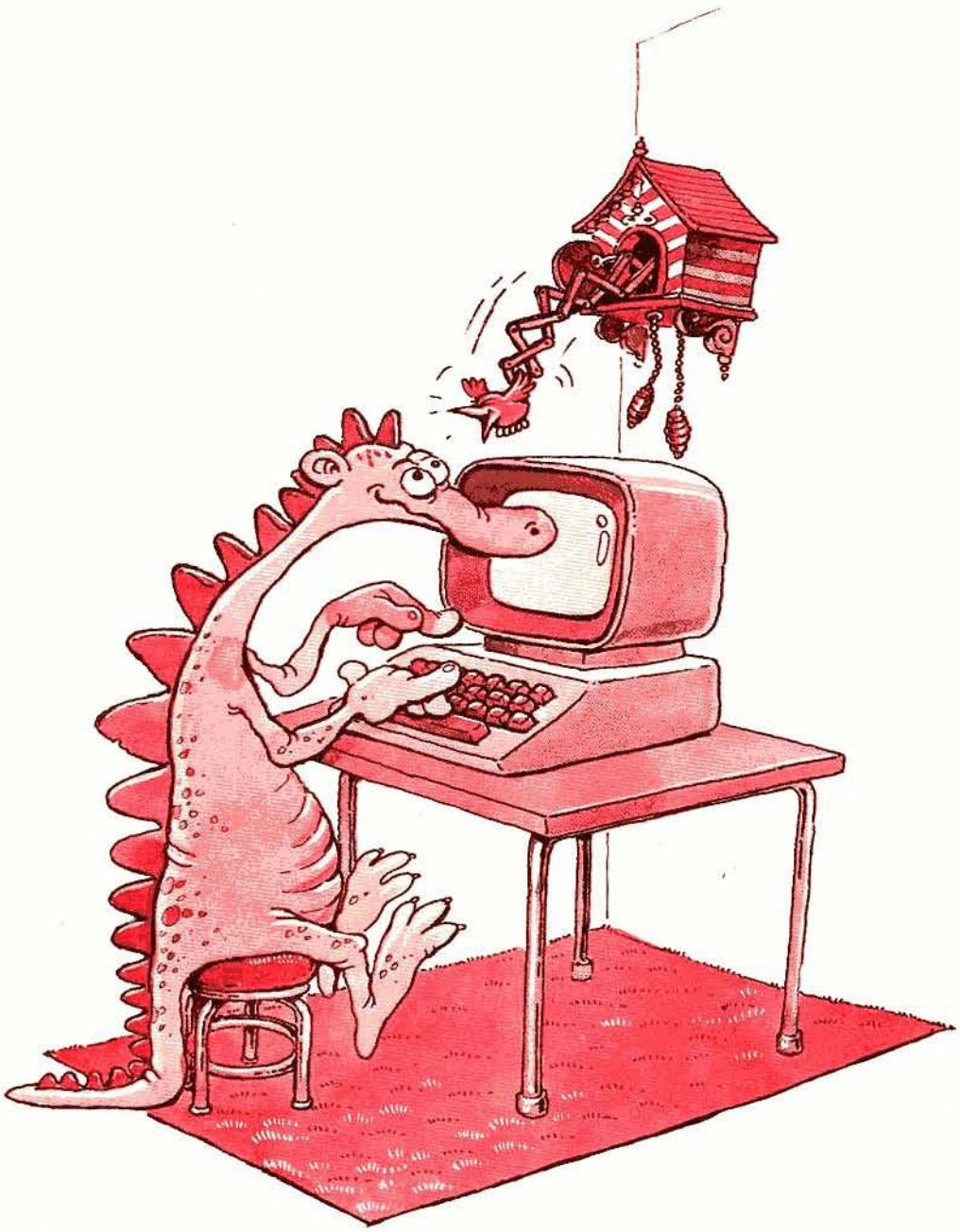
Aqui estão as instruções BASIC, prontas para integrar a Cobra Programa.

Esta é uma variável numérica. Seu casaco tem uma etiqueta com seu nome e valor impresso. Ela está infeliz porque quer voltar para sua área reservada na memória.



Este é seu melhor amigo, o Fluxograma. Ele o ajudará a projetar programas que funcionam.

Falando e



em BASIC

1

Eu afirmei no prefácio que "você estará escrevendo programas BASIC em até mesmo 1 hora" — então por que começar com um capítulo de conceitos e definições? Não estaremos perdendo tempo? Ao contrário: nosso propósito é *aprender* e *reter* informações, e um verdadeiro aprendizado requer profundidade de entendimento. A informação apresentada neste capítulo ajudará você a entender melhor o que é um programa, como um programa BASIC é executado, e o vocabulário dos computadores.

Antes de começar a escrever seu primeiro programa, existem alguns conceitos e definições importantes que você deve aprender. Quando você souber estes termos, eu poderei explicar *o que acontece* e *o que fazer*, da maneira mais simples possível, e você estará apto a seguir adiante rapidamente. Então, leia este capítulo cuidadosamente para que você realmente entenda o que está fazendo.

Nós começaremos por aprender a dar instruções a um computador — isto é chamado de *programação*. A seguir explanaremos a necessidade de uma *linguagem programada*, como o BASIC, seus dialetos e seus usos. Finalmente, examinaremos os componentes do *sistema do computador* e aprenderemos alguns dos jargões técnicos usados para descrever estes componentes.

Programando

Seu computador é uma máquina projetada para processar *informação* — tanto textual quanto numérica. Por exemplo, você pode fazer seu computador desenvolver palavras e sentenças em uma tela — isto é conhecido como *processamento de textos* — ou você pode fazê-lo converter uma medida expressa em onças para seu valor em gramas — isto é conhecido como *processamento numérico*. Para ordenar este processamento ao seu computador é preciso fornecer-lhe instruções na forma de "linguagem" que ele entenda. Cada computador pode "entender" apenas (isto é, reconhecer e executar) um pequeno número de instruções diferenciadas (algumas centenas).

As instruções que um computador pode entender diretamente são chamadas de *instruções de linguagem da máquina*. Estas instruções estão armazenadas em formato *binário*, isto é, em grupos de 0's e 1's na memória do computador. Cada 0 ou 1 é chamado de *bit* e um grupo de oito bits é chamado de *byte*.

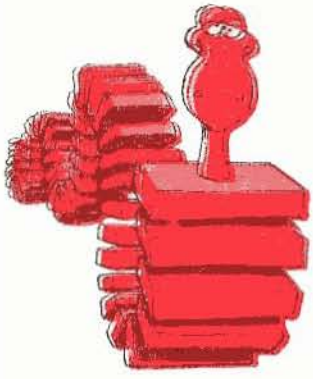
A seqüência de instruções que executa algo proveitoso é chamada de *programa* (uma seqüência de instruções que não executa nada é um *erro*). Seu computador executa uma instrução de cada vez. Infeliz-

mente, escrever um programa de computador (uma seqüência de instruções) na linguagem de máquina, isto é, na forma binária, é um processo lento e tedioso. O ideal seria darmos comandos escritos ou falados, na linguagem do dia a dia (em português) ao computador e ele executá-los. Mas isto não é possível, já que um computador não pode entender nada da nossa linguagem usual — seja ela falada ou escrita. A razão para isto é muito simples: um computador executa ordens exatas e estritas; ele é lógico e preciso, e isto requer instruções claras e objetivas, numa forma e seqüência próprias. O problema é que a linguagem falada mente por si só — sentenças podem ser ambíguas e muitas vezes sua interpretação depende de gestos e expressões faciais. Esse tipo de comunicação não pode ser interpretado por um computador.

Até mesmo o português escrito cuidadosamente pode ser insuficientemente preciso para um computador. Por exemplo: você não pode dizer a um robô computadorizado para "ir à cozinha e cozinhar um ovo", e esperar resultados, a menos que ele tenha sido programado para se movimentar em sua cozinha. Um robô tem que ser treinado (ou programado) antes de poder operar em um ambiente como o nosso. E, mesmo que seja treinado para saber se movimentar em *sua* cozinha, ele pode não ter sucesso na cozinha de seu amigo, porque as coisas podem estar colocadas em lugares diferentes. Lembre-se: a comunicação com um computador deve ser clara, precisa e objetiva.

É por esta razão que "linguagens" simplificadas foram inventadas para a comunicação com os computadores. A linguagem binária (também conhecida como linguagem de máquina) é a linguagem mais fácil para o entendimento do computador. No entanto, esta linguagem é de difícil entendimento para as pessoas. Outrossim, outras linguagens têm sido inventadas para facilitar a comunicação. Estas linguagens assemelham-se ao idioma comum e são chamadas de *linguagens de alto nível.*

"Eu amo BASIC!
Por favor, fale BASIC comigo."



"Lembra-se de mim? Eu sou o programa e sou feito de instruções."

Para a comunicação clara e efetiva com um computador, apenas um número limitado de palavras em nosso idioma pode ser usado, como ordens predeterminadas. Entretanto, sentenças ou *comandos* que especificam instruções para um computador devem obedecer às estritas regras gramaticais, chamadas de *sintaxe* da linguagem. A combinação deste vocabulário restrito com a sintaxe é chamada de *linguagem programada*. O BASIC é uma destas linguagens.

Em suma, uma *linguagem programada* é um conjunto de regras (a sintaxe), palavras e símbolos (o vocabulário), que permite a você escrever instruções para um computador, em um formato que seja claramente entendido. A seqüência dessas instruções é chamada de *programa*.

Aqui está um exemplo. Suponha que você queira

```
somar 2 + 2
```

e mostrar o resultado. Usando BASIC, você escreveria:

```
1 R = 2 + 2
2 PRINT R
```

onde R ocupa o lugar de "resultado".

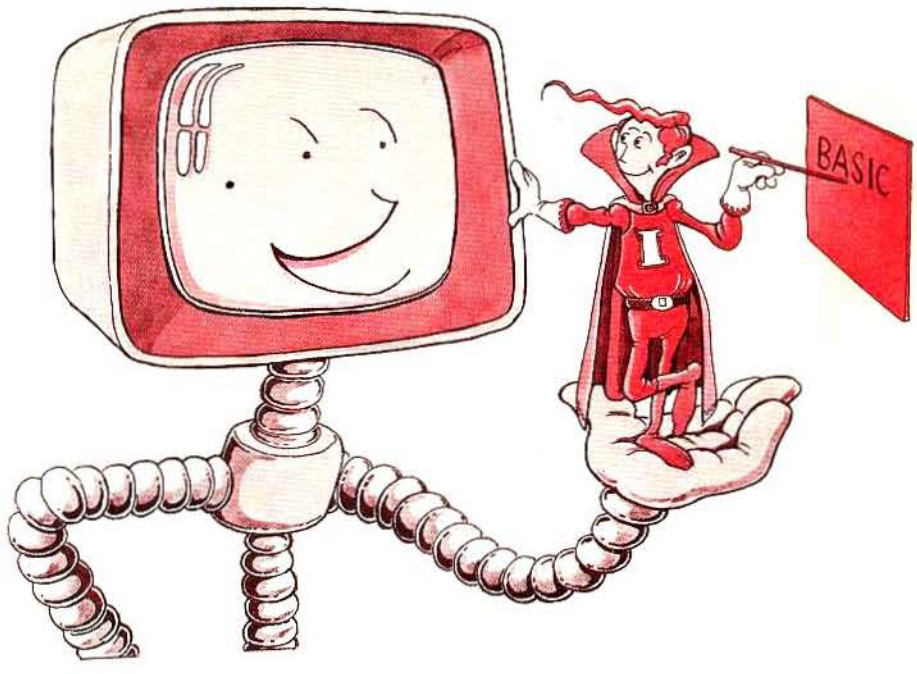
"Lembra-se de mim? Eu sou o Intérprete BASIC, pronto para traduzir suas instruções para o computador. Eu sou um programa e resido na memória de seu computador."

Mas, espere... Nós dissemos antes que a única linguagem que um computador pode entender diretamente é a *linguagem da máquina*; e agora estamos escrevendo instruções para um computador na linguagem fechada do idioma inglês. Isto não será uma contradição?

Não há contradição. Na verdade, o computador não pode entender diretamente o BASIC e, por outro lado, nenhuma outra *linguagem de programação de alto nível* (uma linguagem que usa o inglês em sentenças). Portanto, para ser entendido por um computador, um programa escrito em linguagem de alto nível como BASIC, deve ser *interpretado* por um programa especial, muito apropriadamente chamado de *intérprete*. Em outras palavras, você fala BASIC com seu computador, através de um intérprete. Então, para executar um programa BASIC, seu computador deve ter um intérprete BASIC. Vamos agora aprender o que um intérprete faz.

O Intérprete BASIC

O intérprete BASIC lê cada instrução BASIC que você digita no teclado e a traduz automaticamente numa seqüência de instruções em linguagem de máquina, que seu computador pode entender e executar. Este processo é completamente invisível para você (isto é, ele se processa dentro do seu computador). Desde que você ative o programa intérprete de seu computador, para todos os propósitos práticos,



"Quando eu estou na memória, seu computador fala BASIC."

seu computador pode falar em BASIC. Seu computador também pode ser capaz de falar outras linguagens de programação (se for provido dos intérpretes apropriados).

Há vários tipos de intérpretes BASIC que você pode usar no seu computador. Aqui nós explicaremos os dois tipos essenciais: o *residente* e o *não-residente*. A maioria dos pequenos computadores é provido de um intérprete BASIC residente. Este intérprete é chamado de residente porque ele mora permanentemente na memória do computador. Ele está à disposição imediatamente, tão logo o computador seja ligado, para qualquer ordem emitida como B ou BASIC.

No momento em que o símbolo de confirmação BASIC (*pronto* (>)) apareça em sua tela, você sabe que o computador está pronto para executar as instruções BASIC solicitadas.

Um intérprete *residente* geralmente tem uma desvantagem: ele pode prover, apenas, uma pequena versão do BASIC. Desde que um intérprete *residente* é "construído" na memória permanente de um computador, ele tem que ser pequeno em tamanho, pois o tamanho total da memória de um computador é limitado. A memória de um computador deve conter o programa, incluindo o Intérprete BASIC, e ainda possuir espaço suficiente para cálculos, administração do sistema e dados a serem operados. Estas necessidades de espaço limitam o tamanho, assim como diminuem a complexidade do intérprete *residente*. Em computadores que vêm com uma pequena quantidade de

memória, o intérprete residente é muitas vezes um "minúsculo BASIC", que impõe limites àquilo que você quer fazer com ele.

Alguns *residentes* BASIC, no entanto, são suficientes para aprender a programar em BASIC, ao menos para os propósitos deste livro. Mais tarde, quando você aprender a escrever programas mais complexos, provavelmente vai querer mais recursos. Você então irá querer um Intérprete *não-residente.*

Para prover mais características, um intérprete deve ser mais complexo e conseqüentemente, maior no tamanho. Será guardado num dispositivo de memória como um cassete ou *diskette*, de forma a não ocupar a maior parte da memória disponível, todo o tempo. Nos computadores bem pequenos é necessário, usualmente, adquirir memória adicional, passível de acomodar um intérprete maior.

O intérprete mais completo é fornecido com uma versão de BASIC chamada: *BASIC completo, BASIC ampliado, BASIC de ponto flutuante*, ou *BASIC avançado*, dependendo do fornecedor; em cassete BASIC ou *diskette* BASIC, dependendo do meio de armazenamento magnético utilizado. Usualmente, todos os programas escritos em BASIC *residente* podem ser executados sem mudanças num BASIC avançado. Nestes casos, as duas versões são descritas como sendo *ascendentemente compatíveis.*

Para usar um cassete ou um *diskette* BASIC, você deve primeiro transferir o intérprete do cassete ou do *diskette* para dentro da memória do computador. Isto é chamado de *carregamento* do intérprete. Você deve então dar uma ordem específica como F BASIC, para ativar seu intérprete BASIC avançado. O intérprete então mostra uma indicação na tela que é geralmente diferente da indicação do BASIC *residente*, para não criar nenhuma confusão. Somente então você pode introduzir as instruções BASIC.

Deixe-nos agora explicar o que é BASIC, como ele foi criado, e os dialetos resultantes.

O que é BASIC?

Linguagens de alto nível foram criadas para facilitar ao usuário dar instruções a um computador, isto é, programá-lo. Através dos anos, centenas de linguagens de programação foram inventadas.

No início, os computadores eram usados primariamente com propósitos científicos e as linguagens de programação eram criadas para facilitar os cálculos numéricos. Assim, o ascendente das linguagens: o FORTRAN ("FORmula TRANslator") foi criado propositalmente para cálculos numéricos específicos. O FORTRAN, no entanto, tinha várias deficiências e várias linguagens novas foram criadas. O BASIC é uma dessas linguagens; COBOL, APL e PASCAL foram outras que se tornaram amplamente utilizadas.

A invenção do BASIC representou a maior revolução. O BASIC foi criado para ser simples e fácil de aprender. Além do mais, ele é *interativo*. Vamos ver o que isto significa.

BASIC é a sigla de "Beginners All-purpose Symbolic Instruction Code" (Código de Instrução Simbólica de Propósito Geral para Principiantes). Ele foi inventado em 1964 por John Kemeny e Thomas Kurtz do Dartmouth College, trabalhando subvencionados pela National Science Foundation. A vantagem dos autores foi projetar uma linguagem que pudesse ser facilmente usada por um iniciante. Eles tiveram sucesso em seus propósitos. A partir desse dia, o BASIC tornou-se uma das mais fáceis linguagens de programação para se aprender.

Pelo fato de o BASIC ter sido projetado para ser interativo — na verdade, ele foi a primeira linguagem interativa — um usuário pode interagir com o programa em um terminal ao invés de submeter as séries de cartões perfurados IBM como nas antigas linguagens. O BASIC originalmente foi processado no sistema de tempo compartilhado, GE225 no Dartmouth College. Os terminais estavam disponíveis em todo o campus e vários usuários tinham acesso ao computador, simultaneamente.

O sucesso do BASIC foi rápido. A General Electric (GE) imediatamente decidiu usá-lo comercialmente. Kemeny e Kurtz publicaram o primeiro livro sobre BASIC em 1967. A Hewlett Packard (HP) e a Digital Equipment Corporation (DEC) decidiram tornar o BASIC utilizável na maioria de seus computadores.

O BASIC oferece duas enormes vantagens sobre linguagens como o FORTRAN:

1. **Para o usuário:** O BASIC é a linguagem mais fácil para se aprender, *especialmente* para os iniciantes.

2. **Para o fabricante:** O BASIC é a linguagem mais fácil para integrar em um computador. Pelo fato de a linguagem ser simples, o intérprete também é simples e racional, e exige apenas uma pequena participação da memória.

O terceiro fator que contribuiu para o enorme sucesso do BASIC foi o advento dos econômicos microcomputadores. Quando os microcomputadores se tornaram realmente utilizáveis, por volta de 1970, o BASIC tornou-se a linguagem de programação universal para estes pequenos computadores. Como o intérprete BASIC na versão simplificada do BASIC requer apenas 4K (4,096 bytes) de memória, mesmo os menores computadores puderam acomodar um residente BASIC. (Lembre-se de que um residente BASIC se refere a um intérprete armazenado permanentemente na memória do computador.) Os mais recentes computadores de porte têm grandes memórias (64K ou mais

— sendo que 1K se refere a 1,024 bytes) e podem, portanto, receber versões mais poderosas de BASIC.

Hoje, o BASIC é utilizado na maioria dos computadores. Através dos anos, os fabricantes têm adicionado extensões e "novas características" à linguagem de tal forma que a linguagem BASIC é provavelmente hoje a linguagem de computador menos padronizada. Não há dois BASIC iguais. Na verdade, o BASIC tornou-se uma família de linguagem e não mais uma linguagem única. Apesar de muitos padrões terem sido propostos, nenhum fez sucesso até hoje. Isto quer dizer que você deve reaprender BASIC em cada computador? Não de todo. Desde que você conheça a essência do BASIC, comum a todas as versões, você pode facilmente aprender as novas características que cada versão oferece. Todo BASIC tem essencialmente o mesmo núcleo de instruções. Você aprenderá mais sobre estas instruções, à medida que for lendo este livro.

Qual BASIC?

Seu computador provavelmente tem várias versões *diferentes* de BASIC, disponíveis. Inicialmente descreveremos os dois tipos essenciais de BASIC: o BASIC *residente* (geralmente um mini BASIC) e o BASIC *não-residente* (um BASIC completo ou BASIC avançado). Vamos agora examinar sumariamente algumas das suas diferenças.

Um "mini" ou "micro BASIC" tem menos características e convenções que um "BASIC avançado" ou um "BASIC ampliado". A limitação usual de um "mini BASIC" é que ele opera apenas com números inteiros, isto é, ele não opera com frações. Esta versão de BASIC é também chamada de "BASIC de números inteiros". Em contraste, uma versão BASIC aperfeiçoada que também opera com frações é chamada de "BASIC de ponto flutuante". Esta característica é altamente desejável se você planeja fazer cálculos.

Um mini BASIC geralmente armazena informações no cassete, mas não no *diskette*. Assim como geralmente armazena programas, mas não dados e também não opera com "arquivos". Normalmente, ele pode apenas manipular informações que são parte de um programa ou que tenham sido fornecidas pelo teclado. Por outro lado, um "BASIC avançado", usando unidades de disco, oferece grande conveniência e poder para operar um conjunto de informações (arquivo) e programas.

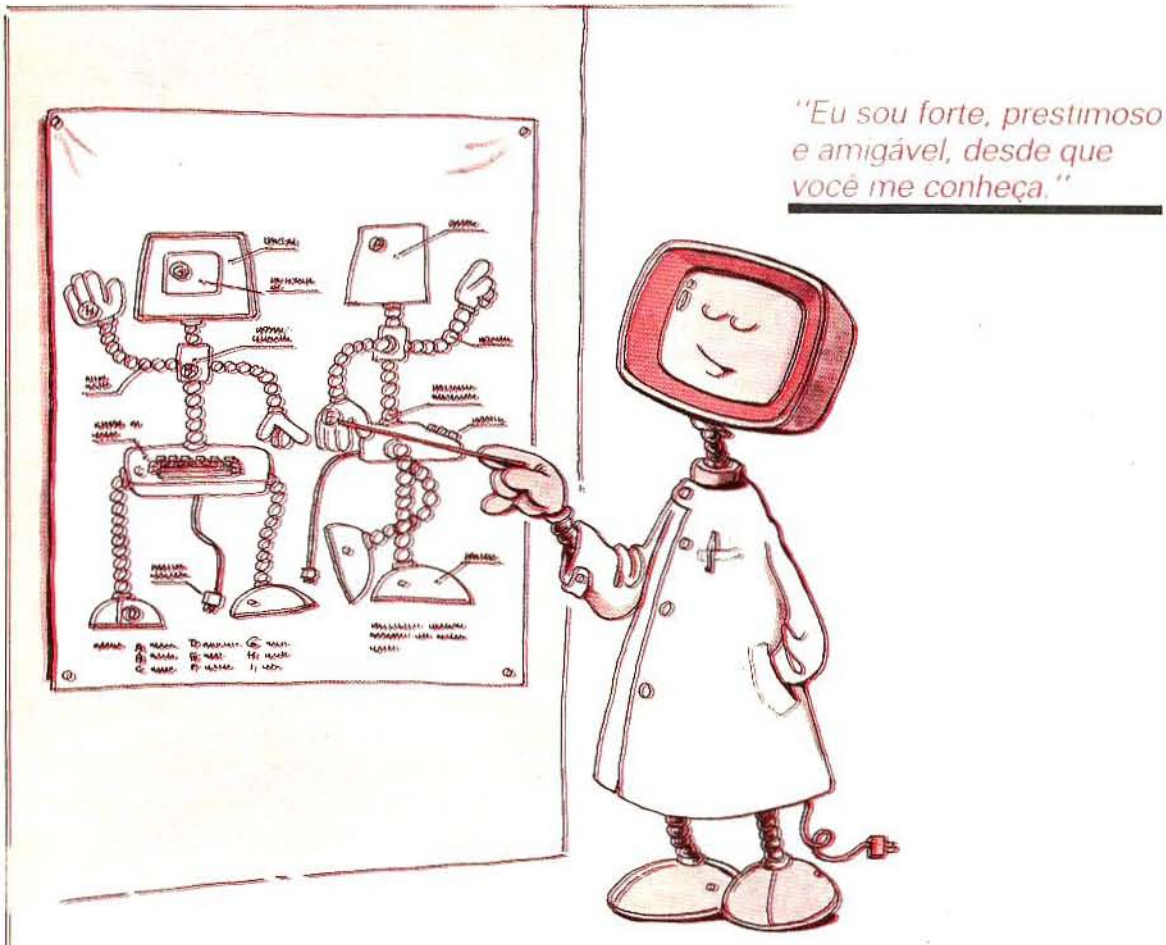
Alguns fabricantes fornecem versões ainda mais "avançadas" de BASIC que são específicas para os fabricantes. Um usuário é geralmente alertado para este fato pelo rótulo ou etiqueta "BASIC avançado". Isto representa que recursos mais poderosos estão disponíveis

para o programador experiente. Outrossim, o uso destes recursos especiais geralmente torna um programa BASIC incompatível com outros intérpretes BASIC.

O BASIC usado neste livro é um mini BASIC universal; dessa forma, você poderá aprender experiências aplicáveis a todas as versões BASIC. Desejamos, no entanto, apontar variações e extensões comuns, através do texto.

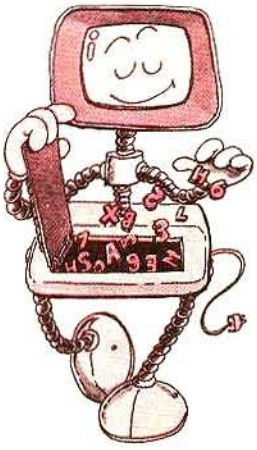
Em suma, você não precisa se preocupar com a versão BASIC que está usando. Mais tarde, se quiser escrever programas mais complexos, pode estudar o manual de referência para a versão BASIC que você quer usar. Finalmente a informação sobre algumas das características avançadas de BASIC, será apresentada no último capítulo deste livro.

Agora que entendemos mais sobre linguagens de programação, em geral, e em BASIC, em particular, vamos aprender mais sobre o computador e como ele processa informações.



Seu Computador

Seu computador processa informações e se comunica com você através do teclado e da tela, além de, eventualmente, uma impressora. O *teclado* é usado para mandar informações para o computador. Sempre que uma tecla for pressionada, o código eletrônico correspondente ao caráter desta tecla é enviado ao computador, onde ele é reconhecido e processado ou então ignorado. O teclado é seu *dispositivo de entrada*, ele provê de informação o computador.



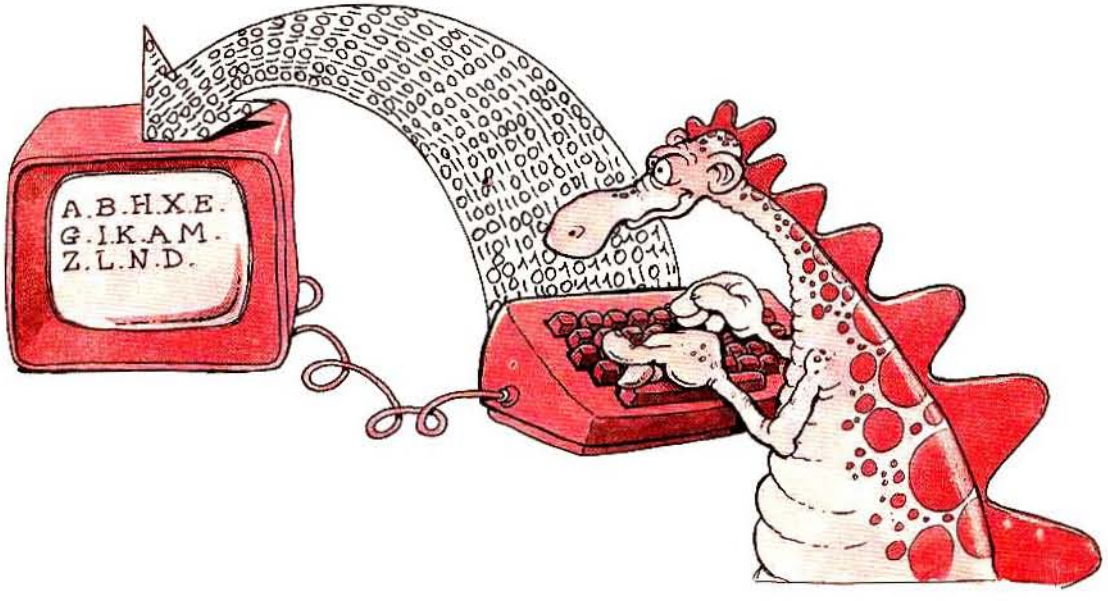
"Este é meu teclado."

O teclado manda informação para o computador

A *tela* ou C.R.T (tubo de raios catódicos) mostra as informações geradas pelo programa. Normalmente, cada caractere que você pressiona no teclado aparece na tela. Ele é enviado primeiramente para o computador e depois "surge" na tela. Geralmente não há conexão direta entre o teclado e a tela. Toda comunicação é feita através do computador. Isto está ilustrado na página seguinte.

Dependendo do projeto do sistema, o *computador* em si pode ser acondicionado em uma caixa separada ou integrada ao teclado, a tela e/ou a unidade de disco. Mas, independentemente do conjunto, o computador propriamente dito inclui uma unidade de processamento (a unidade central de processamento), uma memória e várias ligações (*interfaces* — dispositivos eletrônicos para ligação com impressoras e outros periféricos). Vamos examinar esses três elementos.

A *unidade central de processamento* (U.C.P.) identifica as instruções do programa, uma instrução de cada vez; traz da memória e as executa. A U.C.P. requer alguns componentes. Estes componentes são chamados de *circuitos integrados*, ou *chips*. Todos os microcompu-



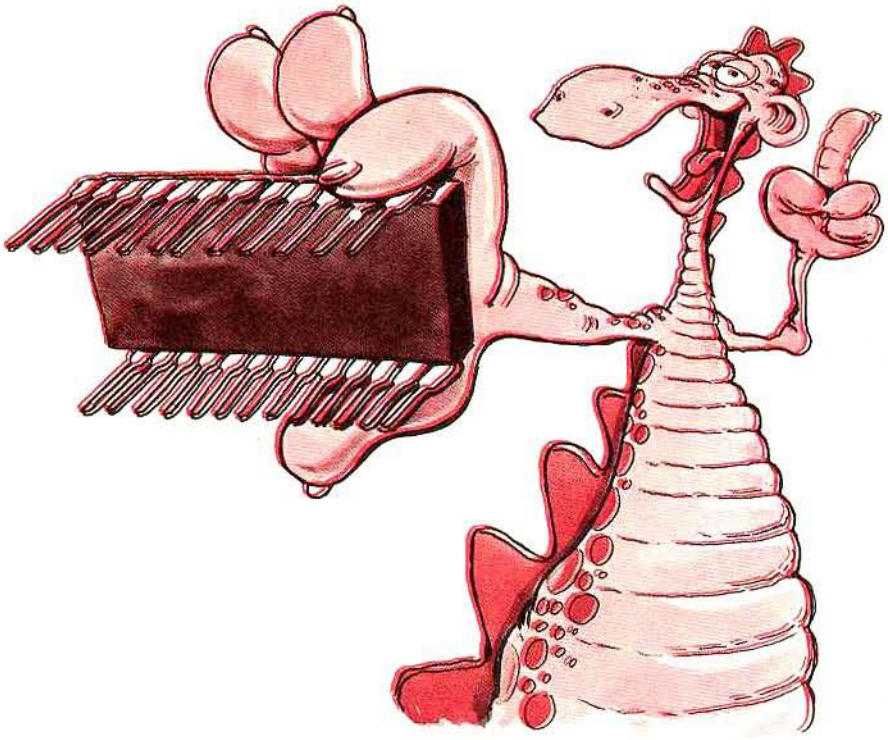
"Eu preciso de uma entrada para saber o que você quer."

tadores usam um *chip microprocessador* como um elemento essencial da U.C.P. Um *chip* típico é mostrado na ilustração seguinte.

A *memória* armazena os programas e todas as informações que o programa utiliza, lê ou produz, durante sua execução. Para a execução de um programa, este deve primeiro estar inserido na memória do computador. Por exemplo: se um programa está originariamente armazenado em cassete ou *diskette*, ele deve ser transferido para a memória do computador. Isto é chamado de *carga (loading)* do programa. A memória do computador deve ser suficiente para acomodar os programas de maior tamanho, além dos dados que o programa utilizará.

Dois tipos de memória estão presentes em seu computador: ROM e RAM. O tipo "normal" de memória que você usará para armazenar é a RAM ou memória de acesso aleatório. A RAM (Random Access Memory) é uma memória de onde se pode ler e onde se pode escrever. As informações podem ser escritas na RAM e lidas a partir dela. Uma RAM parece exatamente com um microprocessador mostrado abaixo, com exceção de que existe um *chip* diferente dentro dela. Infelizmente, no estágio atual da tecnologia, este tipo de memória é volátil, o conteúdo da RAM desaparece no momento em que a energia é desligada.

Por causa disto, ao final de uma sessão, você sempre deve armazenar seu programa em um meio não-volátil, como um cassete ou um *diskette*, caso queira guardá-lo. Lembre-se ainda de que deve haver um intérprete BASIC na memória (na RAM ou ROM) antes que você possa executar seu programa BASIC.

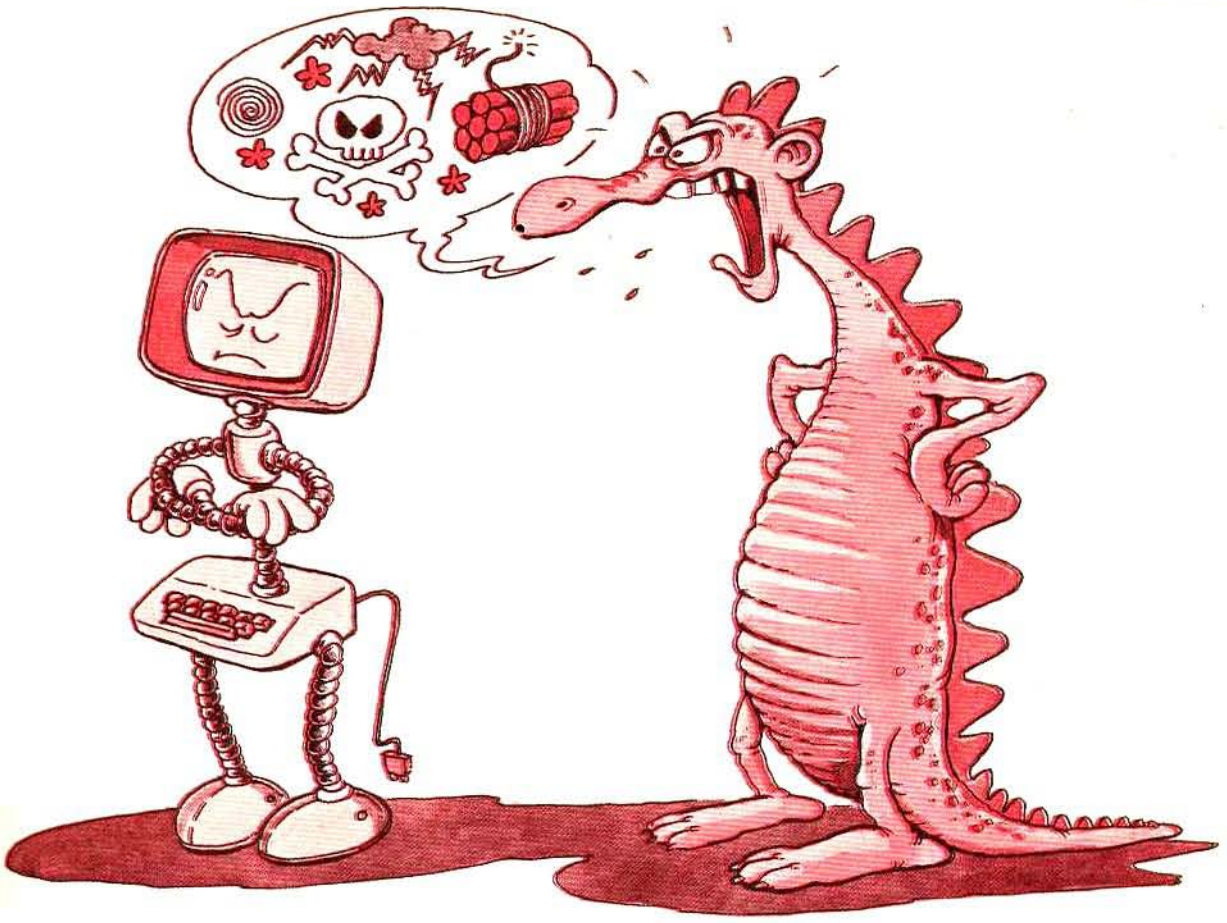


"Este é um chip microprocessador."

A ROM (Read Only Memory) é uma memória da qual apenas se pode ler. Este tipo é permanentemente carregada com programas pelo fabricante e não pode ser mudada. Ela é não-volátil (permanente) e nunca é apagada. Ela geralmente contém um intérprete BASIC residente, além de um programa especial, o *monitor*, o qual é necessário para a comunicação com o computador a partir do momento em que ele é ligado.

Se sua ROM não contém nada, o computador não saberá o que fazer quando você pressionar as teclas de seu teclado. No mínimo, seu ROM deve conter um monitor. Este monitor examina as informações enviadas pelo teclado e reage executando ações gerenciais como ativar o intérprete BASIC residente ou o carregamento de um programa do cassete.

Você não pode usar o ROM para armazenar qualquer outro programa. Todos os outros programas que você der entrada no seu computador são carregados na RAM. Algumas vezes você pode usar cartuchos de programas em seu computador. Nestes casos os programas são armazenados em *chips* ROM dentro do cartucho.



Seu computador não fará nada até que você introduza um programa em sua memória

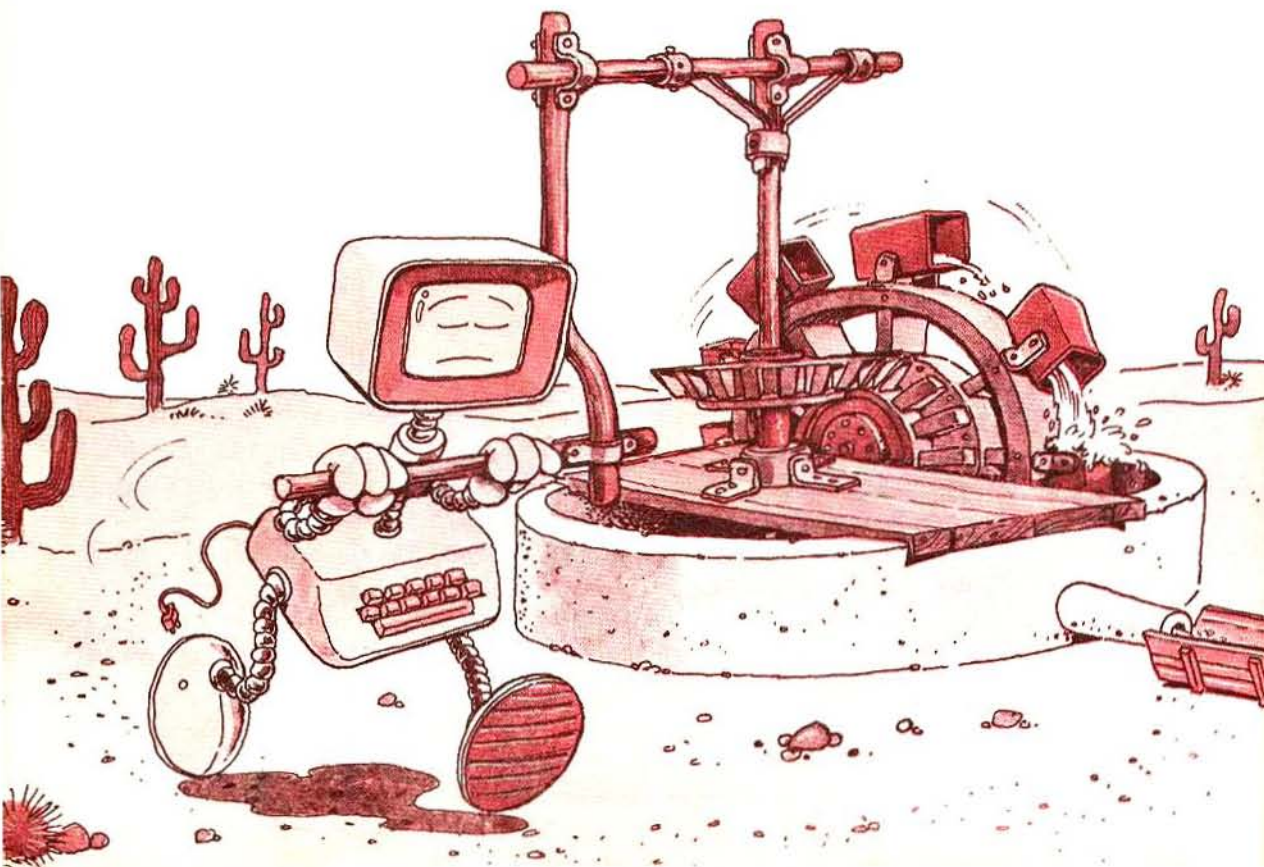
Finalmente, dois dispositivos adicionais são comumente conectados a um computador: um banco de memória e uma impressora. Estes dispositivos são conectados através de *interfaces* — os dispositivos eletrônicos necessários para conexão de periféricos. O periférico pode ser um arquivo cassete ou uma ou mais *unidades de disco*. (**Nota**: Ambos os dispositivos usam um meio magnético para registrar informações e podem armazenar muito mais informações do que a memória interna eletrônica do computador). Estes dispositivos especiais requerem uma conexão eletrônica (*interface*) especial, na caixa do computador, que permita sua comunicação com o computador. Muitos computadores pessoais trazem embutida uma *interface* para um arquivo de fita; no entanto, uma ou mais *interfaces* separadas são geralmente necessárias para conectar uma ou mais unidades de disco ou impressora ao computador.

Uma impressora é necessária para se obter a permanente impressão dos programas ou dos resultados. Uma impressora é um *disposi-*

tivo de saída, assim como a tela de um tubo de raios catódicos. Existem instruções específicas no BASIC para comandar informações para ambos: a tela ou a impressora.

Finalmente, um *modem* é um outro dispositivo freqüentemente utilizado. Um *modem* permite a você se comunicar com outro computador ou terminal, através da linha telefônica comum. Ele é muito útil quando usado por uma rede de lojas comerciais ou para ter acesso a *bancos de dados* (conjunto de informações). Nós aprendemos, até aqui, o vocabulário necessário. Antes de passarmos para o capítulo 2 e começarmos a usar o computador, *é necessário ainda uma recomendação.*

O programa monitor está o tempo todo pronto para executar todas as tarefas comuns

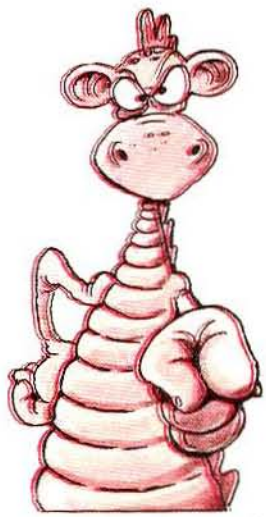


Os Computadores e a Sintaxe

Os computadores são rápidos, pacientes e precisos. Eles fazem apenas aquilo que lhes é dito para fazer, e o fazem perfeitamente. Para se comunicar com sucesso com seu computador, você deve ser exato. Se você cometer um erro ou engano ao escrever uma instrução BASIC, não causará danos ao computador, mas seu programa não será executado com sucesso. Ele geralmente irá parar e dizer: "erro de sintaxe".

Recorde que a *sintaxe* é um grupo de regras que especificam a forma correta de escrever uma instrução BASIC. Regras de sintaxe são rígidas. Por exemplo: você não pode usar um significado aproximado. Se as regras especificam um ponto, você não pode usar uma vírgula ou dois pontos em seu lugar. Qualquer desvio é rigidamente interpretado pelo computador e tem um significado preciso. Qualquer desvio representará um erro, ou ao final, resultados inesperados.

Lembre que se você não obedecer exatamente às regras, seu programa provavelmente não funcionará. Não tente ser criativo. As regras são simples, diretas e fáceis de seguir, ao escrever as instruções do programa. É melhor guardar sua criatividade para o projeto do programa e para o planejamento das tarefas a serem realizadas. Portanto, é importante que você siga cuidadosamente as instruções e recomendações dadas neste livro.



"Lembre-se ... seja exato."

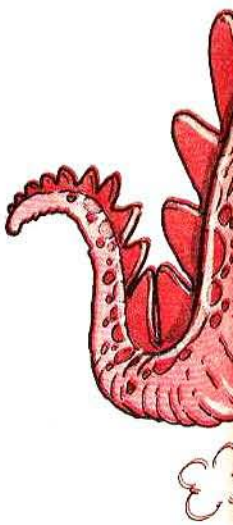
A Comunicaç

2

Neste capítulo você aprenderá a se comunicar com seu computador. Você aprenderá as instruções em BASIC e a fazer o computador escrever palavras e sentenças. A informação trocada entre você e o computador incluirá programas (instruções BASIC que você escreve) e dados (os números ou caracteres que você envia ou recebe).

Em primeiro lugar você aprenderá a usar o teclado de um computador e então poderá começar a enviar-lhe instruções. Em particular, você aprenderá a mover o cursor na tela e corrigir os erros de impressão. Então você escreverá suas primeiras instruções em BASIC e fará o computador desenvolver mensagens na tela. Você aprenderá a diferença entre execução *imediata* e execução *programada*. Finalmente, você aprenderá os passos que envolvem a escrita de um programa simples e sua execução.

No final deste capítulo você estará familiarizado com a experiência básica necessária para a comunicação com seu computador.



ão com o seu Computador

Usando o Teclado

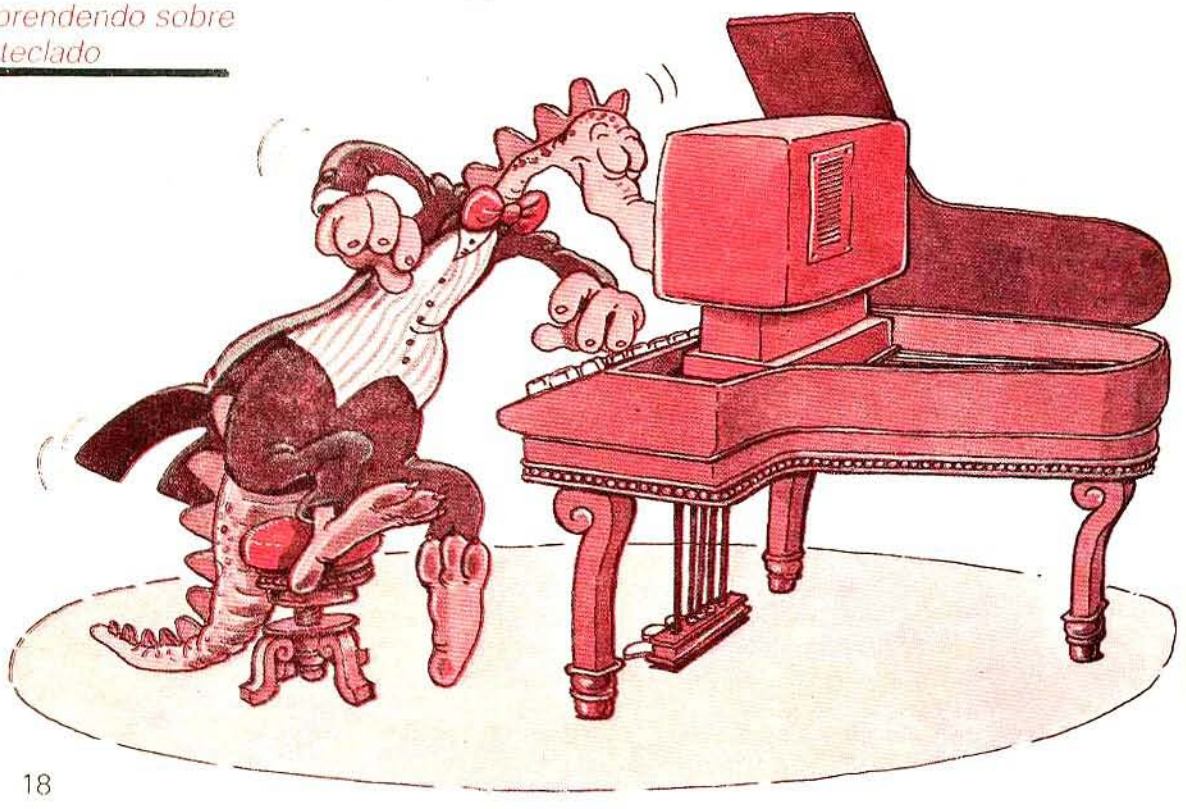
O desenho abaixo mostra um teclado de computador. Note que ele é muito semelhante ao teclado comum de uma máquina de escrever, exceto pelas teclas adicionais especiais.

O teclado do computador

O teclado do computador pode ter vários modelos e combinações de teclas e, com exceção de alguns poucos detalhes, todos têm características essenciais comuns. As teclas principais são:

1. As letras do alfabeto (de A a Z)
2. Os dígitos (de 0 a 9)
3. Símbolos como =, +, *, ", e $
4. Uma tecla de "retorno do carro", geralmente marcada com RETURN, CR, ou ↵
5. Uma tecla SHIFT (Troca) e uma tecla de controle — CTRL
6. Uma tecla RUBOUT (apagar) e uma tecla ESC ou BREAK (Interrupção).

Aprendendo sobre o teclado



Alguns teclados estão providos de teclas adicionais, como um conjunto separado de teclas numéricas e teclas com funções especiais. Agora examinaremos a função e o uso de todas estas teclas.

Caracteres, Números e Símbolos

O propósito de cada tecla de *letra, número* e *símbolo especial* é óbvio. Estas teclas são utilizadas com o mesmo propósito que os da máquina de escrever comum.

A Tecla RETURN (tecla de introdução)

Numa máquina de escrever, a tecla de retorno realiza duas ações: ela reconduz o carro ao começo de uma linha, e avança o papel para a linha seguinte. Daí o seu nome. Num computador, entretanto, a tecla RETURN geralmente move o *cursor* para o começo da linha seguinte na tela (o cursor é um quadrado brilhante ou um traço de sublinhar que indica a posição seguinte onde um caractere será escrito na tela). Em um computador, a tecla RETURN poderia ser chamada apropriadamente de tecla de entrada (ENTER), já que sua função precípua é introduzir um caractere, uma linha do texto ou um dado na memória do computador. Na verdade, alguns teclados nomeiam esta tecla de ENTER. Outros identificam-na com uma seta (↵).

Em nenhum caso, uma instrução para o computador, incluindo as instruções em BASIC, podem ser terminadas com um RETURN. O RETURN significa "entrada de linha na memória". Na verdade, o que você digita numa linha é ignorado pelo computador até você pressionar RETURN. Desta maneira, você pode corrigir erros que tenha cometido antes de introduzir a linha na memória do computador.

O RETURN é usado apenas em tempo de digitação. Apesar de necessário, o RETURN não é armazenado como parte de uma instrução de programa. Neste capítulo introdutório, para ajudá-lo a aprender, mostraremos todos os caracteres que você deve digitar e desenvolveremos um símbolo indicando RETURN no final de cada linha. Este símbolo aparecerá como ↵.

Lembre-se: você tem que pressionar a tecla RETURN para transmitir uma instrução ao computador. Da mesma forma, quando o computador, mais tarde, lhe pedir valores, você os introduzirá pressionando a tecla RETURN.

A Tecla SHIFT (tecla de troca)

A tecla de troca trabalha igual àquela da máquina de escrever. Ela permite a você trocar os símbolos que aparecem nas teclas. Com exceção das teclas de letras, a maioria das teclas tem dois símbolos im-

pressos: um superior e um inferior. Se você não pressionar a tecla de troca, o símbolo inferior é o que geralmente aparece quando a tecla é pressionada. Quando você pressiona SHIFT junto com a tecla, o símbolo superior será gerado. Com as teclas de letras você gera letras maiúsculas e minúsculas; no entanto, em alguns teclados simplificados, as teclas de letras geram apenas letras maiúsculas; a posição inferior é usada por outros símbolos ou funções. O BASIC "padrão" exige que as instruções sejam escritas com letras maiúsculas. Portanto, neste livro, usaremos apenas letras maiúsculas em nossos programas. Se o seu teclado for provido de letras minúsculas, você pode muito bem introduzir e manipular textos que as incluam.

A maioria dos teclados também são providos de uma tecla de fixação (CAPS LOCK), o que lhe permite operar o teclado na posição superior. Batendo nesta tecla *novamente*, liberta-se a "fixação". O propósito da tecla SHIFT é reduzir o número total de teclas. Usando SHIFT, dobra-se o número de símbolos ou comandos que cada tecla pode gerar.

Vamos praticar o que acabamos de aprender. Vá ao seu teclado e bata teclas aleatoriamente. Pressione qualquer tecla, em seguida use a tecla SHIFT e olhe os caracteres resultantes na tela. Agora pressione RETURN e veja o que acontece.

A Tecla CTRL (tecla de controle)

A tecla de controle (CTRL) é utilizada para introduzir rapidamente comandos, freqüentemente utilizados no computador. Não existe em máquinas de escrever. A tecla CONTROL é usada, como a tecla SHIFT, pressionando-a para baixo, segurando-a embaixo e pressionando uma outra tecla do teclado ao mesmo tempo. Isto é chamado de geração de um caractere de controle. Por exemplo, "CTRL A" é gerado apertando-se e segurando-se a tecla CTRL e a tecla A simultaneamente. Esta é a maneira conveniente de gerar um comando ou código de controle — pressionando apenas duas teclas. (**Nota**: os *códigos de controle* são fornecidos para facilitar o uso dos comandos mais freqüentemente utilizados, tal como o movimento do cursor na tela, isto é, esquerda, direita, para cima, para baixo, por uma posição, palavra ou sentença — e para suprimir ou inserir um caractere ou uma linha inteira de texto). O uso de caracteres de controle é específico para cada programa usado no computador, e seus efeitos estão explicados na documentação do programa. Nós não usaremos nenhum código de controle neste livro. Você pode, no entanto, querer aprender ao menos um: o código requerido para interromper um programa mal feito (normalmente CTRL C).

A Tecla RUBOUT (tecla de apagar)

A tecla RUBOUT ou tecla de "apagar" é usada para uma função absolutamente necessária. Ela pode ser incluída em alguns teclados, desde que a mesma função possa ser obtida pelo movimento do cursor de volta sobre o caractere.

A Tecla ESC ou BREAK (tecla de interrupção)

A tecla ESC ou BREAK é geralmente incluída como um "código de controle" padrão, comum a todos os programas. Ela permite que você pare qualquer programa, apertando apenas uma tecla. Cada programa, seja um intérprete BASIC ou seu próprio programa, deve permitir um comando especial como END ou EXIT que finaliza sua execução. No entanto, no caso de acontecer alguma coisa inesperada, e você quiser parar a execução do programa imediatamente, você pode geralmente usar a tecla ESC. Às vezes, você também pode usar um código de controle (como CTRL C) e dessa forma usar a tecla CTRL para executar esta função.

O teclado numérico

O Teclado Numérico

O teclado numérico (mostrado à esquerda) é muito usado em aplicações onde os números devem ser digitados rapidamente. Este teclado é usado do mesmo modo que um teclado de calculadora de mesa. Suas teclas simplesmente duplicam as funções das teclas usuais de um teclado regular (de 0 a 9, +, −, ×, ÷, e =).

As Teclas de Função

As teclas de função são um modo prático de introduzir no computador os comandos de uso mais freqüente. Pressionando uma tecla, de função única, obtemos o mesmo resultado da digitação: uma ordem longa como PRINT ou EXECUTE.

Assim, você pode pressionar uma única tecla para apagar um caractere, inserir um bloco de caracteres ou executar um programa. Estas teclas são feitas especialmente pelos fabricantes e dispensam esforços de digitação. Entre as teclas úteis podemos citar as que giram a tecla, limpam a tela, suspendem a página na tela ou abaixam-na, e as teclas que posicionam o cursor: ↑, ↓, ←, →

O Cursor

Lembre-se de que o cursor é um quadrado ou traço de sublinhar que mostra sua exata posição na tela. Geralmente ele pisca, de forma que

você pode vê-lo facilmente. Aqui está um cursor, mostrando onde você está na tela, após ter digitado HELLO:

```
> HELLO □
```

Movendo o cursor para trás e para frente, para cima e para baixo, na tela, você pode sobrepor caracteres e modificar qualquer texto que tenha digitado anteriormente. Isto é muito conveniente para se fazer mudanças. Você usará o cursor, freqüentemente, para corrigir erros de digitação.

A maioria dos teclados também possui ao menos 4 teclas de funções para posicionar o cursor na tela: ↑, ↓, ←, →.

Aprenda a mover o cursor na tela

Você pode querer, agora, voltar ao teclado e praticar algumas das coisas que você aprendeu. Quando você se sentir razoavelmente familiarizado com o teclado, volte e aprenda a escrever instruções BASIC para o computador.

Falando em BASIC

Para falar BASIC com seu computador, você precisa de um intérprete BASIC na memória. Se o seu computador tem um intérprete BASIC

residente, você não deve fazer nada. No entanto, alguns sistemas exigem que você digite:

```
B ↵
```

ou:

```
BASIC ↵
```

para ativar o intérprete. Se o seu computador não tem um intérprete BASIC residente, você deve carregá-lo (transferi-lo) de um *diskette* ou de um cassete. Quando o intérprete está ativado você obterá uma confirmação como:

```
XBASIC 2.1 READY
>
```

XBASIC é o nome dado ao intérprete, 2.1 é a versão. Sucessivas versões são liberadas continuamente e estes números permitem que você distinga uma das outras. > é *espaço*. (O símbolo de espaço pode variar. Neste livro usaremos o símbolo >.) O espaço ou pausa é uma mensagem do intérprete BASIC, que significa: "Estou pronto. Vá adiante e diga-me o que fazer". Tão logo você veja este símbolo, saberá que o intérprete BASIC está pronto e esperando por suas instruções.

Agora prosseguiremos sabendo que:

1. Um intérprete BASIC está na memória de seu computador.

2. A pausa do BASIC é usada pelo seu intérprete (identificada aqui pelo símbolo >) e aparece em sua tela. Se o símbolo não aparecer, pressione RETURN e veja o que acontece. Se não funcionar, desligue seu computador e ligue-o novamente.

Agora nós escreveremos nossa primeira instrução BASIC para o computador. Digite o seguinte (exatamente como aparece aqui):

```
PRINT "ALO"
```

A tela mostrará o seguinte:

```
XBASIC 2.1 READY
> PRINT "ALO" □
```

O > na esquerda é a *pausa*, dizendo a você que o intérprete está esperando por uma instrução. Os caracteres da direita foram digitados por você. Nada aconteceu ainda. Você lembra por quê?

Falta pressionar a tecla RETURN para dar *entrada* à sua instrução. Agora pressione RETURN. Sua tela mostrará o seguinte:

```
XBASIC 2.1 READY
> PRINT "ALO"
ALO
> □
```

O intérprete recebeu sua instrução e imediatamente executou-a escrevendo ALO como foi requisitado. Em seguida executará uma nova pausa (>) dizendo a você que está pronto para uma nova instrução.

Vamos examinar nossa primeira instrução BASIC mais detalhadamente:

```
PRINT "ALO"
```

Esta instrução tem 2 partes: PRINT e "ALO". PRINT é uma *palavra reservada*, que tem um sentido específico para o intérprete. "ALO" é a mensagem a ser executada entre aspas. Você pode usar qualquer mensagem entre aspas. Vamos tentar outra. Digite:

```
PRINT "ESTE E UM OUTRO TESTE" ↵
```

O que se segue deverá aparecer em sua tela:

```
> PRINT "ESTE E UM OUTRO TESTE"
ESTE E UM OUTRO TESTE
> □
```

Tente novamente. Execute sua própria mensagem, mas lembre-se, se você esquecer uma das aspas ou se escrever errado PRINT, não funcionará e você receberá uma mensagem de erro. Vá adiante. Tente. Você não provocará nenhum dano ao compùtador.

Você talvez não se lembre por que esta instrução é chamada "PRINT", quando na verdade ela meramente mostra a informação em sua tela e não imprime. Mesmo que você tenha uma impressora li-

gada ao seu computador, nada será impresso. Você provavelmente esqueceu a razão. Os primeiros terminais eram como máquinas de escrever que, na verdade, imprimiam (e não mostravam) a informação. E, mesmo que a tecnologia tenha mudado, a instrução BASIC não mudou. Uma outra instrução BASIC é hoje usada para enviar informações à impressora, ao invés de à tela do vídeo. Esta instrução é chamada LPRINT porque é usada essencialmente para *listar* programas no papel.

Antes de prosseguir, esteja certo de que a pausa BASIC está em sua tela, isto é, de que o intérprete BASIC está pronto para aceitar um outro comando. Se a pausa não aparecer, você não pode executar um comando BASIC. Tente pressionar RETURN ou CTRL C, ou, caso não funcione, tente religar seu sistema.

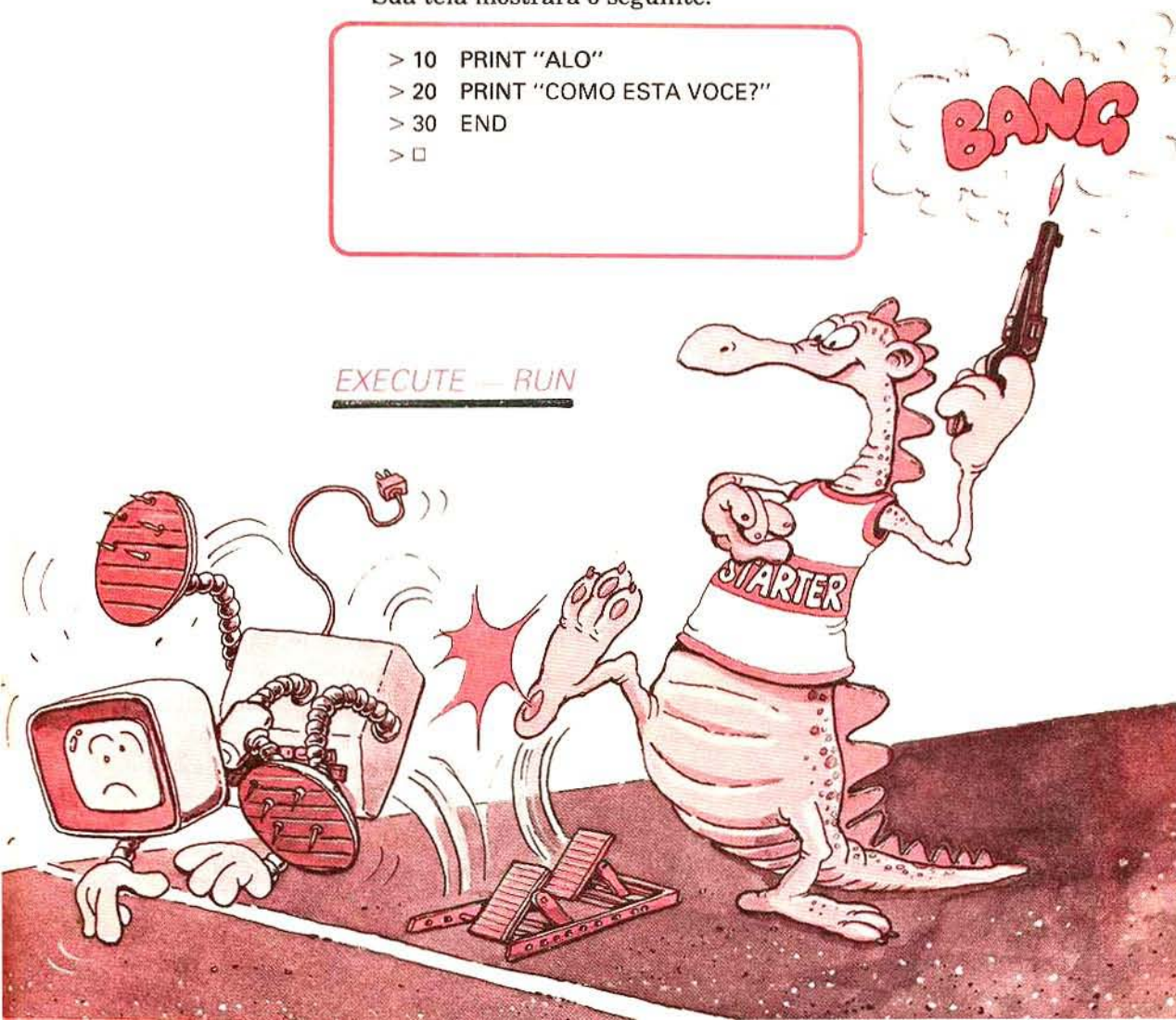
Nós escreveremos agora nosso primeiro programa BASIC, ao invés de executar uma única instrução. Digite o seguinte:

```
10  PRINT "ALO" ↵
20  PRINT "COMO ESTA VOCE?" ↵
30  END ↵
```

Sua tela mostrará o seguinte:

```
> 10  PRINT "ALO"
> 20  PRINT "COMO ESTA VOCE?"
> 30  END
> □
```

EXECUTE — RUN

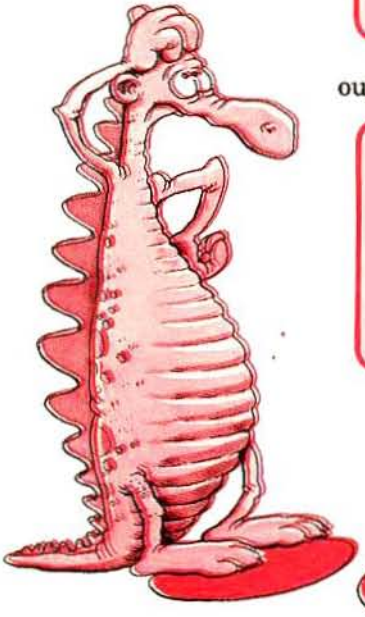


Você deve estar surpreso pois nada aconteceu. O computador simplesmente responde com um >. Estas três linhas são mais do que três instruções BASIC. Elas constituem um pequeno *programa* BASIC. Note que cada linha começa com um número, este número é chamado de número de linha ou *rótulo*. Ele diz ao computador que queremos escrever um programa completo, e não executar cada instrução apenas. Agora digite:

```
RUN ↵
```

(**Nota**: Este comando pode ser diferente em alguns intérpretes.) Você deve ver em sua tela:

```
ALO
COMO ESTA VOCE?
>□
```



ou, dependendo de seu intérprete:

```
ALO
COMO ESTA VOCE?
END
>□
```

"Como funciona?"

Como funciona? Primeiramente nós criamos um programa de três linhas e o guardamos na memória, uma linha de cada vez, pressionando RETURN. Então, executamos o programa digitando RUN. Esta é a maneira normal de escrever e executar um programa. Nós seguiremos este procedimento de agora em diante. Toda linha deve ser precedida de um rótulo numérico e o programa será executado automaticamente na ordem dos rótulos.

Se em algum momento você digitar uma instrução sem número de linha, em resposta a um >, a instrução será executada na hora e não será memorizada. Isto é chamado de *modo imediato* ou *modo operador* em BASIC.

Se em algum momento, você digitar uma instrução BASIC rotulada em resposta a um >, a instrução será memorizada quando você bater RETURN, mas não será executada até que você digite RUN. Isto é chamado de *modo suspenso* ou *modo normal*. Vamos demonstrar como funciona: nosso programa de três linhas está armazenado na memória. Ele pode ser executado a qualquer tempo, e quantas vezes se quiser. Agora, digite:

```
RUN ↵
```

novamente e você verá:

```
ALO
COMO ESTA VOCE?
> □
```

Vamos examinar a memória do computador e mostrar seu conteúdo. Digite:

```
LIST ↵
```

e você verá em sua tela:

```
10  PRINT "ALO"
20  PRINT "COMO ESTA VOCE?"
30  END
> □
```

Seu programa está listado na tela como foi armazenado na memória do computador.

Você pode até guardar seu primeiro programa no seu cassete ou *diskette*. Para fazer isto o comando usual é:

```
SAVE ↵
```

você pode, então, tê-lo de volta, um pouco depois, digitando:

LOAD ↵

Para demonstrar a diferença entre instrução imediata e suspensa, digite:

PRINT "ADEUS" ↵

Você vê na tela:

```
> PRINT "ADEUS"
ADEUS
> □
```

Agora digite:

LIST ↵

Novamente você verá na sua tela:

```
10  PRINT "ALO"
20  PRINT "COMO ESTA VOCE?"
30  END
> □
```

Se você não tiver usado um número de linha, a instrução se perde!

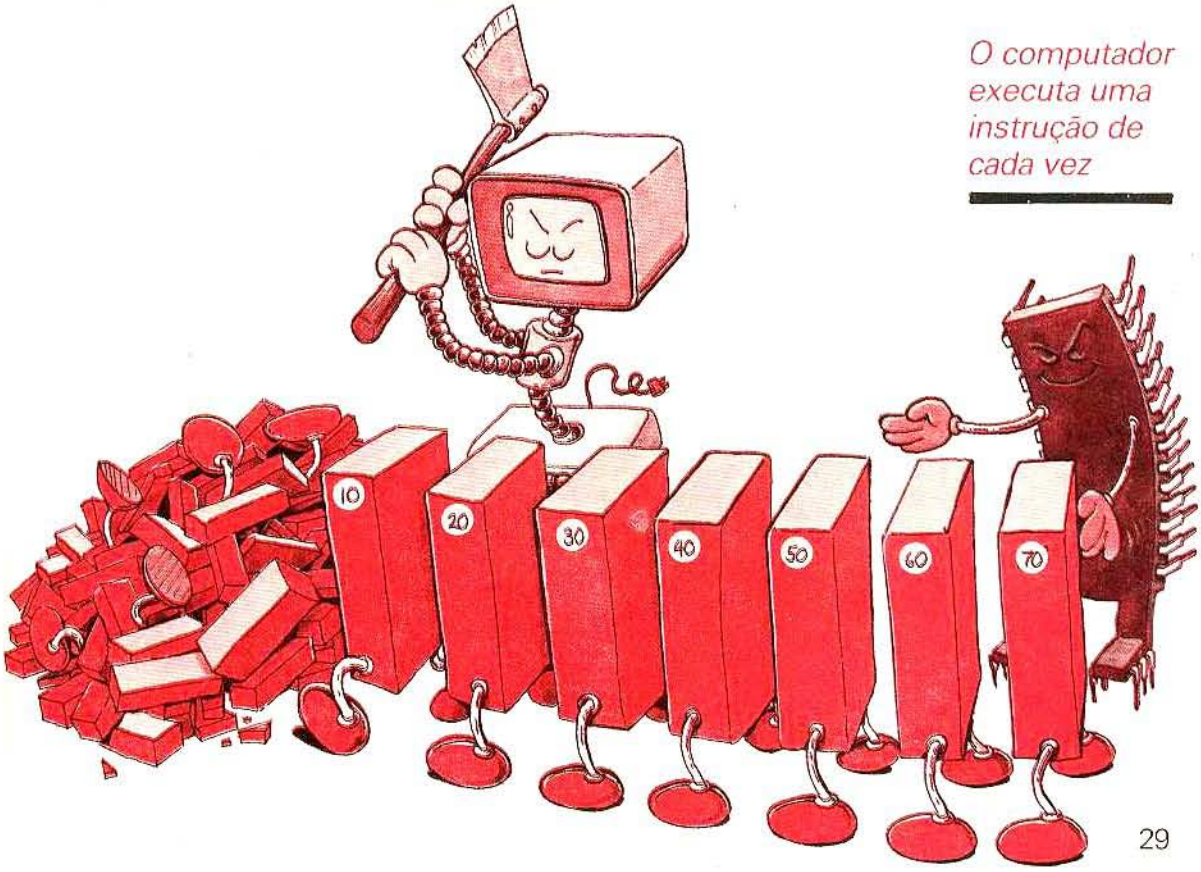
A instrução imediata: PRINT "ADEUS", não está lá para ser vista. Ela não foi memorizada.

Sumário de um Programa

Em resumo, uma instrução imediata é executada tão logo você a digita. Ela não é armazenada na memória do computador, e você precisará digitá-la sempre que quiser executá-la. Esta facilidade é chamada de modo de *execução imediata*. Na prática, esta maneira não é usada com muita freqüência — geralmente quando você quer verificar alguns valores depois de o programa ter parado. Experimente digitando instruções BASIC e veja o que acontece.

Quando uma linha é precedida de um rótulo, a maneira é chamada de: modo de *execução suspensa*. Neste modo, cada linha do programa é armazenada na memória do computador. Quando um programa completo tiver sido digitado, você simplesmente pressiona o comando RUN e o programa inteiro será executado. Lembre que, quando você desliga o computador, o que a memória RAM contém desaparece, incluindo qualquer programa que você possa ter digitado. Se você quiser guardar seu programa para usar mais tarde, você deve transferi-lo (SAVE) para um cassete ou *diskette*.

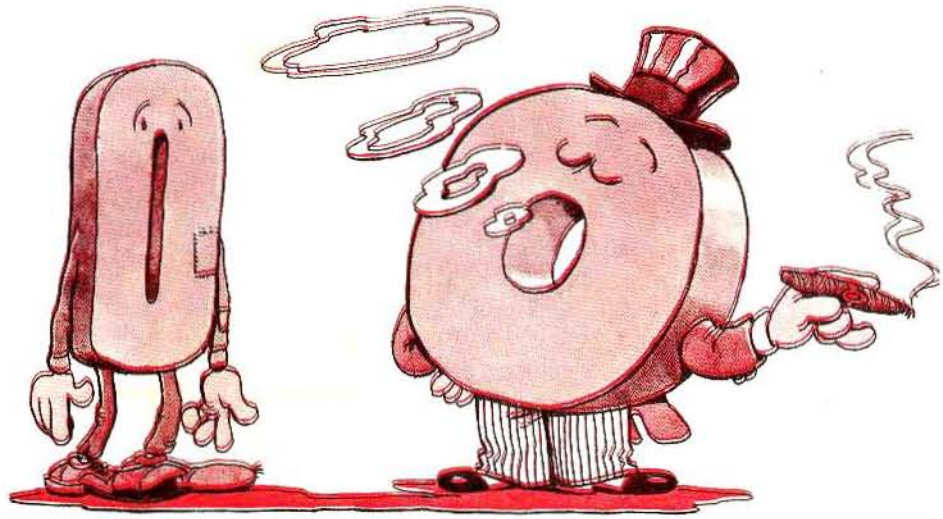
Cada linha de um programa BASIC deve começar com um rótulo. O rótulo especifica a ordem na qual a linha deve ser executada. Portanto, a linha 10 será executada antes da linha 20.



O computador executa uma instrução de cada vez

O comando END, linha 30 em nosso exemplo, é opcional nos BASIC mais modernos, mas é necessário nos mais antigos. O comando END diz ao intérprete que ele chegou a um END legítimo e não a um acidental, durante a execução.

Não confunda número 0 e a letra O



Como detalhe final, note que o dígito 0 (zero) deve parecer diferente da letra O. A maneira tradicional de evitar a confusão tem sido a de mostrar o dígito 0 (zero) como Ø — um O cortado por dentro. No entanto, hoje, o zero pode ser facilmente executado na tela como um símbolo mais fino do que aquele da letra O, e o Ø é raramente usado. Nos textos mais novos de BASIC você encontrará tanto o 0 (zero) como a letra O cortados por dentro para evitar confusão.

Um Programa mais Longo

RUN, LIST e SAVE são chamados *comandos*. Eles são palavras reservadas, usadas em si mesmas para especificar ações especiais do sistema do computador, ou mais exatamente pelo intérprete. Estes comandos são também parte do BASIC. Nós aprenderemos progressivamente outras instruções e comandos à continuidade deste livro.

Vamos agora escrever um programa mais completo, usando o PRINT e o comando NEW. Digite o seguinte:

```
NEW ↵
```

E em seguida:

```
LIST ↵
```

Nada aconteceu. Não há nenhum programa na memória. O comando NEW limpa a memória do computador. Agora digite:

```
NEW
10  PRINT "ESTE" ↵
20  PRINT "E" ↵
30  PRINT "OUTRO" ↵
40  PRINT "EXEMPLO" ↵
50  END ↵
```

Vamos descobrir o que este programa faz. Digite:

```
RUN ↵
```

Agora você verá em sua tela:

```
ESTE
E
OUTRO
EXEMPLO
> □
```

Um "END" pode aparecer ou não, dependendo do seu BASIC. Nosso programa mostra o texto como esperado. Vamos verificar se o programa foi corretamente armazenado na memória, digitando:

```
LIST ↵
```

Você agora verá em sua tela:

```
> LIST
10  PRINT "ESTE"
20  PRINT "E"
30  PRINT "OUTRO"
40  PRINT "EXEMPLO"
50  END
> □
```

O comando NEW foi utilizado para limpar a memória do computador, isto é, apagá-la e criar espaço para um novo programa. Se o comando NEW não for usado, uma instrução nova só apagará uma outra antiga se seus rótulos forem idênticos. Tal procedimento pode levar a erros, pois as instruções antigas podem ser "esquecidas" dentro do novo programa. Se você não usar NEW, toda vez que digitar uma instrução com rótulo 10, ela automaticamente apagará a instrução anterior do mesmo número de linha. Mas, *cuidado*, se você previamente escreveu uma instrução com rótulo 15 e depois digitou um programa em cima sem o NEW, a instrução "15" viria da memória do computador e seria automaticamente incorporada em seu novo programa em seqüência, (desde que o rótulo 15 não seja usado em seu programa novo). Vamos demonstrar isto. Digite:

```
15 PRINT "* * * * *" ↵
```

A seguir:

```
RUN ↵
```

Sua tela mostrará o seguinte:

```
ESTE
* * * * *
E
OUTRO
EXEMPLO
>□
```

Como você pode ver, a nova instrução PRINT rotulada 15, foi automaticamente inserida no programa antigo após a instrução 10. Vamos dar um LIST ao programa. Digite:

```
LIST ↵
```

Sua tela mostrará:

```
10 PRINT "ESTE"
15 PRINT "*****"
20 PRINT "E"
30 PRINT "OUTRO"
40 PRINT "EXEMPLO"
50 END
```

Você pode verificar que a instrução 15 é, agora, parte do seu programa. Lembre-se de que cada vez que você digitar uma instrução para o programa com um rótulo, ela é automaticamente inserida na memória do computador em sua seqüência própria.

Nós demonstraremos agora que quando você usa um número de linha que *já existe,* esta instrução nova automaticamente apagará a anterior. Vamos agora usar esta característica para apagar a instrução 15. Digite.

```
15 PRINT "....." ↵
```

Em seguida:

```
RUN ↵
```

Sua tela mostrará:

```
ESTE
.....
E
OUTRO
EXEMPLO
>□
```

Como você pode ver, sua nova afirmação PRINT com rótulo 15 superpôs-se à anterior. Verifique digitando:

```
LIST ↵
```

Sua tela mostrará o seguinte:

```
10 PRINT "ESTE"
15 PRINT "....."
20 PRINT "E"
30 PRINT "OUTRO"
40 PRINT "EXEMPLO"
50 END
>□
```

"Eis como eu apago!"

Para evitar acidentes, quando escrever um novo programa, você deve usar o comando NEW para limpar a memória de seu computador e evitar interferência de instruções "esquecidas" de programas anteriores.

Vamos agora apagar a instrução 15. Há várias maneiras de fazer isto. Aqui nós digitaremos:

```
15 ↵
```

Esta instrução consiste apenas em um rótulo. Isto é chamado de *instrução vazia*. A instrução 15 não faz nada, exceto apagar alguma versão anterior. Agora digite RUN. Você verá o seguinte em sua tela:

```
ESTE
E
OUTRO
EXEMPLO
>□
```

Seja cuidadoso. Esta característica pode ser perigosa. Se você digitar:

```
20
```

por acidente, e apertar o RETURN, você apagará a versão anterior da instrução 20 e colocará em se lugar uma instrução "vazia" que não faz nada. Para evitar surpresa, verifique sempre sua listagem de programa, antes da execução:

Sumário

Nós agora aprendemos a escrever programas BASIC elementares que mostram informações na tela. Nós escrevemos um programa BASIC usando instruções rotuladas. Nós discutimos por que os programas devem ser precedidos do comando NEW e terminados com a instrução END. Nós vimos como um programa é armazenado automaticamente na memória do computador, após introduzido e, assim, podendo ser executado, pela digitação do comando RUN. Nós vimos também como um programa pode ser mostrado com o comando LIST. Finalmente, nós usamos o comando SAVE para guardar um programa numa fita cassete ou *diskette*.

Nós aprendemos que a execução das instruções do programa é feita na ordem dos rótulos. Se você duplicar um número rotulado, seja intencionalmente ou por engano, a nova instrução apagará automaticamente qualquer instrução prévia com o mesmo número de linha. Também, se a qualquer tempo você adicionar uma linha com um novo rótulo, o intérprete a inserirá automaticamente em sua seqüência própria, dentro do programa.

Neste capítulo nós introduzimos vários conceitos novos. Se você realmente quiser aprender como programar, é essencial que comece praticando o que aprendeu. Seguem-se alguns autotestes e exercícios. Você deve fazê-los. As respostas aos exercícios selecionados são dadas no final deste livro.

Exercícios

2-1: Escreva um programa que imprima: "Tenha um bom dia".

2-2: Escreva um programa que imprima:

```
AAAAA
BBBB
CCC
DD
E
```

2-3: Escreva um programa que imprima:

```
********
*T I T U L O*
********
```

2-4: Defina os seguintes termos:

a. rótulo
b. execução programada
c. execução imediata
d. instrução vazia
e. cursor
f. tecla de controle
g. conjunto de teclas específicas
h. palavra reservada
i. símbolo

2-5: Qual é a diferença entre PRINT e LPRINT?

2-6: Você pode executar um programa inteiro digitando instruções uma de cada vez no modo imediato?

2-7: Por que usar a instrução NEW antes de digitar um novo programa?

2-8: Você pode digitar instruções de programa com a numeração fora de seqüência?

2-9: Dê exemplos de alguns comandos BASIC.

2-10: A instrução a seguir é um modo válido para mostrar a palavra EXEMPLO?

```
PRINT EXEMPLO
```

2-11: Qual é o uso da tecla RETURN?

2-12: Explique como se pode apagar a instrução 20 de um programa?

2-13: Se você já digitou a instrução 30 e deseja incluir uma nova instrução 30, deve eliminar primeiro a anterior já digitada?

2-14: Escreva um programa que mostre o seguinte:

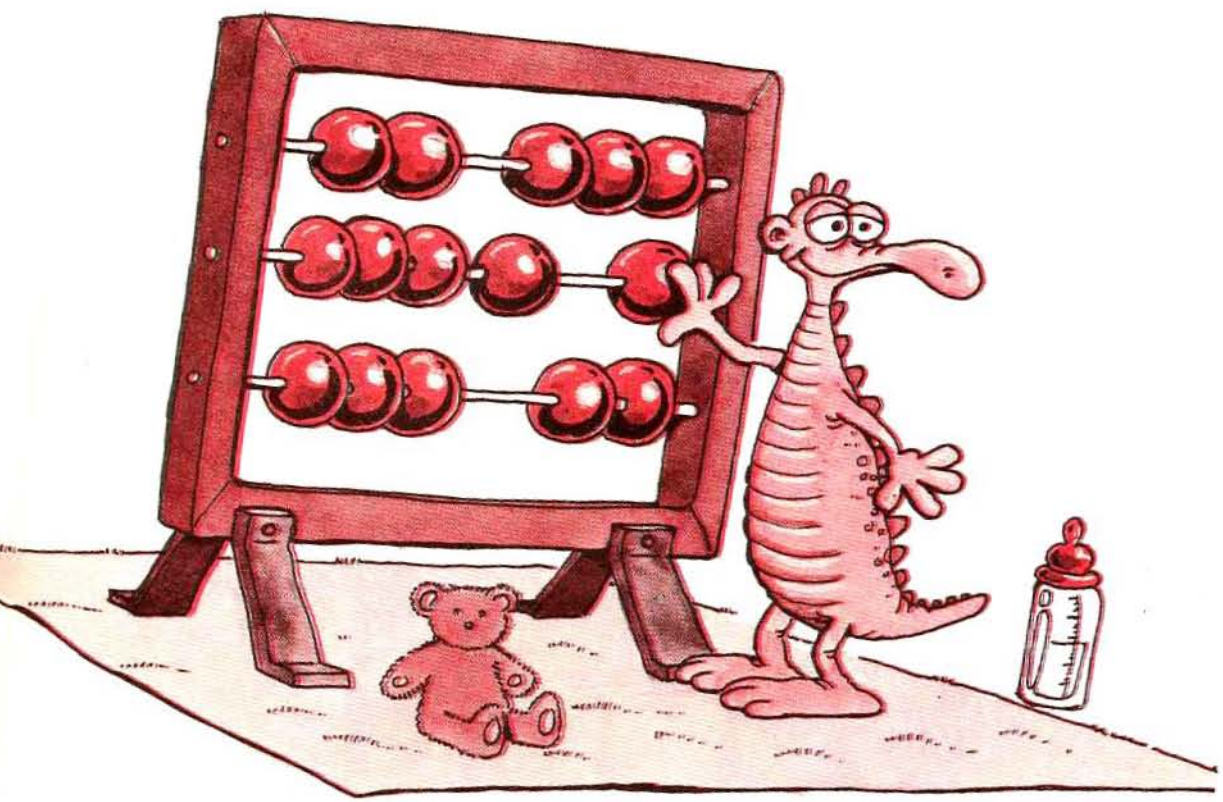
```
TTTTTT   H    H   EEEEEE
  TT     H    H   EE
  TT     H    H   EE
  TT     HHHHH    EEEE
  TT     H    H   EE
  TT     H    H   EE
  TT     H    H   EEEEEE
```

Calculando

3

Neste capítulo nós começaremos a usar números. Nós os mostraremos na adição, subtração, multiplicação e divisão. Aprenderemos a realizar cálculos, usando simples operadores aritméticos, e descreveremos outros importantes operadores existentes em BASIC.

com BASIC



Imprimindo Números

Até agora, nós imprimimos apenas textos. Vamos, então, imprimir números. Digite:

```
PRINT 3 ↵
```

O resultado deve ser:

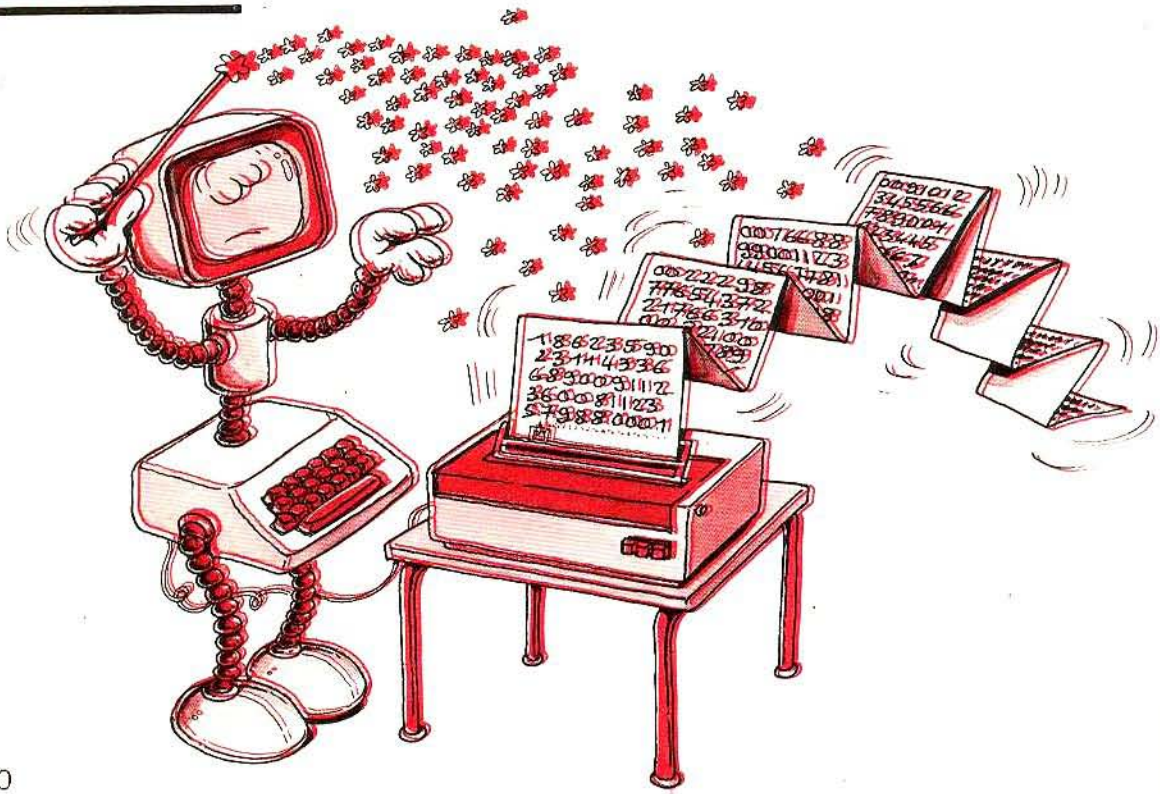
```
3
```

Lembre-se de nossa explicação no capítulo 2 de que uma instrução está no modo imediato quando uma instrução não precisa ser precedida por um rótulo e é executada imediatamente. Neste capítulo nós escreveremos todos os exemplos no modo imediato de forma que você possa executá-los pressionando apenas algumas teclas.

Note que em BASIC os números não precisam ser limitados por asteriscos ou aspas, apenas os textos: o uso de asteriscos permite que o intérprete diferencie facilmente os textos do usuário das palavras reservadas BASIC, como PRINT. O texto inserido em asterisco é chamado de *string* e uma *string* pode incluir números.

"Eu mostrarei a você outras maneiras de imprimir números."

Vamos agora tentar alguns números longos como 100, 1.000, 10.000, etc. Em algum ponto, seu intérprete BASIC recusará impri-



mir e executará uma mensagem como: "NÚMERO GRANDE DEMAIS". Cada intérprete BASIC limita o tamanho dos números inteiros que ele pode manipular, em um valor específico. Se você precisa trabalhar com números inteiros maiores que o máximo valor permitido pelo seu intérprete, você deve usar um intérprete BASIC diferente, que permita números maiores.

Recorde que o intérprete BASIC que permite o uso de números inteiros é chamado de BASIC INTEIRO. O BASIC inteiro, no entanto, não permite o uso de números decimais. Vamos descobrir se o seu intérprete BASIC permite números decimais. Digite:

```
PRINT 1.5 ↵
```

Se você vir 1.5 em sua tela, seu BASIC pode manipular números decimais. Se o seu BASIC não puder manipular números decimais, provavelmente aparecerá uma mensagem de erro ou um ? em sua tela.

No jargão dos computadores, números decimais são chamados de *números de ponto flutuante*. Um intérprete BASIC que permita o uso desses números é chamado de BASIC de *ponto flutuante*.

Notação Científica

Vamos conversar um pouco mais sobre números decimais. Assim como nos inteiros, no caso de números decimais, o intérprete reterá apenas um certo número de dígitos. Por exemplo: o valor correto de um terço é:

```
0.33333333333 ... (etc.)
```

Dentro do computador, o valor pode ser armazenado como:

```
0.3333333 (apenas sete dígitos significantes são retidos depois do zero)
```

Diz-se que o valor correto foi *truncado* (cortado) no sétimo dígito. (**Nota:** isto é uma aproximação, mas geralmente é suficiente.)

Se o seu intérprete BASIC permite números decimais, ele também usará uma *representação científica* para estes números. Quando um número é muito grande ou muito pequeno, ele será mostrado na notação científica para ganhar espaço. Eis aqui um exemplo:

```
3.2E6
```

Significa:

$$3.2 \times 10^6 = 3200000$$

10^6 significa 10 na potência 6, isto é, 10 multiplicado 6 vezes por ele mesmo:

$10 \times 10 \times 10 \times 10 \times 10 \times 10 = 1.000.000$

Similarmente:

1.12E − 7

Significa:

$1.12 \times 10^{-7} = 0.000000112$

10^{-7} significa 1/10 na potência 7, isto é, 1/10 multiplicado 7 vezes por ele mesmo (1/10 é 10^{-1}).

Usando a Aritmética

Vamos agora realizar cálculos aritméticos simples. Digite:

PRINT 2 + 2 ↵

O resultado que aparece é:

```
4
```

Nós realizamos nosso primeiro cálculo aritmético. O símbolo da adição + é chamado um *operador*. Um operador é um símbolo que representa uma operação a ser realizada sobre um ou mais operandos. O BASIC provê no mínimo cinco operadores aritméticos:

− *(menos)*

\+ *(mais)*

* *(multiplicação)*

/ *(divisão)*

^ ou ** *(exponenciação) (^ é muitas vezes mostrado como* ↑ ou [ ).

Agora tente este exemplo. Digite:

PRINT 2 * 3 ↵

O resultado deve ser 6. O símbolo * é o símbolo da multiplicação. O símbolo usual da multiplicação, ×; poderia ser confundido com a letra X, por isso as linguagens de programação usam o símbolo *.

Aqui estão outros exemplos de instruções aritméticas válidas:

```
PRINT 1 + 2 * 3
```

O resultado é:

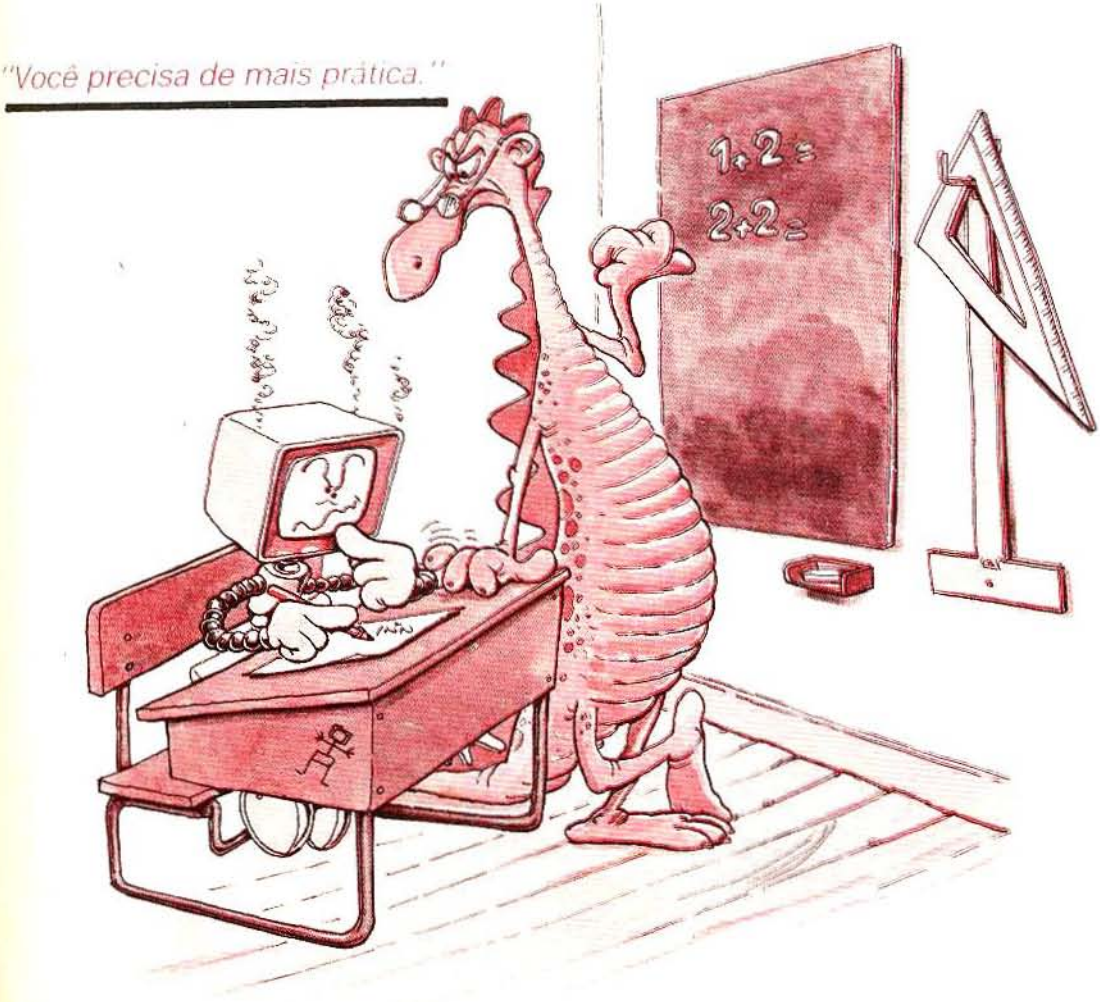
```
7
```

```
PRINT 3 - 2
```

O resultado é:

```
1
```

```
PRINT 8/2
```



O resultado é:

```
4
```

```
PRINT 1 + 2 + 3 + 4
```

O resultado é:

```
10
```

Se o seu BASIC permite números decimais, as seguintes instruções também são viáveis:

```
PRINT (6/3 + 12/4)/2
```

O resultado é:

```
2.5
```

Note que parênteses foram usados neste exemplo para clareza do grupamento das operações. Digite:

```
PRINT 2 + 3 + 4/2
```

O resultado é:

```
7
```

A divisão (/) foi realizada primeiro. Isto se deve ao fato de que em BASIC se é dada uma escolha (isto é, se não usarmos parênteses), a divisão (/) ou a multiplicação (*) ocorrerão antes da adição (+) ou da subtração (−). Se você pretendesse dividir o grupo 2 + 3 + 4 por 2, então deveria necessariamente digitar:

```
PRINT (2 + 3 + 4)/2)
```

O resultado seria então:

```
4.5
```

A divisão então seria realizada sobre o grupo (2 + 3 + 4). É uma boa prática usar livremente os parênteses, para evitar qualquer confusão. Por exemplo, a seguinte expressão (ou grupo de valores e operações)

$$\frac{1 + 2 + 3}{4 + 5} \times 3$$

poderia ser traduzida na seguinte expressão BASIC:

```
((1 + 2 + 3)/(4 + 5))* 3
```

ou

```
(1 + 2 + 3)/(4 + 5)* 3
```

Pelo fato de a execução ocorrer da esquerda para a direita, quando os operadores têm a mesma prioridade, a divisão ocorrerá, aqui, antes da multiplicação.

Se você tivesse que escrever o seguinte em BASIC:

```
(1 + 2 + 3)/(4 + 5)* 3
```

seria equivalente a:

$$\frac{1 + 2 + 3}{(4+5) \times 3}$$

Usar parêntese determina grupos. Esteja certo de colocar corretamente sempre um parêntese na direita, quando houver outro na esquerda.

Vamos agora usar nossa nova experiência em computação para mostrar valores inteiros.

Formatos de Impressão

Se você digitar:

```
PRINT "O PRODUTO DE 2 POR 3 E", 2 * 3 ↵
```

O resultado será:

```
O PRODUTO DE 2 POR 3 E 6
```

Na instrução PRINT acima nós misturamos texto e números, separados por uma vírgula. Mais precisamente usamos uma *expressão*, 2 * 3, preferivelmente a um número. Agora digite:

```
PRINT "O PRODUTO DE 2 POR 3 E", 2 * 3 ↵
```

e você obterá:

```
O PRODUTO DE DOIS POR TRES E, 2 * 3
```

Esta é uma instrução BASIC válida, mas não aquela que você pretendia. Lembre-se de que qualquer coisa entre aspas é mostrada literalmente. A vírgula ou ponto e vírgula devem estar *fora das aspas* para funcionar corretamente.

O comando PRINT pode ser usado para imprimir vários itens na mesma linha. Os itens, entretanto, devem ser separados por um ponto e vírgula ou vírgula. O ponto e vírgula resultará num pequeno espaço entre os itens que foram impressos, enquanto a vírgula resultará num espaço mais longo. Como um tabulador numa máquina de escrever, o símbolo vírgula é usado para criar *tabulações*, isto é, campos na tela. Esta técnica é conveniente para mostrar tabelas.

Vamos testar esta nova característica. Digite:

```
PRINT 1;2;3 ↵
```

Sua tela mostrará:

```
1 2 3
```

Agora digite:

```
PRINT 1,2,3, ↵
```

Só coloque números razoáveis em seu programa

Sua tela mostrará:

```
 1  2  3
```

Vamos agora calcular o percentual de imposto sobre uma venda de Cr$ 1234. O percentual de imposto é de 6.5%. Assumiremos que o seu BASIC permite números decimais. A instrução é:

```
PRINT "O IMPOSTO SOBRE AS VENDAS E"; 1234 * 6.5/100 ↵
```

O resultado é:

```
O IMPOSTO SOBRE AS VENDAS E 80.21
```

Nós também podíamos digitar:

```
PRINT "O IMPOSTO SOBRE AS VENDAS E"; 1234 * 0,065 ↵
```

e obteríamos o mesmo resultado. Há várias maneiras equivalentes de escrever um programa.

Você pode imprimir vários itens numa linha. Olhe:

```
PRINT 1;2;3;4;5;6;7;8;9; "VARIOS ITENS" ↵
```

A tela mostrará:

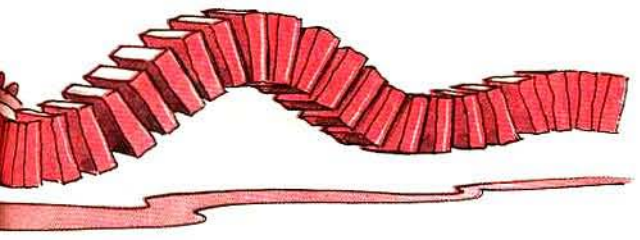
```
1 2 3 4 5 6 7 8 9 VARIOS ITENS
```

Nós aprendemos até aqui a realizar cálculos aritméticos simples e a mostrar os resultados. Vamos usar esta nova experiência para resolver alguns problemas simples.

Exemplos de Aplicação

Vamos calcular o consumo de milhas por galão de um carro. A fórmula matemática é:

CONSUMO EM MILHAS = DISTANCIA (em milhas) ÷ GASOLINA (em galões)



Vamos supor que a distância percorrida foi de 510 milhas e a quantidade de gasolina utilizada tenha sido de 20.2 galões. Aqui está a instrução escrita em BASIC:

```
PRINT "O CONSUMO E"; 510/20.2 ; "MPG" ↵
```

Para os leitores que trabalham com o sistema métrico nós converteremos o cálculo em litros por quilômetros. Um galão vale 3.8 litros. Uma milha equivale a 1.6 quilômetros. O consumo em litros por quilômetros é:

```
PRINT "O CONSUMO EM GASOLINA E"; (20.2 * 3.8)/(510 * 1.6); "L
      POR KM" ↵
```

Aqui está um outro problema simples. Dada a temperatura em graus Fahrenheit, o equivalente em Celsius é calculado pela fórmula:

valor CELSIUS = (valor FAHRENHEIT − 32) × 5/9

Para calcular o equivalente Celsius de 79°F, digite:

```
PRINT "79 GRAUS F ="; (79 - 32) * 5/9; "GRAUS C" ↵
```

O resultado é:

```
79 GRAUS F = 26.1111111 GRAUS C
```

Para completar, note que 5/9 não foi incluído no parêntese, o que não importa, desde que a * ou a / seja executada primeiro.

Sumário

Neste capítulo aprendemos a realizar cálculos aritméticos e a mostrar textos e resultados na mesma linha. Nós usamos esta nova ex-

periência para automatizar o cálculo de fórmulas simples escrevendo uma instrução BASIC de uma linha.

Até aqui especificamos todos os valores dentro da instrução BASIC em si. O que queremos fazer agora é escrever um programa e então introduzir valores, repetidamente, pelo teclado, de forma que valores diferentes possam ser usados num programa, sem ter que reescrevê-lo. Vamos realizar isto com *variáveis*. Este é o tópico do próximo capítulo.

Exercícios

3-1: Escreva uma instrução em BASIC que calcule:

$$\frac{5+6}{1+2\div 3}$$

3-2: Escreva uma instrução em BASIC que calcule:

$$1+1/2\ \frac{1}{1+1/2}$$

3-3: Escreva uma instrução em BASIC que calcule o equivalente em Fahrenheit de 20°C.

3-4: Dada uma velocidade de 100 Km/hora, calcular a velocidade equivalente em m.p.h. (1 milha = 1.6Km)

3-5: Calcule o número de segundos em um dia, uma semana, um mês e um ano.

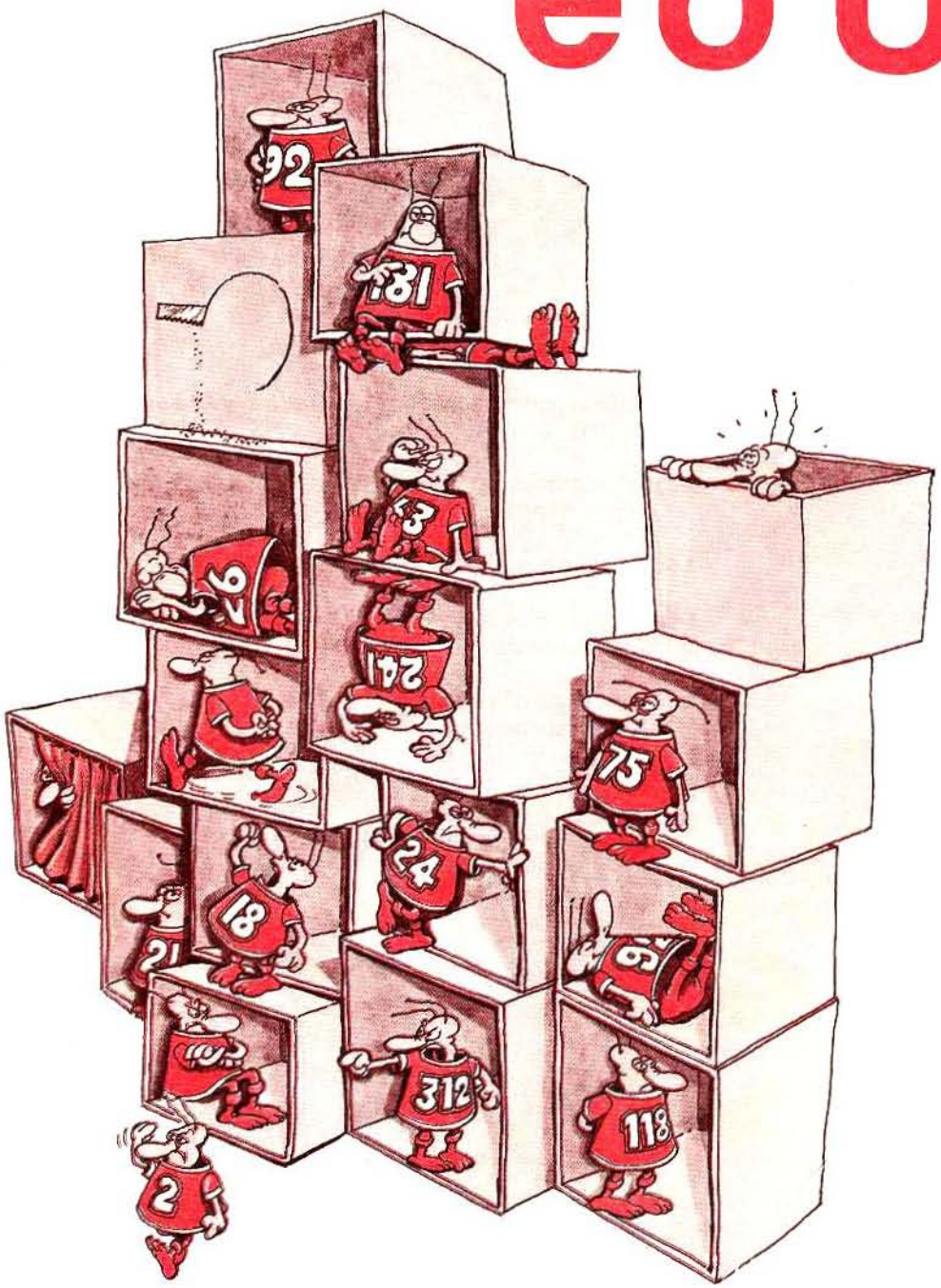
3-6: Assumindo uma velocidade média de 55 m.p.h., calcule o tempo necessário para viajar 350 milhas.

3-7: Assumindo que um ano tenha 365 dias, calcule o número de dias que você já viveu até hoje.

3-8: Calcule o salário equivalente anual de uma pessoa sabendo-se que:

a) pagamento semanal (existem 52 semanas em um ano)
b) pagamento bissemanal (multiplique por 26)
c) pagamento mensal
d) pagamento horário (multiplique por 2080).

A Memorizaç
e o Us



...o de Valores
...o de Variáveis

4

Neste capítulo aprenderemos a escrever programas que possam ser utilizados, repetidamente, sem mudanças e que mostrem resultados diferentes, dependendo dos dados fornecidos através do teclado. Até aqui, para obter resultados de um cálculo aritmético, como 2 + 3, nós incluímos na instrução do programa os números a serem operados. Agora aprenderemos a escrever programas que possam ser executados muitas vezes, com dados diferentes. O programa especificará as operações a serem realizadas. Os dados entrarão pelo teclado digitados pelo usuário no momento em que o programa é executado. Isto torna o programa reutilizável.

Além disso, introduziremos o conceito de variável e aprenderemos a usar duas novas instruções: INPUT e LET.

Vamos começar agora aprendendo a fornecer informações ao programa, enquanto ele está sendo executado.

Digite o seguinte programa: (**Nota**: Nós não utilizaremos mais ↵ para indicar RETURN no final de cada linha):

```
10  INPUT A
20  PRINT A; 2 * A; 3 * A
30  END
```

Agora execute esse programa, digitando RUN. Você verá em sua tela algo como:

```
?
```

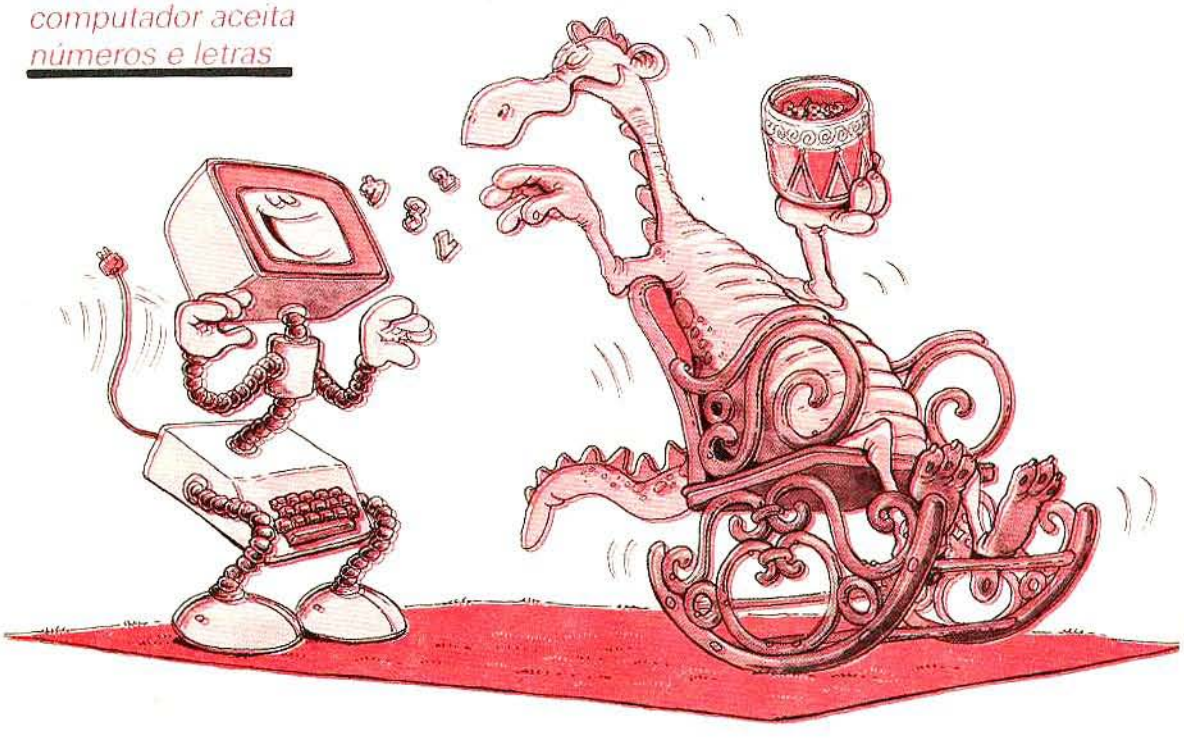
Geralmente um "?" aparece em sua tela, com o cursor piscando perto dele, para fazê-lo lembrar-se da digitação de um dado necessário. Agora digite um número, por exemplo, 3; complete a entrada do dado pressionando a tecla RETURN. Sua tela mostrará agora:

```
3 6 9
>
```

Seu programa foi executado. Vamos ver o que aconteceu. A primeira linha foi:

Com INPUT, seu computador aceita números e letras

```
10  INPUT A
```



Esta instrução pediu a você para fornecer um número pelo teclado. O programa mostrou um ? e parou — esperando por sua entrada (INPUT). O valor 3 que você forneceu foi, em seguida, lido em substituição ao A. "A" é chamado de: uma *variável*. Este é o nome usado para armazenar um valor. Formalmente uma *variável* é o nome dado a uma locação da memória. Exemplos de alguns nomes de variáveis são: A, B, C, F, Z1, G2. A maioria dos BASIC permite nomes variáveis com várias letras, como por exemplo: NUMBER 1, SOMA, TAXA, RESULTADO.

A segunda instrução foi:

```
20  PRINT A; 2 * A; 3 * A
```

Esta instrução resultou na impressão de 3,2 * 3,3 * 3 ou:

```
3 6 9
```

Usando a instrução INPUT, é ainda possível entrar com diversos valores ao mesmo tempo. Aqui está um exemplo. Digite:

```
10  INPUT A, B
20  PRINT A; A * 2; B; B * 2
30  END
```

Agora digite RUN. Você verá o "?" aparecer em sua tela. Digite dois números, 2 e 3, separados por uma vírgula e, em seguida, pressione RETURN. Você verá em sua tela:

```
2 4 3 6
```

Vamos examinar o que aconteceu. A primeira entrada do programa foi:

```
10  INPUT A, B
```

Esta instrução resultou na solicitação de dois valores a serem fornecidos por você, os quais serão armazenados nas variáveis A e B. Lembre-se de que variáveis são nomes de locação de memória. A segunda instrução do programa foi:

```
20  PRINT A; A * 2; B; B * 2
```

Esta instrução resultou na impressão dos valores: 2, 2 * 2, 3, 3 * 2, ou:

```
2 4 3 6
```

Vamos tentar executar este programa novamente. Desta vez tente digitar:

```
5,8
```

e o resultado será:

```
5 10 8 16
```

Nós podemos usar este programa repetidamente e obter novos resultados, digitando novos valores no teclado. Nós fizemos nosso programa reutilizável, usando nomes variáveis (A e B) no lugar de valores explícitos.

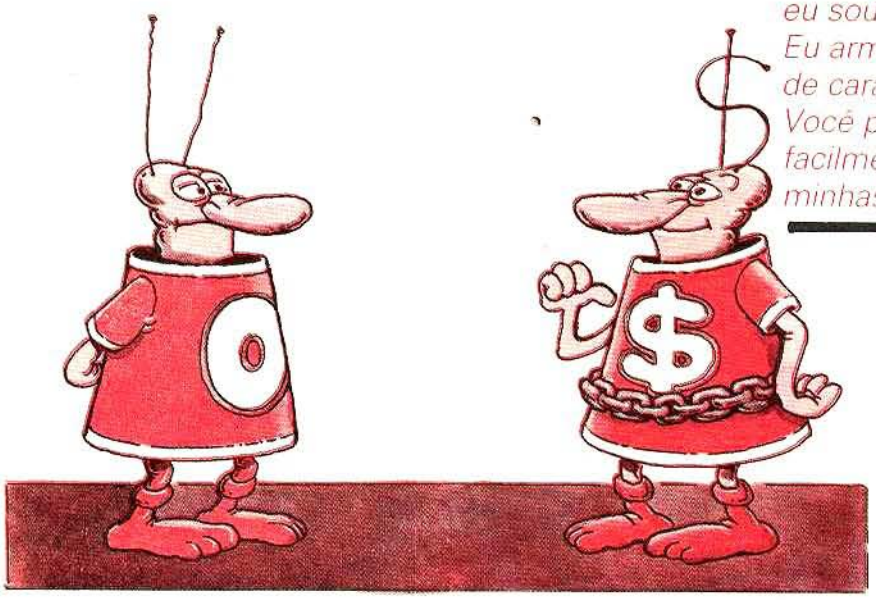
A fim de tornar este programa realmente reutilizável, vamos agora melhorar seu estilo e sua legibilidade. Suponha que queiramos guardar o programa e usá-lo RUN novamente daqui há alguns dias. Nós podemos esquecer o que ele faz ou podemos não lembrar quantos valores devem ser fornecidos. Podemos reescrever o programa para mostrar esta informação na tela. Aqui está uma versão aperfeiçoada:

```
10  PRINT "* ESTE PROGRAMA MULTIPLICA QUAISQUER DOIS
          NUMEROS POR 2 *"
20  PRINT "DIGITE DOIS NUMEROS"
30  INPUT A, B
40  PRINT "PRIMEIRO NUMERO:"; A, "DOBRO:"; 2 * A
50  PRINT "SEGUNDO NUMERO:"; B, "DOBRO:"; 2 * B
60  END
```

e aqui está o resultado: (**Nota**: através do texto, nós mostraremos os dados fornecidos pelo usuário em **negrito**):

```
* ESTE PROGRAMA MULTIPLICA QUAISQUER DOIS NUMEROS
        POR 2 *
DIGITE DOIS NUMEROS
?5,7
PRIMEIRO NUMERO:  5    DOBRO: 10
SEGUNDO NUMERO: 7    DOBRO: 14
```

Nós aprendemos até aqui a fornecer dados numéricos ao programa usando a instrução INPUT. Também introduzimos o conceito de variável. Vamos agora aprender a usar estas técnicas efetivamente e a desenvolver programas mais complexos.



Os Dois Tipos de Variáveis

Há dois *tipos* de variáveis em BASIC: *numérica* e *string**. Variáveis numéricas representam números; variáveis *string* representam texto. Estes dois tipos de variáveis são diferentes; a variável *string* tem um "$" no final do nome. Ela também é usada de forma diferente. Por exemplo, você pode somar números, mas não textos. Vamos aprender primeiro sobre as variáveis numéricas, depois sobre as variáveis *string*.

Variáveis Numéricas

Vamos aprender as regras para nomear uma variável numérica; depois aprenderemos a usar estas variáveis efetivamente. Nós estamos prontos para usar duas variáveis numéricas chamadas A e B, desde o começo desse capítulo. Nós demos a elas um valor entrando com números pelo teclado. Vamos aprender agora a dar nomes a uma variável.

Quando nomeamos uma variável, todos os BASICs incluindo o BASICs original de Dartmouth, permitem o uso de uma letra, opcio-

* N.T.: O termo em inglês *string* pode ser traduzido como variável alfanumérica. Entretanto, dado ao uso universal do termo *string* usaremos alternadamente ambos os termos no decorrer do livro, tentando fixá-los junto ao leitor.

nalmente seguida por um único número. Aqui estão exemplos de nomes válidos de variáveis:

A *(uma letra)*
B *(uma letra)*
Z *(uma letra)*
A1 *(uma letra e um dígito)*
A2 *(uma letra e um dígito)*
B2 *(uma letra e um dígito)*

Com essa definição, os seguintes nomes não são legais:

12 *(não começa com uma letra)*
A2B *(muitos caracteres)*
BA *(apenas uma letra é permitida)*
1B *(deve começar com uma letra)*
AB1 *(muito longo — apenas uma letra é permitida)*

Toda variável deve ter um nome

A vantagem dos nomes curtos é reduzir o tamanho e a complexidade do intérprete BASIC. A desvantagem desses nomes curtos é a dificuldade de lembrar. Por exemplo, o longo nome RESULTADO é

mais descritivo e memorizável do que o curto R. Para melhorar a legibilidade, a maioria dos novos BASIC permite nomes longos, isto é, uma seqüência de caracteres. Usualmente, você pode usar qualquer número de letras consecutivas, seguidas de dígitos opcionais; até um comprimento máximo. Por exemplo: os seguintes nomes são as variáveis legais mais longas.

VENCEDOR	ESTUDANTE1
PERDEDOR	ESTUDANTE2
RESULTADO	ESTUDANTE14
SOMA	CASO24

Os seguintes não são legais:

R2D2	*(letras não são consecutivas)*
3TIMES	*(começa com um dígito)*
A-ONE	*(símbolo ilegal)*

É claro que um programa com nomes de variáveis longos e explícitos é mais legível. Nesse capítulo usaremos ambos, nomes curtos e longos, para que você possa usar ambas as convenções. Lembre-se de que nomes longos são simplesmente uma questão de conveniência e não afetam em nada o programa.

Há ainda mais uma restrição a nomes: você não pode usar um nome que seja uma *palavra reservada*, isto é, um nome que tenha um significado para o seu intérprete BASIC. Por exemplo, você não pode usar LIST, END ou RUN como nomes de variáveis. (**Nota**: a lista de palavras reservadas comuns aparece no final desse livro, assim como no final do manual de referência do seu intérprete BASIC). Alguns BASICs estão ainda proibidos de usar uma palavra reservada, mesmo como *parte* de um nome. Por exemplo, OR é uma palavra reservada padrão — por esta razão, em alguns BASICs, você não pode usar PORT como um nome de variável.

Agora que nós sabemos como criar nomes legais para variáveis numéricas, vamos aprender a dar um nome para um pedaço de texto conhecido como *string* (cadeia de caracteres).

"Eu sou uma variável string. Eu contenho texto. Meu nome termina com um $."

Variáveis *String* (variáveis alfanuméricas)

Primeiro vamos conhecer uma cadeia de caracteres (*string*). Eis um exemplo de cadeias:

"RESULTADO"
"ESTE E UM EXEMPLO"
"MEU NOME E JOAO"
"25 VEZES 4 ="

Note que uma cadeia de caracteres está normalmente inserida entre aspas; de outra maneira ela poderia ser confundida com um nome de variável. Uma cadeia deve conter uma seqüência de caracteres, exceto o símbolo de aspas (cada versão de BASIC tem, no entanto, uma maneira de contornar esta limitação). O comprimento de uma cadeia de caracteres é sempre limitado, o limite típico deve ser 128 caracteres.

Vamos conhecer agora as variáveis *string*. Quando o valor armazenado em uma variável é um texto (isto é, uma cadeia de caracteres), ao invés de um número, a variável é chamada de *variável string*.

O nome de uma variável *string* é como um nome de uma variável numérica, exceto pelo fato de que ela deve ter um "$" no final. Aqui estão alguns nomes curtos válidos para variáveis *string*.

```
A$
R$
A1$
B5$
```

Se o seu BASIC permite nomes longos, aqui estão vários nomes válidos:

```
NOME$
PRIMEIRO$
CIDADE$
UNIDADE25$
```

Lembre-se de que um nome como BRAND$ pode ser ilegal para alguns BASIC que contêm a palavra reservada AND. Mas não se preocupe, porque o intérprete provavelmente lhe dirá isto, tão logo você o digite.

Vamos ilustrar o uso de variáveis *string*, com um programa que o cumprimente pelo nome:

```
10  PRINT "EU SOU HAL, O COMPUTADOR"
20  PRINT "QUAL E SEU PRIMEIRO NOME";
30  INPUT PRIMEIRO$
40  PRINT "E QUAL E SEU ULTIMO NOME";
50  INPUT ULTIMO$
60  PRINT "ALO,"; PRIMEIRO$;" "; ULTIMO$;"!"
70  PRINT "AGORA EU SEI SEU NOME!"
80  PRINT "EU GOSTO DE"; PRIMEIRO$; "COMO UM PRIMEIRO
        NOME."
90  END
```

Se o seu BASIC exige nomes curtos, use P$ no lugar de PRIMEIRO$ e U$ no lugar de ULTIMO$. Aqui está uma amostra. Note que os caracteres que você digita aparecem aqui em negrito:

```
> RUN
EU SOU HAL, O COMPUTADOR.
QUAL E SEU PRIMEIRO NOME? JOAO
E QUAL E SEU ULTIMO NOME? SOUZA
ALO, JOAO SOUZA!
AGORA EU SEI SEU PRIMEIRO NOME!
EU GOSTO DE JOAO COMO PRIMEIRO NOME.
>
```

Você pode se comunicar com seu computador! Vamos anotar algumas das características básicas desse programa. Vamos começar com a linha 20:

```
20  PRINT "QUAL E SEU PRIMEIRO NOME";
```

Note que essa linha termina com um ponto e vírgula. O ponto e vírgula significa "mostre o caractere seguinte imediatamente após esse texto". O resultado da instrução 20 e sua resposta na instrução 30 é:

```
QUAL E SEU PRIMEIRO NOME? JOAO
```

Se você tivesse escrito:

```
20  PRINT " QUAL E SEU PRIMEIRO NOME"
```

com o ponto e vírgula ao final, o resultado teria sido:

```
QUAL E SEU PRIMEIRO NOME
? JOAO
```

Naturalmente que você pode escolher uma ou outra maneira. O formato é uma questão de preferência. Mas lembre-se, caso você queira que os caracteres digitados apareçam na mesma linha da mensagem anterior, use o ponto e vírgula no final da instrução PRINT. Caso contrário, a posição seguinte na tela será o começo da linha seguinte.

Agora nós já sabemos mais sobre as variáveis numéricas e *string*; vamos usá-las na continuidade do diálogo com HAL, o computador.

Vamos acrescentar o seguinte em nosso programa:

```
90  PRINT "QUAL E O ANO CORRENTE (2 DIGITOS)";
100 INPUT ANO CORRENTE
110 PRINT "EM QUE ANO VOCE NASCEU (2 DIGITOS)";
120 INPUT ANO DO NASCIMENTO
130 PRINT "QUERIDO"; PRIMEIRO$; "; NESTE ANO VOCE TEM OU
        TERA"; ANO CORRENTE - ANO DO NASCIMENTO
140 END
```

Aqui está um diálogo de amostra. Novamente os caracteres que você digitou são mostrados em negrito:

```
QUAL E O ANO CORRENTE (2 DIGITOS)? 84
EM QUE ANO VOCE NASCEU (2 DIGITOS)? 50
QUERIDO JOAO, ESTE ANO VOCE TEM OU TERA 34
```

Vamos examinar o programa em detalhes. A instrução

```
90  PRINT "QUAL E O ANO CORRENTE (2 DIGITOS)";
```

imprime uma mensagem óbvia. Novamente o ponto e vírgula é usado e, portanto, os dois dígitos foram executados na mesma linha. Na afirmação seguinte:

```
100 INPUT ANO CORRENTE
```

ANO CORRENTE é uma variável numérica. O valor 84 que você digitou será lido em seu lugar. Daqui para frente, sempre que o nome ANO CORRENTE for usado, o valor 84 o substituirá automaticamente através do intérprete BASIC. Isto acontecerá na instrução 130.

A instrução

```
110 PRINT "EM QUE ANO VOCE NASCEU (2 DIGITOS)";
```

é idêntica a instrução 90. Note o ponto e vírgula no final. A instrução

```
120 INPUT ANO DE NASCIMENTO
```

é idêntica a instrução 100. ANO DE NASCIMENTO é uma nova variável numérica. Ela contém o valor 50. Na instrução seguinte (130) nós usamos este nome de variável. Note que o valor 50 é automaticamente substituído pelo intérprete. Vamos examinar:

```
130 PRINT "QUERIDO"; PRIMEIRO$; ", ESTE ANO VOCE TEM OU
        TERA"; ANO CORRENTE - ANO DE NASCIMENTO
```

Quando essa instrução é executada ocorre o seguinte:

```
QUERIDO JOAO, VOCE TEM OU TERA 34.
```

Vamos separar em partes esse resultado e examiná-lo:

QUERIDO

(esta é chamada uma constante alfanumérica. Uma constante alfanumérica é uma constante que é parte de uma instrução do programa)

JOAO

(este é o valor da string *que é digitada no teclado e armazenada na variável alfanumérica chamada PRIMEIRO$. Ele permanecerá lá desde que você não use o comando NEW ou não entre com um novo valor em PRIMEIRO$)*

ESTE ANO VOCE TEM OU TERA

(esta é outra constante alfanumérica)

34

(este é o resultado do ANO CORRENTE – ANO DE NASCIMENTO, isto é, 84 – 50 = 34)

Digitando e examinando esse programa você pode, de qualquer forma, ficar com algumas frustrações. Por exemplo, você pode querer introduzir dados atuais, incluindo dia e mês para poder calcular sua idade hoje. No entanto, isto exigirá a comparação do dia e mês de hoje com a data de seu aniversário. Para fazer isto, nós devemos aprender uma nova instrução BASIC:

```
IF (condição) THEN (ação)
```

Nós discutiremos esta instrução em detalhes no capítulo 7. Uma outra coisa que você pode querer fazer é escrever o programa para que se execute repetidamente (assim você não precisa digitar RUN a cada vez). Aprenderemos a fazer isto no capítulo 6 quando iremos discutir a instrução GOTO.

Agora que estamos familiarizados com as variáveis numéricas e as *strings*, vamos aprender a utilizá-las em um programa longo — primeiro designando um valor para uma variável, depois usando a técnica do contador.

Designando um Valor para uma Variável *(A Instrução LET)*

Até aqui, a única maneira de atribuir um valor a uma variável foi com a instrução INPUT. Por exemplo, quando a seguinte instrução é executada:

```
20  INPUT A
```

Você digita um valor como 5.2 (segundo de) e o valor de A é então 5.2.

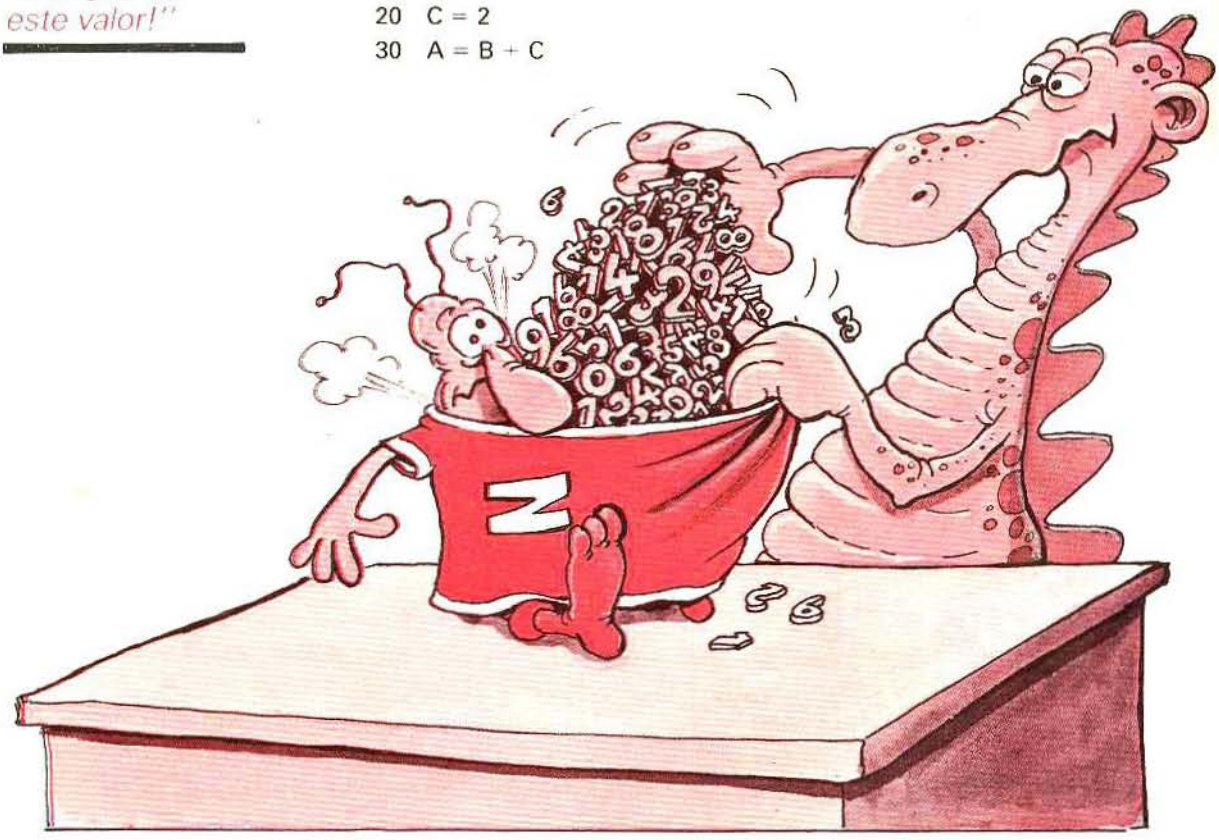
Há, no entanto, outra maneira de atribuir um valor para A. É chamado de *instrução designativa*. Aqui está um exemplo:

```
10  A = 5.2
```

(**Nota**: use 5 em vez de 5.2 se você tem um BASIC de números inteiros). Esta instrução atribui o valor 5.2 para A como parte do programa, assim você não tem que fornecê-lo pelo teclado. Você pode ainda escrever:

"Z, eu quero que você guarde este valor!"

```
10  B = 1
20  C = 2
30  A = B + C
```



Como você pode ver, o valor de A será 2 + 1 = 3, quando a instrução 30 for executada.

Nos BASICs mais novos a instrução designativa deve começar com a palavra reservada LET. Nestes BASICs este exemplo poderia ser escrito:

```
10  LET B = 1
20  LET C = 2
30  LET A = B + C
```

O propósito da instrução LET foi simplificar o projeto do intérprete dizendo explicitamente que a instrução é uma instrução designativa. A instrução LET, algumas vezes, não é usada. Eliminando-a, eliminamos um passo para o programador.

Vamos agora mostrar o valor da instrução de designação. Nós examinaremos dois exemplos. Em ambos, nós calcularemos a soma e a média de dois números. Eis o nosso primeiro programa sem a instrução designativa:

```
10  PRINT "DE ME DOIS NUMEROS"
20  PRINT "EU QUERO CALCULAR SUA SOMA E MEDIA"
30  PRINT "PRIMEIRO NUMERO POR FAVOR:";
40  INPUT A
50  PRINT "SEGUNDO NUMERO POR FAVOR:";
60  INPUT B
70  PRINT "A SOMA DE"; A; "E"; B; "E:"; A + B
80  PRINT "SUA MEDIA E:"; (A + B)/2
90  END
```

E aqui está um típico RUN:

```
> RUN
DE ME DOIS NUMEROS
EU QUERO CALCULAR SUA SOMA E MEDIA
PRIMEIRO NUMERO POR FAVOR: 24
SEGUNDO NUMERO POR FAVOR: 41
O TOTAL DE 24 E 41 E: 65
SUA MEDIA E: 32.5
>
```

Repare que a expressão A + B é repetida duas vezes na instrução PRINT. Este é um inconveniente pequeno. No entanto, num programa mais longo isto inclui a possibilidade de se digitar erros. Em suma, se você quiser mudar o programa para usar uma fórmula diferente, seria necessário reescrevê-lo.

Aqui está um programa equivalente que usa uma *variável intermediária* chamada SOMA, para armazenar o resultado. É mais fácil de ler e com menos possibilidades de erro.

```
10  PRINT " DE ME NUMEROS"
20  PRINT "QUERO CALCULAR SUA SOMA E MEDIA"
30  PRINT "PRIMEIRO NUMERO POR FAVOR:";
40  INPUT A
50  PRINT "SEGUNDO NUMERO POR FAVOR:";
60  INPUT B
70  SOMA = A + B
80  MEDIA = SOMA / 2
90  PRINT "A SOMA DOS NUMEROS ACIMA E:"; SOMA
100 PRINT "SUA MEDIA E:"; MEDIA
110 END
```

Duas novas variáveis são usadas:

```
70  SOMA = A + B
80  MEDIA = SOMA / 2
```

Usando nomes de variáveis adicionais tem-se duas vantagens: o programa é mais claro e é mais fácil para modificar. Por exemplo, vamos admitir que agora queremos mudar este programa para obter a média de três números. Para fazer isto, digitaríamos:

```
62  PRINT "TERCEIRO NUMERO POR FAVOR:";
64  INPUT C
70  SOMA = A + B + C
80  MEDIA = SOMA / 3
```

No resto do programa não haveria modificações. Nós teríamos simplesmente escrito um terceiro número, e modificado a fórmula em um único local. Aqui está o programa completo:

```
10  PRINT "DE ME NUMEROS"
20  PRINT "QUERO CALCULAR SUA SOMA E MEDIA"
30  PRINT "PRIMEIRO NUMERO POR FAVOR:";
40  INPUT A
50  PRINT "SEGUNDO NUMERO POR FAVOR:";
60  INPUT B
62  PRINT "TERCEIRO NUMERO POR FAVOR:";
64  INPUT C
70  SOMA = A + B + C
80  MEDIA = SOMA/ 3
90  PRINT "A SOMA DOS NUMEROS ACIMA E:"; SOMA
100 PRINT "SUA MEDIA E:"; MEDIA
110 END
```

E um típico RUN:

```
DE ME UM NUMERO
QUERO CALCULAR SUA SOMA E MEDIA
PRIMEIRO NUMERO POR FAVOR:? 5
SEGUNDO NUMERO POR FAVOR:? 3
TERCEIRO NUMERO POR FAVOR:? 10
A SOMA DOS NUMEROS ACIMA E: 18
SUA MEDIA E: 6
```

Agora, aprendemos dois métodos para associar um valor a uma variável:

- Podemos usar a instrução INPUT — quando um valor é suprido em tempo de RUN, isto é, enquanto o programa é executado.
- Podemos usar uma instrução designativa — em que um valor ou um método para calcular um valor (uma fórmula) é armazenado no interior do próprio programa.

O primeiro método (usando a instrução INPUT) deveria ser usado quando você espera que o valor explicitamente suprido pela variável (este não é um valor *calculado)* seja diferente cada vez que o programa é executado.

O segundo método (usando uma instrução designativa) deveria ser utilizado toda vez que você usar uma fórmula para calcular o valor da variável ou toda vez que você espera que o valor explicitado (isto é, um valor *não calculado*) permaneça o mesmo, a cada vez que o programa for executado.

Vamos agora aprender as regras completas para escrever uma instrução designativa.

A Sintaxe de uma Designação

As regras (ou sintaxe) para escrever uma instrução designativa são simples: A forma geral de uma instrução designativa é:

< variável > = <expressão>

Deve sempre haver uma variável à esquerda e uma expressão à direita. Mas precisamente, uma expressão é:

- um número ou uma variável ou
- um número ou uma variável seguidos de um operador (assim como +, −, *, /) e uma outra expressão.

Aqui estão algumas expressões de amostra:

3	*(um número)*
A	*(uma variável)*
2 + 2	*(número, operador, número)*
A + 2	*(variável, operador, número)*
A + B * 3	*(variável, operador, expressão)*

Expressões devem ser incluídas entre parênteses, por exemplo:

```
3 + (A + 2)/2
B + ((C * 2) + (D/2))/4
```

Você pode querer pensar numa expressão como um valor, ou como alguma coisa a ser calculada que resulte num valor (em outras palavras, como uma fórmula para um valor calculado).

O símbolo igual (=), usado em uma instrução designativa, não tem a mesma interpretação que tem na matemática padrão. Numa expressão ele significa "recebe o valor de". Por exemplo, você pode escrever:

```
10 A = 1
20 A = A + 1
```

(A expressão A = A + 1 poderia ser sem sentido ou um absurdo na matemática). Em BASIC, após a afirmação 20 ser executada, o valor de A será 1 (o valor prévio de A) + 1 = 2. Para evitar qualquer confusão, várias das modernas linguagens de programação usam a ← ou os símbolos: = ao invés do sinal =. Lembre-se de que em uma instrução designativa em BASIC, o sinal = indica que a variável da esquerda recebe o valor da expressão da direita. Aqui estão exemplos de instruções designativas válidas:

```
A = - 3 + 2 ( - 3 é um número inteiro negativo)
B = A + 1
C = (2 * 3) + (A / B)
MEDIA = SOMA / NUMERO DE PARCELAS
QUADRADO = A * * 2
X = B * * 2 - (4 * A * C)
```

Vamos examinar esta última designação e verificar se ela satisfaz nossa definição.

```
B * * 2
```

é

<variável> <operador> <número>

seguido de

```
- (4 * A * C)
```

isto é,

<operador> <expressão entre parênteses que contém um valor>

Dentro do parêntese

```
4 * A * C
```

é

<número> <operador> <variável> <operador> <variável>

Sim, é uma expressão válida.

No entanto, as seguintes designações não são válidas:

B + C = SOMA	*(apenas uma variável — não uma expressão — à esquerda do sinal =)*
2 = A	*(deve haver uma variável, não um valor, à esquerda)*
SOMA = B + C (D/3)	*(operador faltando depois do C)*
(MEDIA) = (B + C)/2	*(não há parênteses do lado esquerdo)*
A =	*(faltando valor à direita)*

Finalmente, note que no momento em que uma designação é executada, todas as variáveis do lado direito do sinal = devem ter recebido um valor. Se você escrever:

```
10 B = 2
20 SOMA = B + C
30 INPUT C
```

o programa falhará na linha 20 desde que C não tem um valor. Você provavelmente pretendia escrever:

```
10 B = 2
20 INPUT C
30 SOMA = B + C
```

Nós agora aprendemos a sintaxe das designações. Vamos usar esta nova experiência e introduzir uma técnica importante para o uso de designações: a *técnica de contagem*. Nós a usaremos em vários dos nossos programas.

A Técnica da Variável de Contagem

Lembre-se de que uma variável é simplesmente um nome dado para uma locação da memória. Um valor pode ser armazenado numa locação da memória pelo uso da instrução INPUT ou uma instrução designativa (=). Neste exemplo, nós queremos mudar o valor de uma variável, repetidamente, de forma a contar eventos. A técnica a ser usada para fazer isto é chamada de: *a técnica de contador de variáveis*.

Vamos agora demonstrar como designações sucessivas podem mudar o valor da variável N. Digite o seguinte no modo imediato. (**Nota:** este modo também é chamado de *modo de calculadora*.):

```
N = 1
```

O valor é automaticamente armazenado em N. Vamos verificar o que ocorre. Digite:

```
PRINT N
```

O valor 1 aparece. Agora digite:

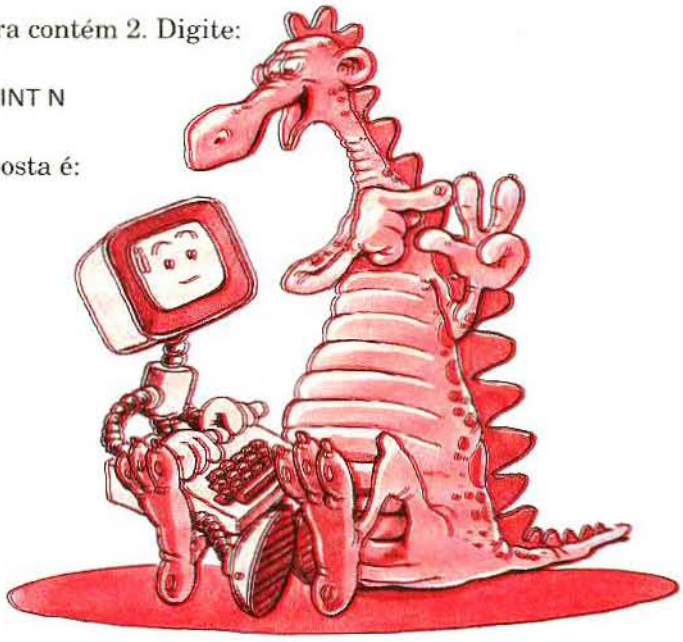
```
N = 2
```

N agora contém 2. Digite:

```
PRINT N
```

A resposta é:

```
2
```



"Vamos brincar com o contador!"

"Eu sou uma variável de contagem."

O valor 2 substituiu o valor 1 em N. Digite:

```
N = 3
```

Agora digite:

```
PRINT N
```

e verifique que o valor 3 foi armazenado em N. O mecanismo trabalha. Nós, daqui a pouco, usaremos esta técnica para contar eventos. Em outras palavras, uma variável pode ser usada como um contador de eventos. Aqui está um exemplo de um programa avançado que calcula quantas vezes você informa um número no teclado. Ele pára quando você atinge zero.

```
10  SOMA = 0
20  SOMA = SOMA + 1
30  PRINT "ENTRADA DE QUALQUER NUMERO. DIGITE ZERO PARA
    PARAR";
40  INPUT NUMERO
50  PRINT "VOCE DEU ENTRADA A"; SOMA; "NUMEROS"
60  IF NUMERO <  > ZERO THEN 20
70  END
```

Mais tarde, no capítulo 6, nós examinaremos instruções como a instrução 60 detalhadamente. A instrução significa: IF NUMERO não é igual (< >) a zero, THEN (então) execute a instrução 20. Aqui está um típico RUN:

```
ENTRADA DE QUALQUER NUMERO. DIGITE 0 PARA PARAR? 5
VOCE DEU ENTRADA A 1 NUMERO.
ENTRADA DE QUALQUER NUMERO. DIGITE 0 PARA PARAR? 1
VOCE DEU ENTRADA A 2 NUMEROS.
ENTRADA DE QUALQUER NUMERO. DIGITE 0 PARA PARAR? 2
VOCE DEU ENTRADA A 3 NUMEROS.
ENTRADA DE QUALQUER NUMERO. DIGITE 0 PARA PARAR? 3
VOCE DEU ENTRADA A 4 NUMEROS.
ENTRADA DE QUALQUER NUMERO. DIGITE 0 PARA PARAR? 4
VOCE DEU ENTRADA A 5 NUMEROS.
ENTRADA DE QUALQUER NUMERO. DIGITE 0 PARA PARAR? 0
VOCE DEU ENTRADA A 6 NUMEROS.
END
```

Neste programa o valor de SOMA é *inicializado* em 0 na primeira instrução. Ele acrescenta mais um, a cada vez que um novo número é

introduzido. Esta é uma variável de contagem. Nós veremos vários exemplos desta técnica conforme formos escrevendo mais programas. Mais tarde, aprenderemos a ajustar tal programa, caso não queiramos ter o 0 contado como número quando paramos o programa.

Sumário

Neste capítulo aprendemos a escrever programas que podem ser utilizados repetidamente. Estes programas fornecerão novos resultados, dependendo dos valores introduzidos pelo teclado. Nós conseguimos isto através do uso de variáveis e de valores designados para elas.

Uma variável deve ser entendida como um nome dado para uma locação da memória, na qual um valor ou um texto pode ser armazenado ou acumulado.

Nós aprendemos a mudar o conteúdo de uma variável usando uma instrução designativa ou um INPUT. Nós podemos agora escrever um programa simples que automatize um diálogo ou um cálculo simples.

No entanto, nossos programas agora, consistentemente, excedem dez linhas. Eles estão se tornando longos. Vamos mantê-los claros. No próximo capítulo, antes de proceder a qualquer aprendizado adicional com novos detalhes e técnicas, nós aprenderemos a escrever um programa claro.

Exercícios

4-1: Introduza quatro números pelo teclado e mostre o total, a média e o produto dos mesmos.

4-2: Vamos admitir que seu BASIC permite "variáveis com nomes longos". São válidos estes nomes de variáveis?

a. 24B	e. ALPHA2D	i. PI
b. B24	f. EXEMPLO	j. 3$
c. A + B	g. INPUT	k. TRES
d. A$LUSB	h. INPUT 1	l. NOME$

4-3: Escreva um programa que pergunte o nome do usuário e diga: "EU ACHO QUE CONHEÇO UM (o nome aqui)!"

4-4: Escreva um programa que solicite:

— o nome de um objeto
— o nome de uma peça de mobília
— o nome de um amigo

e então diga: "O SEU AMIGO (nome) TEM UM (objeto) SOBRE UMA (peça de mobília)"?

4-5: Escreva um programa que solicite a cor de seus olhos e então diga: "EU GOSTO DE OLHOS (a cor)".

4-6: As seguintes instruções são válidas?

a. A + 1 = A	d. B + C = A
b. A = A + A + A	e. 3 = 2 + 1
c. A = B + C	f. NUMERO = PRIMEIRO + ULTIMO * 2

Escrevendo u

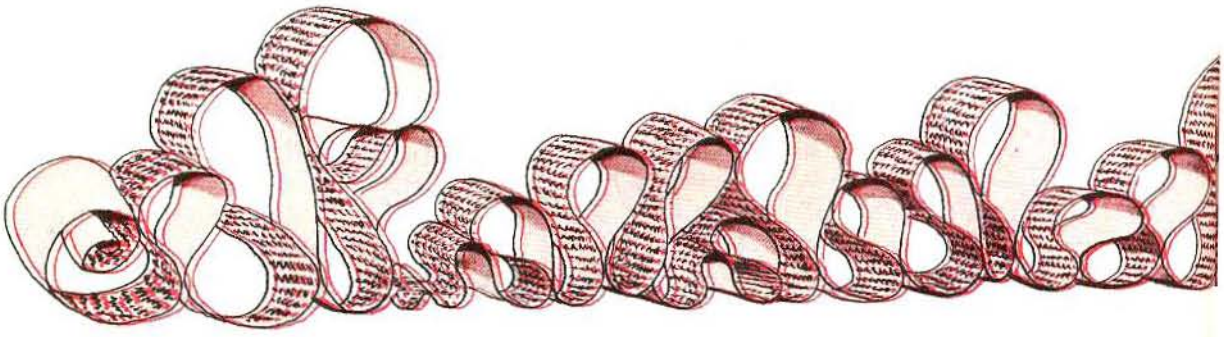
5

Até agora nós escrevemos programas curtos, usando três tipos diferentes de instrução: PRINT, INPUT e designativa (=). Nós também aprendemos a escrever expressões aritméticas simples. Nos capítulos que se seguem, aprenderemos novas técnicas e tipos de instrução, e escreveremos programas mais longos. Antes de prosseguir, no entanto, vamos aprender a fazer com que nossos programas sejam claros e legíveis.

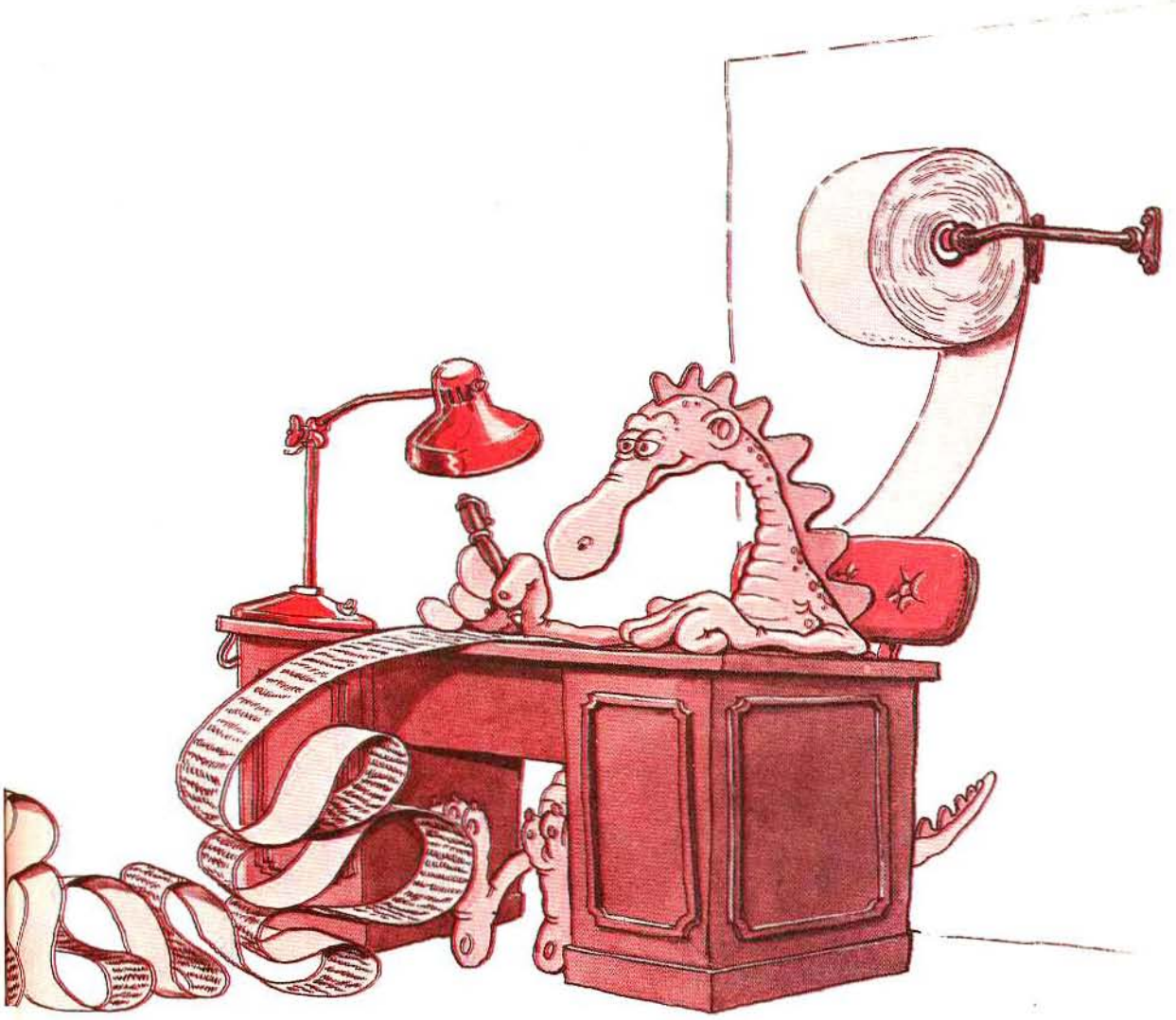
Fazer um programa legível é importante. Se você escreve um programa hoje e vai usá-lo ou mudá-lo daqui a alguns dias, levará provavelmente algum tempo para se lembrar e entender o que o programa faz e como faz. A regra é planejar antecipadamente e fazer cada programa tão legível quanto possível na hora em que o escreve. O propósito desse capítulo é ajudá-lo a melhorar a legibilidade de seu programa. Examinaremos sete técnicas:

1. O uso da instrução REM (introduz *observações* num programa)
2. O uso das múltiplas instruções em uma linha
3. O uso de espaços vazios dentro de uma instrução
4. O uso do "PRINT vazio" e da instrução CLE (para melhorar o "display" da tela)
5. O uso da instrução "INPUT reduzida" (para reduzir o número de linhas de um programa)
6. A seleção de nomes significativos para as variáveis.
7. A numeração apropriada da linha.

Agora descreveremos uma técnica de cada vez.



m Programa Claro



A Instrução REM

Aqui está um exemplo do uso da instrução REM:

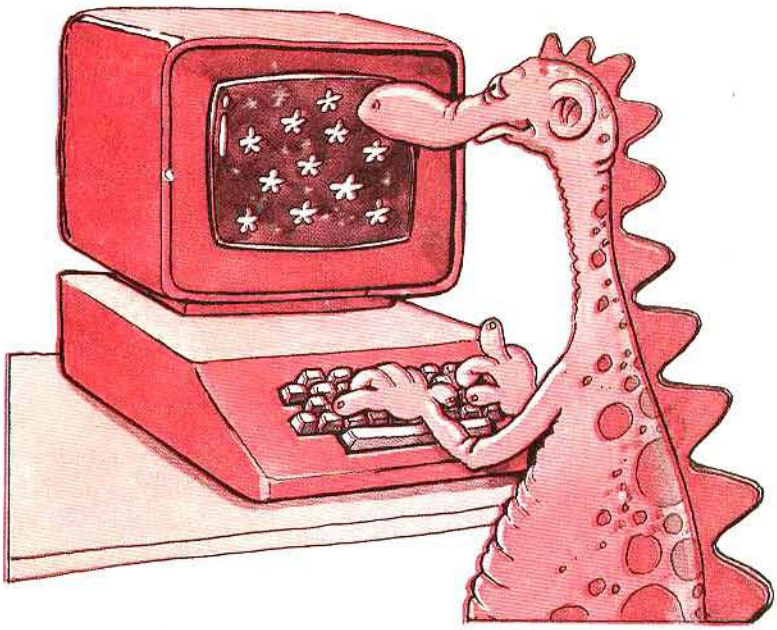
```
10 REM***PROGRAMA DE ADICAO***
20 PRINT "DE ME DÓIS NUMEROS:";
30 INPUT PRIMEIRO, ULTIMO
40 PRINT "SUA SOMA E:"; PRIMEIRO + ULTIMO
50 END
```

A instrução REM é usada para tornar os programas mais legíveis, introduzindo observações no texto. Esta instrução é ignorada pelo intérprete quando o programa é executado e não tem efeito sobre o programa. Aqui estão exemplos adicionais de observações que você pode usar:

```
25 REM AGORA LEIA OS DOIS NUMEROS
45 REM NOS PODEMOS AINDA MULTIPLICA LOS COMO PARTE DA
   INSTRUCAO ACIMA
```

As observações são invisíveis para o intérprete

"Veja como alguns asteriscos fazem seu programa mais legível."



e aqui está o programa resultante:

```
10  REM***PROGRAMA DE ADICAO***
20  PRINT "DE ME DOIS NUMEROS:";
25  REM AGORA LEIA OS DOIS NUMEROS
30  INPUT PRIMEIRO, ULTIMO
40  PRINT "SUA SOMA E:"; PRIMEIRO + ULTIMO
45  REM NOS PODEMOS AINDA MULTIPLICA LOS COMO PARTE DA
        INSTRUCAO ACIMA
50  END
```

Para melhorar a legibilidade, use livremente asteriscos, hifens ou outros símbolos:

```
10   REM***PROGRAMA DE ADICAO***
60   REM - - - - - SEGUNDA PARTE - - - - -
100  REM = = = = = GRAND FINALE = = = = =
200  REM $$$$$ MUDE ESTA SECAO MAIS TARDE $$$$$
```

Várias Instruções em uma Mesma Linha

A maioria dos BASICs permite que você escreva duas ou mais instruções na mesma linha, usualmente separadas por uma vírgula. Aqui está um exemplo:

```
50  PRINT "DIGITE UM NUMERO"; : INPUT NUMERO
```

Isto é equivalente a:

```
50  PRINT "DIGITE UM NUMERO";
60  INPUT NUMERO
```

Note que deve haver ápenas um número de linha por linha. Outrossim, quando duas instruções são escritas na mesma linha há apenas um número de linha à esquerda. Aqui está outro exemplo:

```
100  REM***CALCULE TUDO EM UMA LINHA***
110  SOMA = A+B : PRODUTO = A*B : MEDIA = SOMA / 2
```

Há três instruções na linha 110.

Escrever várias instruções na mesma linha traz benefícios em pelo menos dois casos: para clareza do INPUT e para introduzir observações à direita da instrução. Aqui está um exemplo, mostrando como o INPUT fica mais claro:

```
70  PRINT "INTRODUZA DOIS NUMEROS"; : INPUT N1, N2
```

A linha 70 está escrita, correspondendo ao que acontece na tela. Isto torna o programa mais fácil de ler.

Aqui está um exemplo, mostrando como um REM é introduzido na mesma linha a que se refere a instrução:

```
60  RESULTADO = A+2*B-5 : REM O RESULTADO DEVE SER POSI-
        TIVO
```

Aqui está um outro:

```
50  TEMPERATURA = (FAHR-32) * 5/9 : REM CONVERTER PARA
        CELSIUS
```

Utilizando Espaços em Branco (espaços vazios)

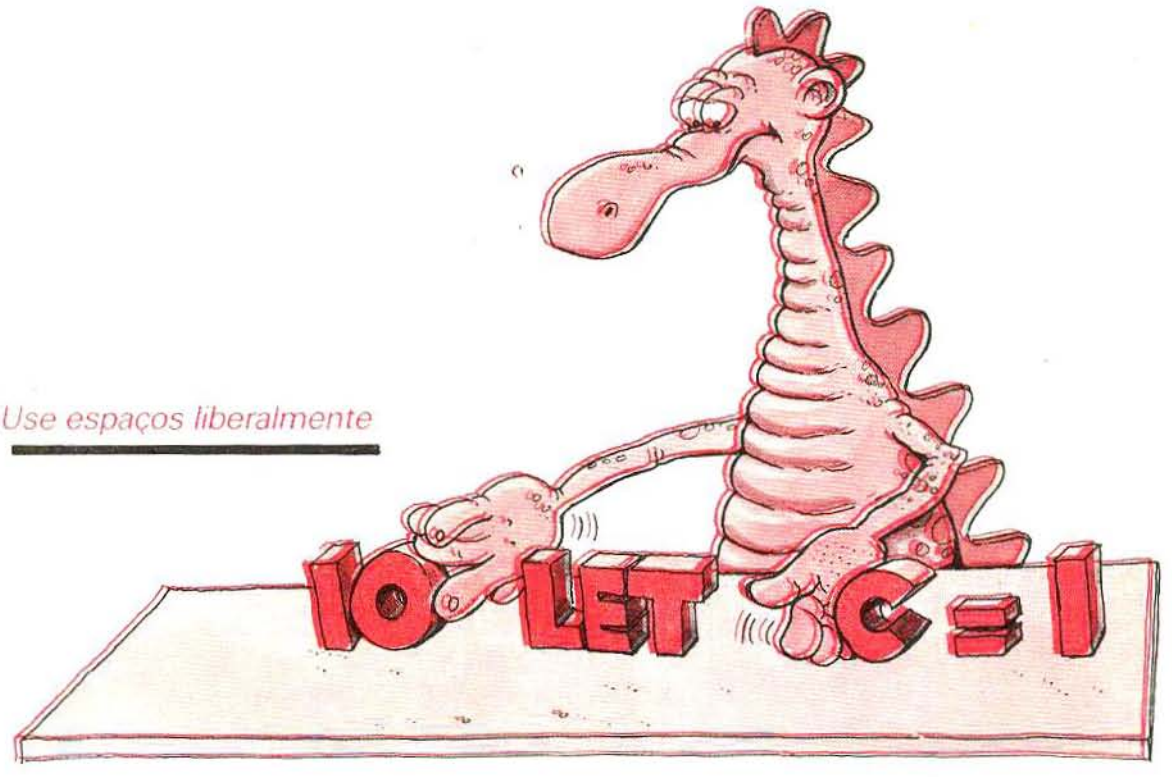
Com exceção de nomes, cadeias de caracteres, ou dados, os brancos são ignorados pelo BASIC. Por exemplo, você pode escrever:

```
20  PRINT4+2*3
```

No entanto, uma instrução como esta é difícil de ler. Para facilitar a releitura de seu programa, use liberalmente os espaços vazios. Use espaços em branco:

- *depois de cada palavra reservada como PRINT ou INPUT.*

Use espaços liberalmente



Aqui estão exemplos:

```
50  PRINT 4
60  INPUT NUMERO
```

▶ *antes e depois de cada operador.*

Aqui estão exemplos:

```
30  PRINT 4 + 2 * 3
40  RESULTADO = A1/((B - C) * D)
```

Você pode ainda querer usar espaços antes de uma palavra reservada, para alinhar ou ressaltar as instruções em seu programa. Aqui está um exemplo:

```
10    PRINT "DEZ"
20    PRINT "VINTE"
90    PRINT "NOVENTA"
100   PRINT "CEM"
200   PRINT "DUZENTOS"
```

Sem os dois espaços alinhadores nas linhas 10, 20 e 90, o programa apareceria assim:

```
10  PRINT "DEZ"
20  PRINT "VINTE"
90  PRINT "NOVENTA"
100 PRINT "CEM"
200 PRINT "DUZENTOS"
```

Finalmente, aqui está um exemplo do uso do espaço vazio dentro da afirmação REM, para facilitar ainda mais a leitura:

```
1  REM  ESTE PROGRAMA MANTEM MEU ESTOQUE
2  REM  DIREITOS RESERVADOS PROPRIOS 1984
3  REM  ESTAS SAO AS VARIEVEIS
4  REM  C    E A COR (1 A 10)
5  REM  U    E O NUMERO DAS UNIDADES (ATE 1.000)
6  REM  S    E O TAMANHO (1 A 50)
7  REM  CST  E O CUSTO UNITARIO
8  REM  R    E O PRECO DA VENDA
9  REM  Q    E A QUANTIDADE DE COMPRA
```

Note como os espaços são utilizados para realçar a legibilidade. Você não pode, no entanto, adicionar espaços no meio de uma palavra reservada, em um nome de variável, ou durante uma resposta a um INPUT, a não ser que os espaços sejam parte de uma cadeia de caracteres.

Aperfeiçoando o Visual de Saída

Duas novas técnicas podem ser usadas para aperfeiçoar a forma como seus resultados aparecem na tela: a instrução CLE (*clear*) e a instrução PRINT vazia.

A instrução CLE limpa a tela ao tempo em que ela é usada. Você pode querer começar cada programa novo com:

```
10  NEW : CLE
```

Esta é uma instrução dupla numa única linha que limpa a memória (NEW) e a tela (CLE).

Aqui está um truque que você pode usar. No final do seu programa você pode escrever:

```
110  CLE : LIST
```

Isto limpará a tela e listará o programa inteiro, antes que ele seja executado.

Vários BASICs, mas não todos, possuem um comando CLE ou similar (CLS, por exemplo). Cuidado que o comando CLE no BASIC Microsoft limpa as variáveis, ao invés da tela. (**Nota**: Se o seu BASIC não tem um CLE, você geralmente pode conseguir o mesmo resultado, digitando:

```
10  PRINT CHR$ (NUM)
```

onde NUM contém o código numérico de um caractere especial, que limpará a tela, especificado no manual.)

A instrução PRINT vazia pode ser usada para mostrar uma linha em branco (salto de linha). Você simplesmente digita:

```
50  PRINT
```

e uma linha em branco é executada. Se você quer pular três linhas na tela, digite:

```
50  PRINT : PRINT : PRINT
```

ou equivalentemente:

```
50  PRINT
60  PRINT
70  PRINT
```

Simplificando a Instrução INPUT

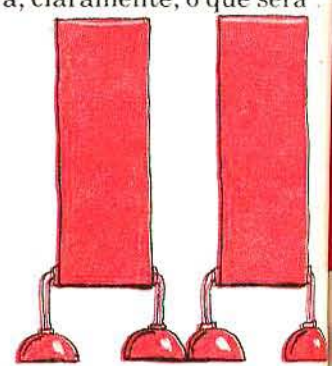
Sempre que você usar uma instrução INPUT, deve lembrar ao usuário do programa, explicando sobre o que entrar. Aqui está um exemplo:

```
50  PRINT "DIGITE DOIS NUMEROS INTEIROS";
60  INPUT IDADE1, IDADE2
```

A seqüência PRINT-INPUT é tão usada que os mais modernos BASICs permitem o encurtamento da instrução INPUT, e você pode escrever:

```
50  INPUT "DIGITE DOIS NUMEROS INTEIROS"; IDADE1, IDADE2
```

o qual é equivalente às duas instruções acima. Esta técnica de simplificação reduz a digitação necessária e mostra, claramente, o que será executado na tela.



Se o seu BASIC não permite esta simplificação, você pode usar uma instrução dupla em uma linha:

```
50  PRINT "DIGITE DOIS NUMEROS INTEIROS"; :
    INPUT IDADE1, IDADE2
```

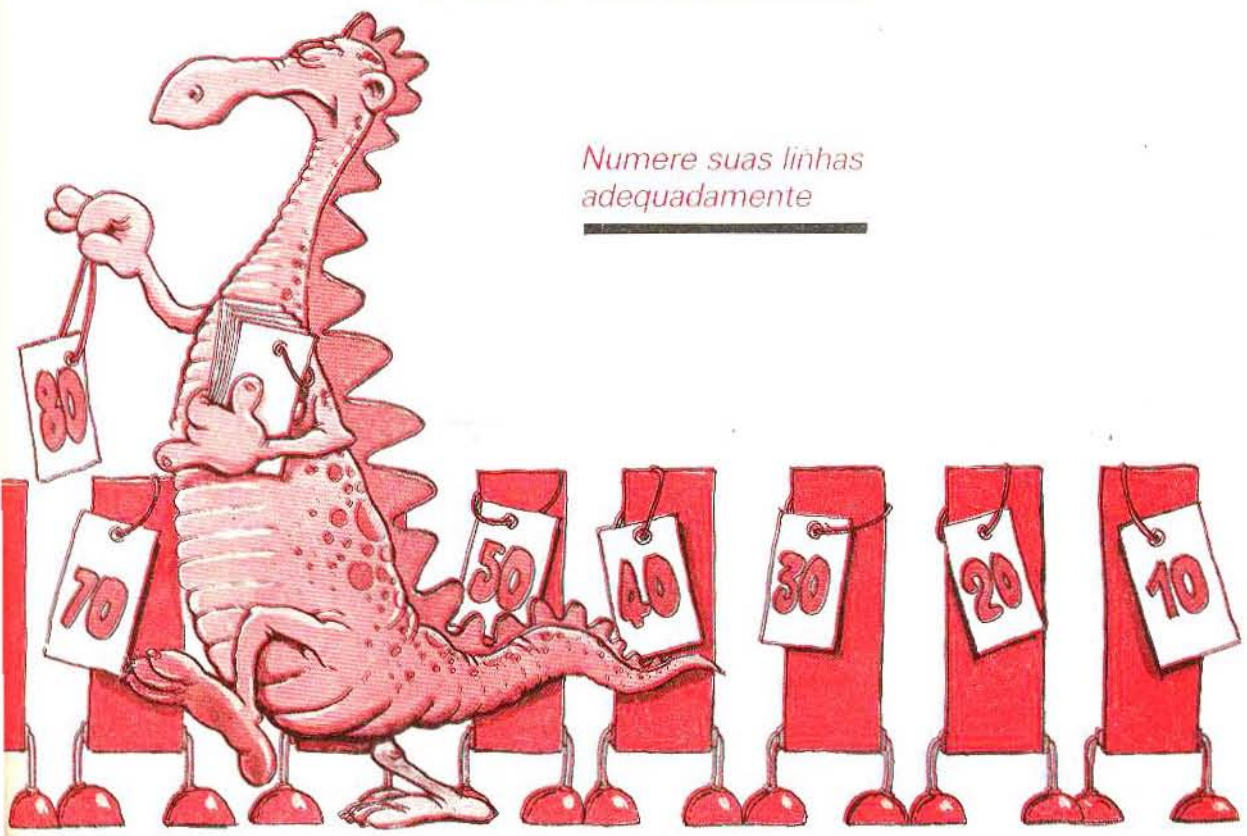
Selecionando os Nomes das Variáveis

Você sempre deveria selecionar nomes de variáveis que possam facilmente lembrar-lhe o que o nome representa. Por outro lado, você pode encontrar dificuldades para escrever um programa longo e cometer vários erros acidentais. Assim, se os nomes de suas variáveis não forem claros, você pode, mais tarde, ser incapaz de imaginar o que faz o seu programa.

Se o seu BASIC permite apenas uma única letra além de um dígito opcional como nome de variáveis, não há muito o que você possa fazer. Aqui estão alguns nomes significativos:

```
10  REM NOS USAREMOS:
20  REM   R   PARA RESULTADO
30  REM   P   PARA PRIMEIRO NUMERO
40  REM   U   PARA ULTIMO NUMERO
```

Numere suas linhas adequadamente



Se o seu BASIC permite letras e nomes variados, utilize-os. Aqui está o exemplo acima, reescrito para utilizar esta facilidade:

```
10  REM AQUI ESTAO AS VARIAVEIS PARA O SEGUNDO NUMERO
20  REM    RESULTADO    PARA RESULTADO
30  REM    PRIMEIRO     PARA PRIMEIRO NUMERO
40  REM    ULTIMO       PARA ULTIMO NUMERO
```

Aplicam-se duas restrições essenciais.

- Seu BASIC usualmente restringirá você a um número máximo de caracteres.
- Você não pode usar uma palavra reservada como PRINT, REM oú INPUT, como um nome de variável.

É uma boa idéia identificar os nomes de todas as variáveis no começo de cada programa, assim como algumas fórmulas e equações que queira utilizar.

A Numeração Apropriada das Linhas

Até aqui, neste capítulo, nós numeramos as linhas com múltiplos de 10:

```
10  (instrução)
20  (instrução)
30  (instrução)
```

Você pode, no entanto, usar qualquer seqüência que queira, contanto que use números inteiros positivos e não vá além do limite do número de linhas do seu intérprete. Por exemplo, você pode escrever:

```
1  (instrução)
2  (instrução)
3  (instrução)
```

ou:

```
100  (instrução)
200  (instrução)
300  (instrução)
```

Nós tomamos a precaução de deixar um espaço padrão entre números consecutivos de instrução para que possamos facilmente adi-

cionar correções ou melhorias mais tarde. Por exemplo, aqui está a versão 1 de um programa:

```
10  REM***PROGRAMA DE MULTIPLICACAO***
20  INPUT "DE ME DOIS NUMEROS";N1,N2
30  PRINT "O PRODUTO E: ";N1*N2
40  END
```

Agora nós queremos clarificar o programa e melhorar o seu visual. Para fazer isto, nós digitaremos a seguinte instrução adicional:

```
 5  NEW : CLE
15  PRINT "ESTE E UM PROGRAMA DE MULTIPLICACAO AUTOMA-
        TICA"
16  PRINT : PRINT : PRINT
35  PRINT : PRINT
```

A nova informação será inserida automaticamente. Vamos agora listar — LIST — o resultado:

```
> LIST
 5  NEW : CLE
10  REM***PROGRAMA DE MULTIPLICACAO***
15  PRINT "ESTE E UM PROGRAMA DE
        MULTIPLICACAO AUTOMATICA"
16  PRINT : PRINT : PRINT
20  INPUT "DE ME DOIS NUMEROS"; N1, N2
30  PRINT "O PRODUTO E:"; N1*N2
35  PRINT : PRINT
40  END
```

Aqui está uma amostragem do nosso programa melhorado:

```
DE ME DOIS NUMEROS? 12, 15
O PRODUTO E : 180
```

Se você planeja fazer alguma correção ou adicionar alguma instrução, deve deixar grandes espaços na numeração de suas linhas e usar, por exemplo:

```
10  (instrução)
50  (instrução)
60  (instrução)
100 (instrução)
```

Muitos BASICs têm um comando especial chamado RENUMBER (renumeração), que automaticamente renumera todas as linhas de seu programa com um incremento de 10. Isto é muito útil em um programa longo onde você pode esgotar uma seqüência de números ao inserir uma correção ou "acréscimo".

Sumário

Escrever programas que funcionem envolve um ingrediente chave: disciplina. É importante ser tão ordenado e organizado quanto possível, ao se escrever programas. Simplificações incluem a possibilidade de enganos. Em particular, use o tempo necessário para tornar claro seus programas. Neste capítulo nós descrevemos e acentuamos as técnicas necessárias para escrever programas claros.

À medida que você escreva programas em BASIC, você deve envidar esforços para seguir as sugestões oferecidas neste capítulo. É essencial que você adquira bons hábitos para programar; caso contrário seus programas podem se tornar ilegíveis ou até mesmo não funcionar, à proporção que se tornem cada vez mais complexos.

Exercícios

5-1: Descreva as técnicas que melhoram a legibilidade da tela.

5-2: O que vem abaixo é legal?

a. A = A + 1

b. A = A + 1

c. PRINT ALPHA + 2

d. SUM = 2 + (3 + (4/5))/2

e. IN PUT NUMERO

f. SUM = 2 2 + 3 3

5-3: Explique por que a maioria dos INPUTs devem ser acompanhados de uma mensagem prévia:

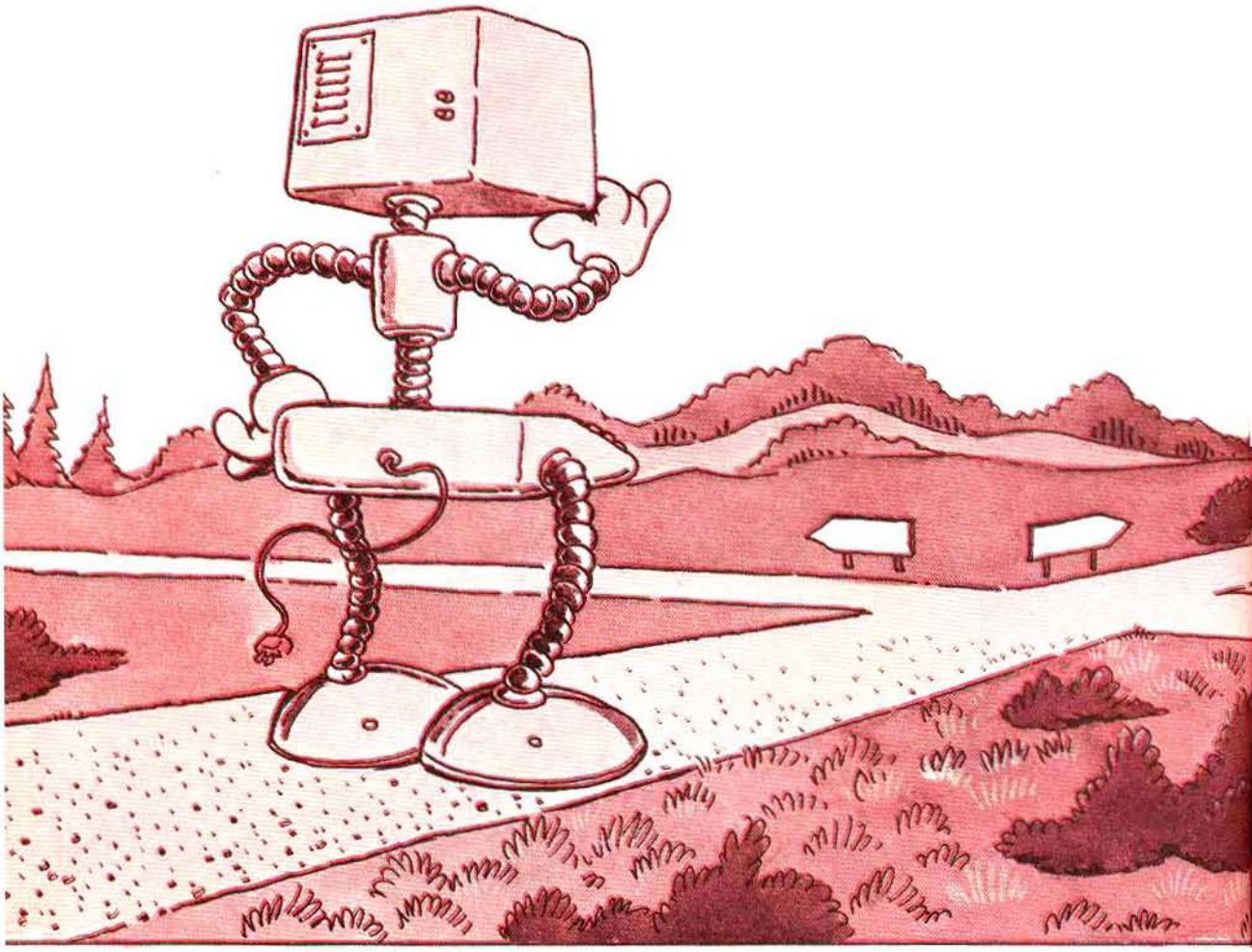
5-4: Escreva três exemplos de INPUTs encurtados.

5-5: Por que usar REMs?

5-6: Qual é o valor de A depois destas duas instruções:

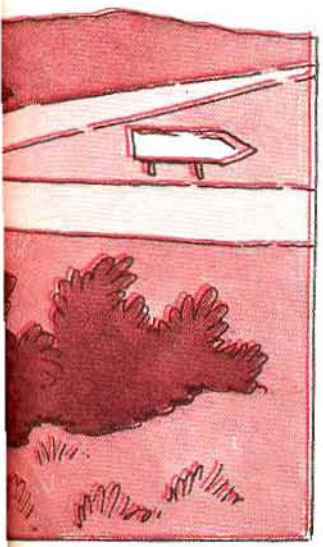
```
30  A= 3
40  REM A = 4
```

Tomando



Decisões

6



Até aqui, aprendemos a nos comunicar com o computador e a utilizar aritméticas simples. No entanto, os programas já realizados foram bobos e simples de serem realizados manualmente; tal fato se deve a termos usado apenas os recursos elementares do computador. Nós não tiramos vantagens dos recursos mais avançados. Por exemplo: os computadores são particularmente bons para realizar duas tarefas: tomar decisões complexas (baseadas em lógica e em valores) e executar tarefas repetitivas muitas vezes, num curto período de tempo. Isto é o que aprenderemos a fazer neste e no próximo capítulo. Em particular, aprenderemos a tomar decisões. Nosso programa se tornará "inteligente" à medida que decide o que fazer.

Em BASIC, decisões de programa são tomadas testando um valor, usando a instrução IF. Se o teste for bem-sucedido, uma parte do programa é executada. Se falhar, uma outra parte é que será executada. Aprenderemos agora a usar a instrução IF para realizar testes. Também aprenderemos o uso da instrução GOTO para forçar o programa a executar um grupo de instruções fora da seqüência numérica:

A Instrução IF

A instrução IF é escrita:

IF (condição) THEN (instrução, isto é, faça algo)

Aqui está um exemplo:

```
IF I = 1 THEN PRINT "UM"
```

O efeito dessa instrução é claro. Se (IF) o valor da variável I é igual a 1, ao tempo da execução desta instrução, então (THEN) a palavra um é impressa. Se I não for igual a 1, nada acontecerá e a instrução seguinte do programa é executada.

I = 1 é chamada de uma *expressão* lógica. A expressão I = 1 é *verdade* quando I é igual a 1; caso contrário, ela é *falsa*.

A informação IF... THEN permite a você testar os valores de uma expressão e executar uma instrução ou outra — isto é, tomar uma decisão — dependendo do resultado do teste. Aqui está um outro exemplo:

```
10  INPUT I
20  IF I = 1 THEN PRINT "UM"
30  END
```

Olhe para a expressão indecisa do Dino. Ele precisa escolher um dos dois caminhos.

Agora siga o programa. Digite "1" no teclado. Sua tela mostrará:

```
>RUN
?1
UM
>
```

Siga o programa novamente. Digite "2" no teclado. Em sua tela aparecerá:

```
>RUN
?2
>
```

Desta vez, nenhuma mensagem foi executada em resposta ao 2.

Vamos agora ensinar nosso programa a reconhecer os números de 1 a 4:

```
10  REM ESTE PROGRAMA RECONHECE OS NUMEROS DE 1 A 4
20  INPUT "DIGITE UM NUMERO INTEIRO:"; NUMERO
30  IF NUMERO = 1 THEN PRINT "UM"
40  IF NUMERO = 2 THEN PRINT "DOIS"
50  IF NÚMERO = 3 THEN PRINT "TRES"
60  IF NUMERO = 4 THEN PRINT "QUATRO"
70  END
```

Vamos executar o programa. Aqui estão duas execuções típicas, como aparecem na tela (em negrito):

```
>RUN
DIGITE UM NUMERO INTEIRO:? 3
TRES
>RUN
DIGITE UM NUMERO INTEIRO:? 5
>
```

Isto é bom, mas ainda não é perfeito. O ideal seria que quando digitássemos 5, o programa respondesse alguma coisa como:

```
EU NAO CONHEÇO ESTE NUMERO.
```

ou então requisitasse um outro número inteiro.

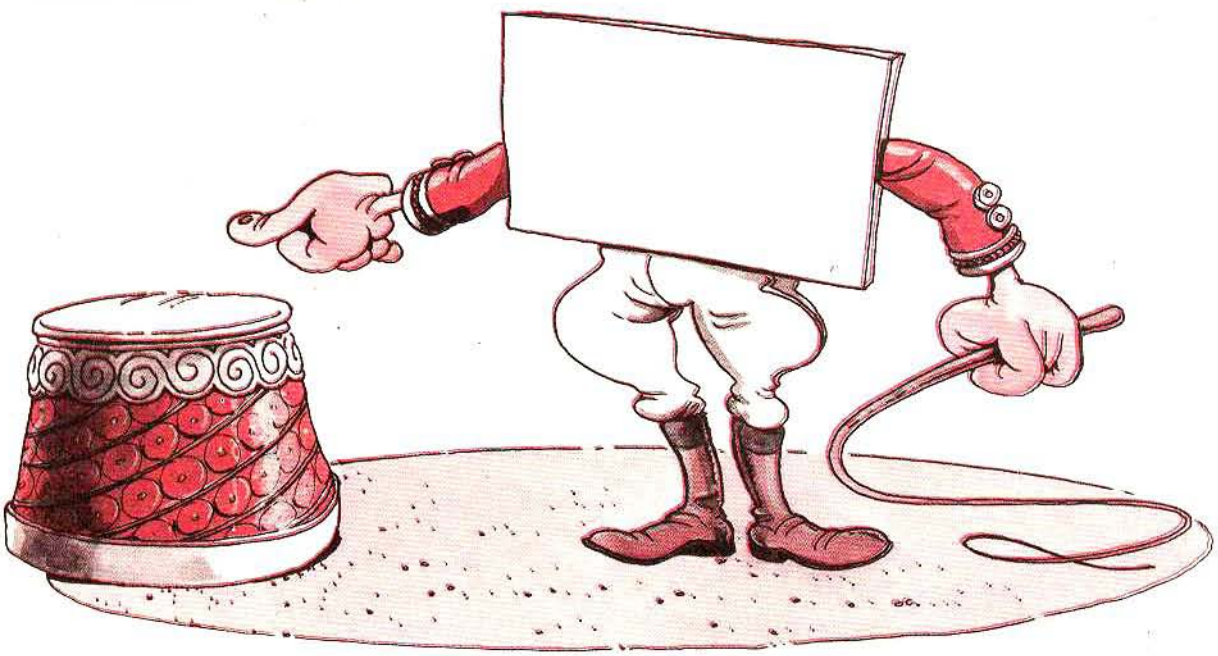
Existe uma característica especial da instrução IF que permite tal solução. Por exemplo, você pode escrever:

```
70  IF NUMERO = 5 THEN 20
```

onde 20 é o número da linha a ser executada caso o teste tenha sucesso. Esta é uma nova forma da instrução IF. Esta instrução indica que se o número é igual a 5, então a linha 20 deve ser executada em seguida. Agora nós pularemos fora da seqüência! Aqui está um exemplo:

```
10  INPUT I
20  IF I = I THEN 50
30  PRINT "VOCE NAO DIGITOU UM 1"
40  END
50  PRINT "VOCE DIGITOU UM 1"
60  END
```

Nós podemos fazer o programa pular fora da seqüência



RUN este programa e digite "1" no teclado. Sua tela mostrará:

```
>RUN
? 1
VOCE DIGITOU UM 1
>
```

RUN o programa novamente e digite um "2" no teclado. Sua tela mostrará:

```
>RUN
?2
VOCE NAO DIGITOU UM 1
>
```

Nosso programa ficou "inteligente", isto é, dá uma mensagem apropriada quando o dado de entrada é ou não 1. Você iria se admirar se tivéssemos chegado ao mesmo resultado usando o formato original da instrução IF. Vamos tentar:

```
10  INPUT
20  IF I = 1 THEN PRINT "ESTE E UM UM"
30  PRINT "ESTE NAO E UM UM"
40  END
```

Agora siga este programa e digite um 1 no teclado. O resultado é o seguinte:

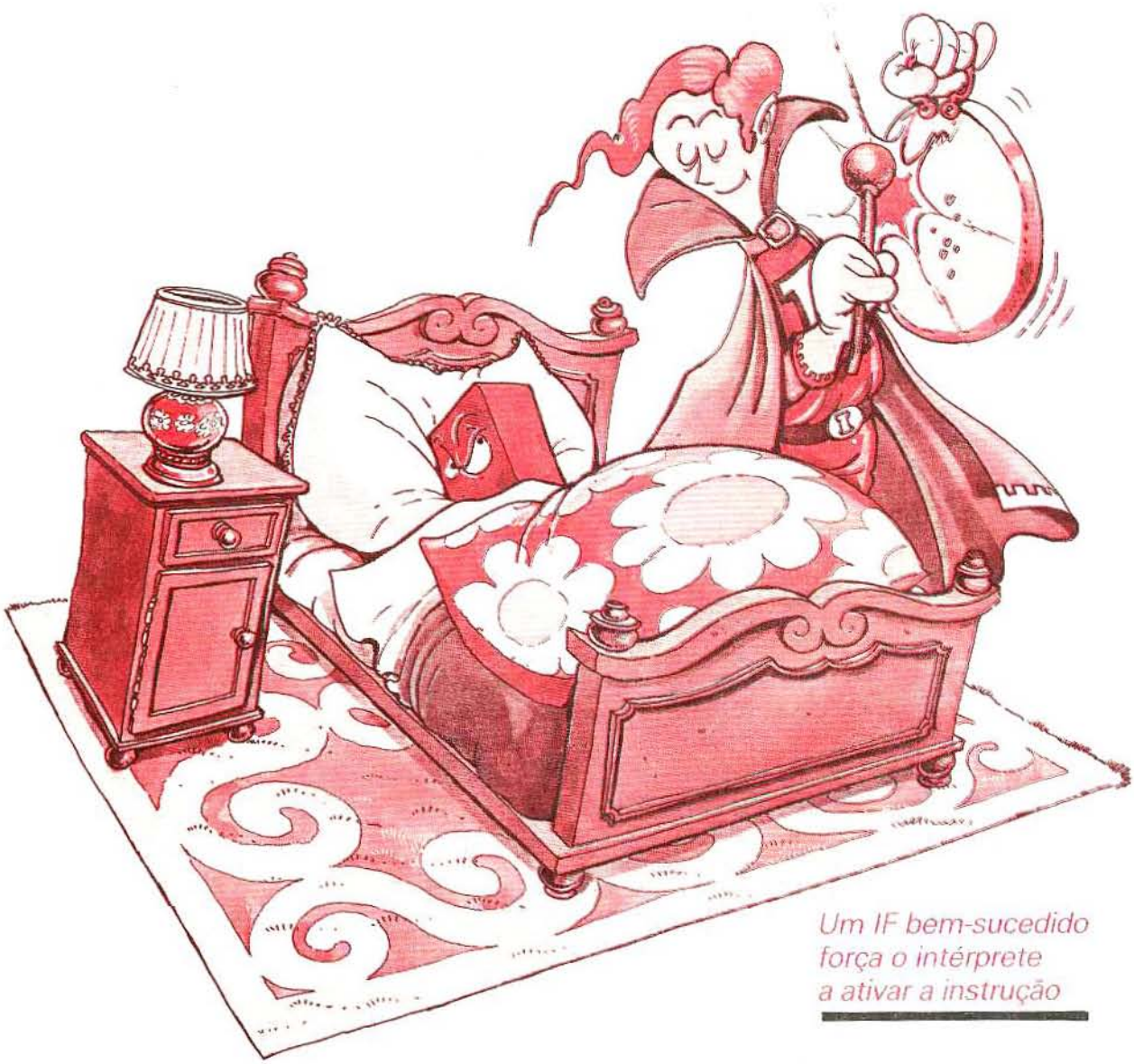
```
>RUN
?1
ESTE E UM UM
ESTE NAO E UM UM
```

Sou um erro no programa. Peguei você.

Não funcionou. Indiferente ao sucesso ou fracasso do IF, a próxima instrução que se segue no programa (a instrução 30) foi executada.

Neste exemplo obtemos primeiro a mensagem correta quando o IF é executado:

```
ESTE E UM UM
```



Um IF bem-sucedido força o intérprete a ativar a instrução

Então a segunda mensagem é executada:

```
ESTE NAO E UM UM
```

A nova forma da instrução IF:

```
IF I = 1 THEN 50
```

elimina este problema. Usaremos esta instrução freqüentemente em nossos programas.

Vamos agora olhar atentamente a instrução IF para utilizá-la plenamente. A forma geral da instrução IF... THEN é a seguinte:

IF (expressão lógica) THEN (instrução executável ou
número de linha).

Vamos examinar agora as expressões lógicas e as instruções executáveis, uma de cada vez.

As Expressões Lógicas

Em nosso exemplo, I = 1 é uma *expressão lógica*; isto é, pode ser tanto *verdadeira* quanto *falsa*. Verdadeiro ou falso são chamados *valores lógicos*. Aqui estão alguns exemplos de expressões lógicas:

```
I = 1          (I igual a 1)
I>4            (I é maior que 4)
NUMERO<100     (NUMERO é menor que 100)
ANO<>5         (ANO não é igual a 5)
IDADE<13       (IDADE é menor que 13)
```

Uma expressão lógica combina valores ou variáveis com operadores lógicos. Para completar, os operadores que você pode usar como expressões lógicas são:

```
=    igual
<>   não igual ou
     diferente        (em matemática isto é escrito como ≠ ou #)

<    menor que
>    maior que
<=   menor que ou
     igual            (em matemática se escreve ≤)
>=   maior que ou
     igual            (em matemática se escreve ≥)
```

Aqui estão algumas expressões lógicas mais complexas:

```
(NUMERO + 2) > 4
(IDADE - 5) > = 10
((2 * I - 5)/2) < 10
2>1
```

Você pode ainda escrever:

```
4>2   (isto é sempre verdadeiro)
4=2   (isto é sempre falso)
```

As seguintes não são lógicas válidas:

2<I<0 *(apenas um operador relacional pode ser usado)*
(2 IDADE − 2) *(expressão não-válida — falta um *. Deveria ser escrito (2 * IDADE − 2) < 5)*

Você pode ainda combinar expressões lógicas usando os *operadores lógicos* AND, OR e NOT. Por exemplo, você pode escrever:

```
IF (IDADE>9) AND (IDADE<20) THEN PRINT "E UM ADOLESCENTE"
```

Esta instrução imprimirá "E UM ADOLESCENTE" sempre que a idade for maior que 9 e menor que 20. O AND será satisfeito, isto é, verdadeiro, quando *ambas* as condições forem verdadeiras. Note que você *não* pode escrever:

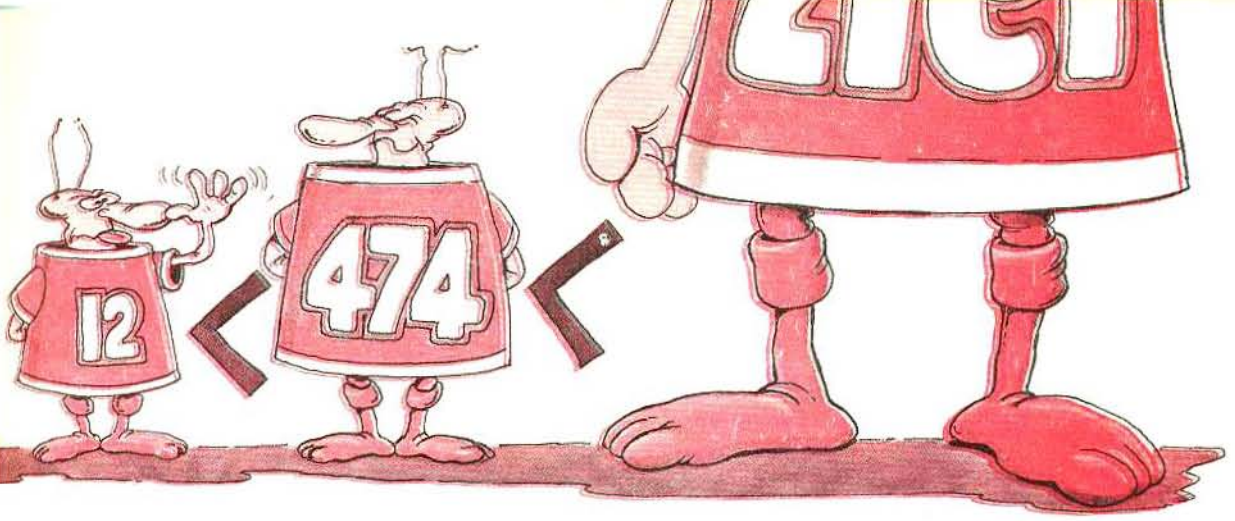
```
IF IDADE (>9 AND < 20) THEN...
```

pois, dentro dos parênteses deve estar apenas uma expressão válida.

Vejamos um outro exemplo, usando duas expressões lógicas:

```
20  INPUT RESPOSTAS
30  IF (RESPOSTA$ = "SIM") OR (RESPOSTA$ = "NAO") THEN 60
40  PRINT "RESPOSTA NAO VALIDA"
    ...
60  PRINT "RESPOSTA VALIDA — PROSSIGA"
```

Este segmento de programa coleta uma resposta na variável RESPOSTA$ (recorde que no final do nome da variável é usado para designar uma variável alfanumérica *(string)*, isto é, uma cadeia de caracteres. SIM ou NÃO são as únicas respostas válidas; este programa verifica a validade do que você digitou.



Operadores relacionais podem ser usados em variáveis

Se você digita SIM, então (RESPOSTA$ = "SIM") é verdadeiro, e o IF é bem-sucedido; desta forma a instrução 60 é executada em seguida e o programa imprime:

```
RESPOSTA VALIDA — PROSSIGA
```

De maneira similar, se você digita NÃO, ocorre o mesmo.

Se você digitar qualquer outra coisa, você obterá um diagnóstico do tipo:

```
RESPOSTA NAO VALIDA
```

Um OR é satisfeito, isto é, *verdadeiro*, quando ao menos uma das condições é verdadeira. O OR não é satisfeito quando ambas as condições são falsas. Por exemplo, se você digitar "YA", então (RESPOSTA$ = "SIM") falha, e a segunda condição de OR é testada (RESPOSTA$ = "NAO"). Ela falha e o programa executa a mensagem:

```
RESPOSTA NAO VALIDA
```

Finalmente, NOT pode ser usado para negar uma condição. Aqui está um exemplo de uma instrução IF complexa:

```
IF ((MEDIA<3.5) AND (LASTEXAM<3.0) AND NOT (ORAL>4.0)) THEN
   PRINT "FALHO"
```

Nesta instrução testamos três condições de uma só vez. Não discutiremos aqui instruções tão complexas quanto esta, mas você pode querer experimentá-las por si próprio.

Vamos recordar a definição da instrução IF:

IF (expressão lógica) THEN (instrução executável
ou linha numerada)

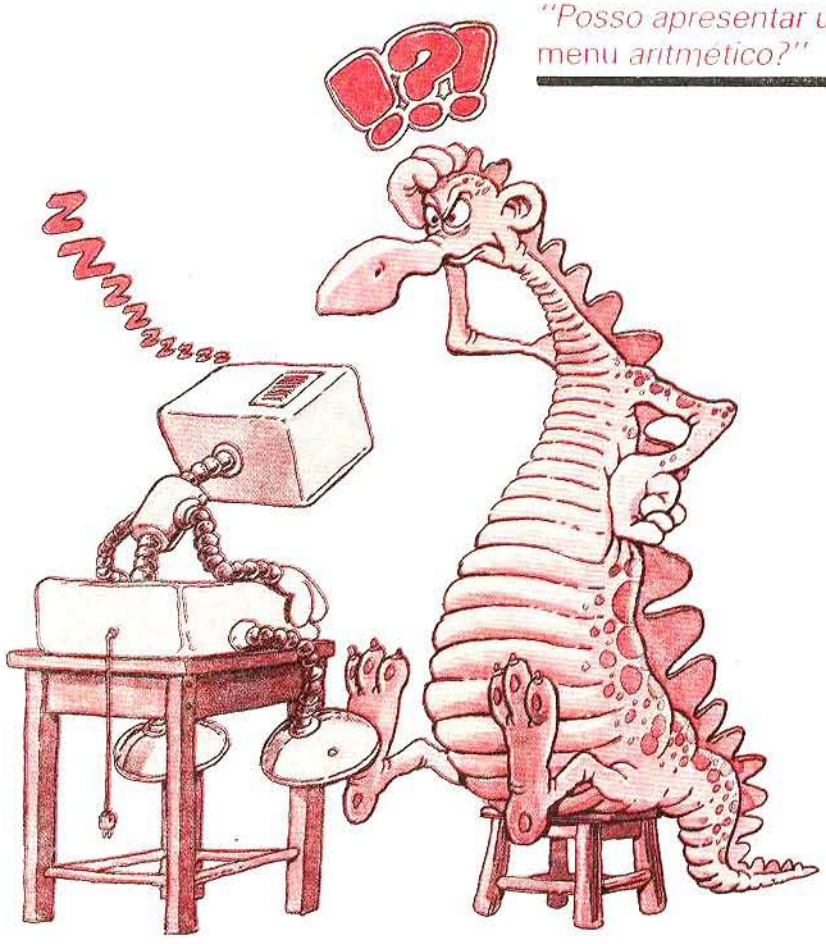
Agora que estamos familiarizados com expressões lógicas, vamos examinar o lado direito desta definição:

THEN (instrução executável ou linha numerada)

Uma instrução executável é simplesmente qualquer instrução que seja executada. Ela pode ser uma designação, um INPUT ou um PRINT. Não pode, no entanto, ser uma outra instrução IF, ou um comando como REM, CLE, NEW ou LIST.

Agora que nós entendemos a teoria da instrução IF, vamos colocá-la em prática.

"Posso apresentar um menu aritmético?"



Um Exercício de Aritmética

Usando nossas novas experiências, vamos agora desenvolver um programa que execute um "menu" na tela. Dependendo da seleção do usuário, este programa educacional conseguirá adições, subtrações, multiplicações ou divisões.

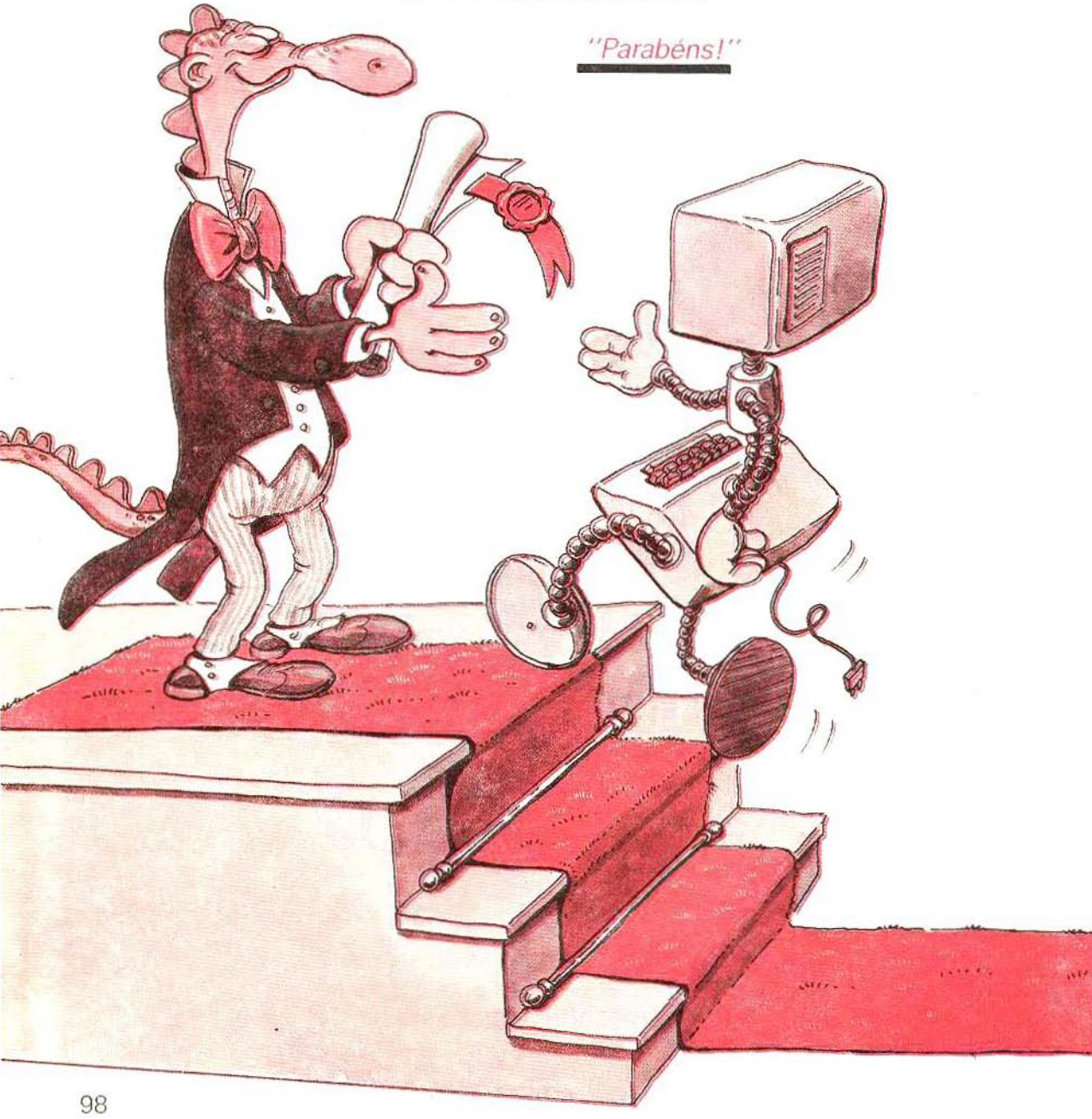
Aqui está o diálogo que planejamos gerar na tela:

```
SEJA BEM VINDO PROFESSOR COMPUTADOR
QUERO VERIFICAR SUA EXPERIENCIA ARITMETICA
O QUE VOCE QUER PRATICAR?
    — ADICAO            (DIGITE 1)
    — SUBTRACAO         (DIGITE 2)
    — MULTIPLICACAO     (DIGITE 3)
    — DIVISAO           (DIGITE 4)
QUAL E A SUA ESCOLHA (DIGITE 1, 2, 3 OU 4)?: 3
-------------------------------------------------
ESTA BEM. VAMOS MULTIPLICAR.
QUANTO E 2 VEZES 3: 6
ESTA CERTO. PARABENS.
```

Agora, aqui está o programa que atende o apresentado:

```
10  REM * * * ESTE PROGRAMA TREINA VOCE EM ARITMETICA * * *
20  PRINT "SEJA BEM VINDO PROFESSOR COMPUTADOR"
30  PRINT "QUERO VERIFICAR SUA EXPERIENCIA ARITMETICA"
40  PRINT "O QUE VOCE QUER PRATICAR?"
50  PRINT " - ADICAO (DIGITE 1)"
60  PRINT " - SUBTRACAO (DIGITE 2)"
70  PRINT " - MULTIPLICACAO (DIGITE 3)"
80  PRINT " - DIVISAO (DIGITE 4)"
90  INPUT "QUAL E A SUA ESCOLHA:"; ESCOLHA
100 IF (ESCOLHA = 1) THEN 200
110 IF (ESCOLHA = 2) THEN 300
120 IF (ESCOLHA = 3) THEN 400
130 IF (ESCOLHA = 4) THEN 500
140 PRINT "ESCOLHA INCORRETA. VOCE DEVE SELECIONAR UM
        NUMERO ENTRE 1 E 4"
150 PRINT "ATE LOGO": END
190 REM ..... ADICAO .....
200 PRINT "ESTA BEM. VAMOS SOMAR"
210 INPUT "QUANTO E 4 + 7 : "; NUMERO
220 IF (NUMERO< > 11) THEN 600
230 PRINT ESTA CERTO. PARABENS: END
```

```
290  REM ..... SUBTRACAO .....
300  PRINT "ESTA BEM. VAMOS SUBTRAIR"
310  INPUT "QUANTO E 9 - 5: ";NUMERO
320  IF (NUMERO < >4) THEN 600
330  PRINT "ESTA CERTO. PARABENS": END
390  REM ..... MULTIPLICACAO .....
400  PRINT "ESTA BEM. VAMOS MULTIPLICAR"
410  INPUT "QUANTO E 2 VEZES 3:"; NUMERO
420  IF (NUMERO < > 6) THEN 600
430  PRINT "ESTA CERTO: PARABENS": END
490  REM ..... DIVISAO .....
```



```
500  PRINT "ESTA BEM. VAMOS DIVIDIR"
510  INPUT "QUANTO E 9 DIVIDIDO POR 3: "; NUMERO
520  IF (NUMERO < > 3) THEN 600
530  PRINT "ESTA CERTO. PARABENS": END
590  REM ..... ERRO DE CALCULO .....
600  PRINT "ERRADO. DESCULPE E ATE LOGO." : END
```

Este programa parece imponente no tamanho, mas é realmente muito simples. Vamos examiná-lo:

As instruções de 20 a 90 produzem o display ou "menu" na tela. O programa verifica a seleção do usuário nas instruções de 100 a 130 (o parêntese depois de cada IF não é obrigatório, ele foi incluído para legibilidade. Se o usuário digitar "1", então (ESCOLHA = 1) é verdadeiro e a instrução 200 é executada a seguir. Se o usuário digitar algum número que não seja 1, 2, 3 ou 4 então a instrução 140 é executada e o programa diz:

```
ESCOLHA INCORRETA. VOCE DEVE SELECIONAR UM NUMERO
ENTRE 1 E 4
ATE LOGO
>
```

isto é, o : END, instrução na linha 150.

No nosso exemplo, digitamos 3. A instrução 100 falhou e, então, a instrução 110 foi executada a seguir. A instrução 120 tem sucesso desde que a escolha = 3 é verdadeira, e a instrução 400 é executada a seguir. Aqui está o segmento do programa correspondente:

```
400  PRINT "ESTA CERTO. VAMOS MULTIPLICAR"
410  INPUT "QUANTO E 2 VEZES 3 :"; NUMERO
420  IF (NUMERO < > 6) THEN 600
430  PRINT "ESTA CERTO. PARABENS" : END
```

Em nosso exemplo, digitamos 6 em resposta à instrução 410. Quando a instrução 420 é executada (NUMERO < > 6) é falso desde que o número = 6. (Lembre-se de que < > significa, diferente de). Em

seguida, a instrução seguinte a ser executada é a instrução 430, e o programa responde com:

```
ESTA CERTO. PARABENS
>
```

e pára até a linha 430, contendo *duas* instruções. A segunda instrução é:

```
: END
```

Olhando para este programa você pode rapidamente descobrir uma nova frustração. Se você digitar outro número que não seja 1, 2, 3 ou 4 depois que o "menu" é mostrado, ou se você der uma resposta aritmética errada, o programa pára abruptamente. O ideal seria que o programa continuasse. Por exemplo, seria bom se depois que o programa dissesse ao usuário que um número sem ser 1 a 4 não é válido, ele pedisse uma nova escolha. Gostaríamos de ser capazes de voltar ao começo do programa e recomeçá-lo, ou mais genericamente, ser capazes de ir a *qualquer* parte do programa. Isto é possível com a instrução GOTO. Vamos examinar esta instrução.

A Instrução GOTO

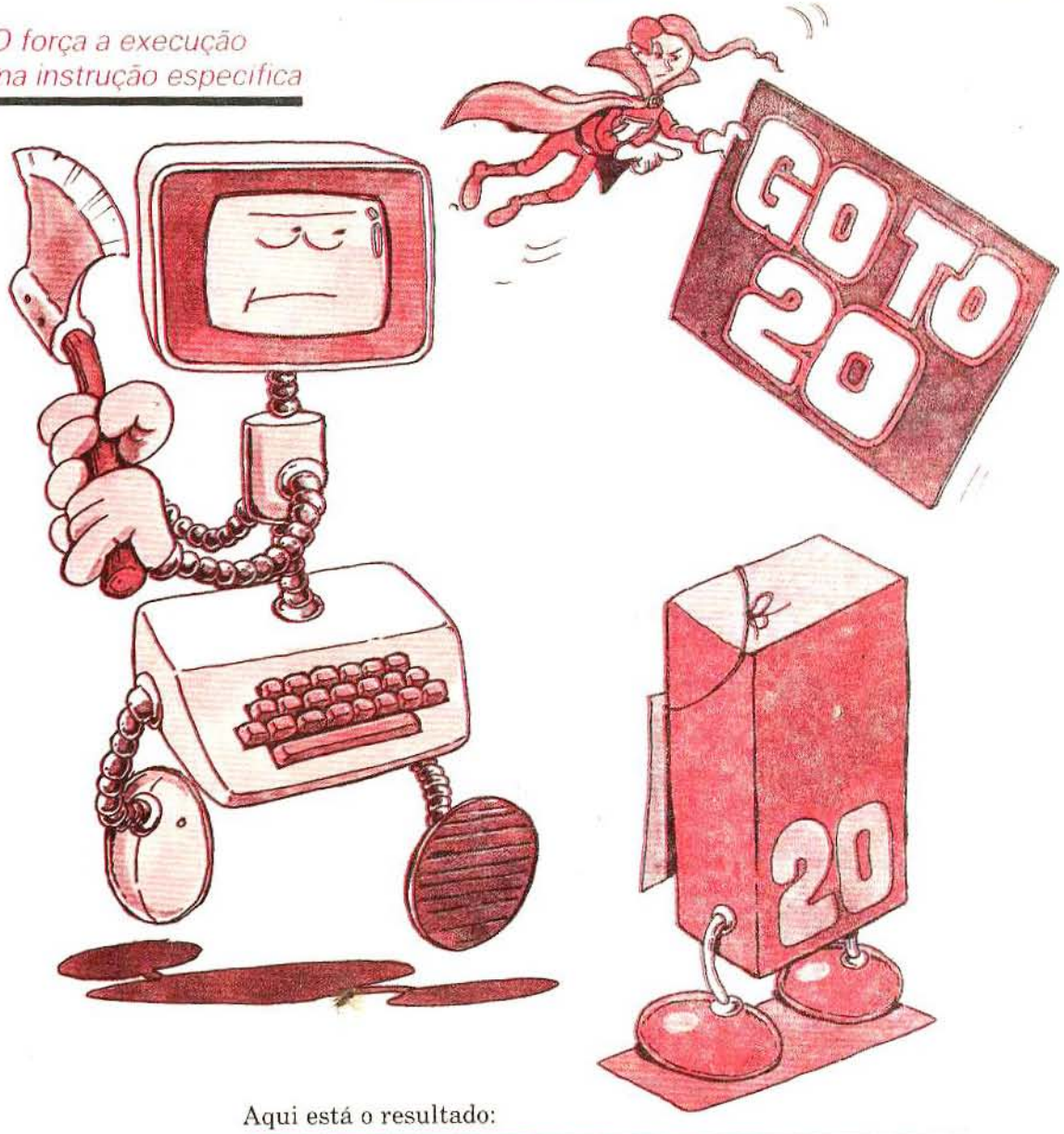
A instrução GOTO é escrita:

```
GOTO (número de linha)
```

Isto força a execução de uma instrução específica. Aqui está um exemplo:

```
10 PRINT "ESTE PROGRAMA RECONHECE 1'S. DIGITE 0 PARA
   PARAR"
20 INPUT "DIGITE UM NUMERO:"; NUMERO
30 IF NUMERO = 1 THEN PRINT "UM"
40 IF NUMERO = 0 THEN 60
50 GOTO 20
60 END
```

GOTO força a execução de uma instrução específica



Aqui está o resultado:

```
> RUN
ESTE PROGRAMA RECONHECE 1'S. DIGITE 0 PARA PARAR.
DIGITE UM NUMERO:? 1
UM
DIGITE UM NUMERO:? 5
DIGITE UM NUMERO:? 25
DIGITE UM NUMERO:? 1
UM
DIGITE UM NUMERO:? 0
>
```

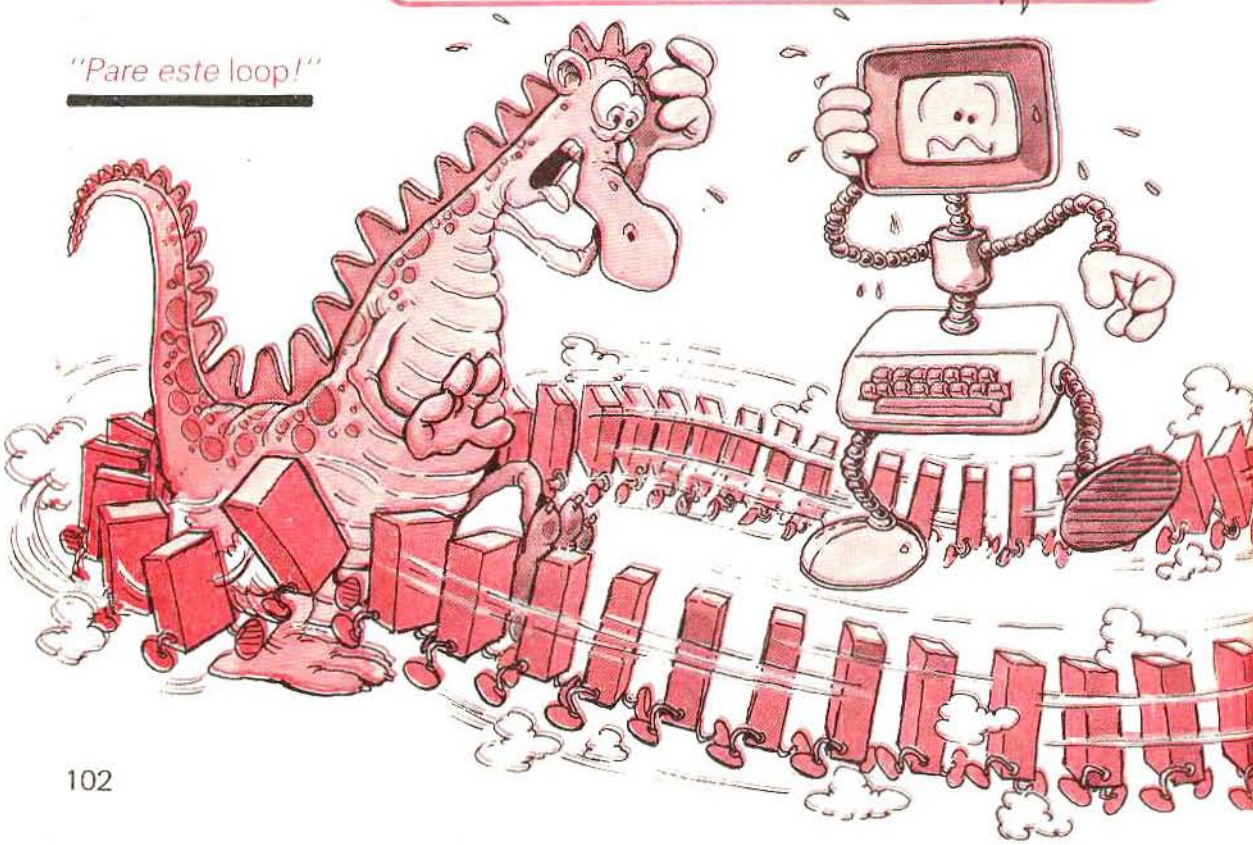
Cada vez que você digitar 1, o programa reconhece-o e executa UM. Cada vez que você digita qualquer outra coisa, o número é ignorado e o programa requisita um novo valor. O programa volta continuamente ao princípio. Isto é chamado um *loop* ou *laço*. O programa volta sobre si mesmo. Se você digita um 0, ele é detectado pela instrução 40 e o programa salta para a instrução 60 e termina. Vamos agora retirar a instrução 40. O programa ficará assim.

```
10  PRINT "ESTE PROGRAMA RECONHECE 1'S. DIGITE 0 PARA
        PARAR."
20  INPUT "DIGITE UM NUMERO;"; NUMERO
30  IF NUMERO = 1 THEN PRINT "UM"
50  GOTO 20
60  END
```

Aqui está a amostra do processamento simples:

```
ESTE PROGRAMA RECONHECE 1'S. DIGITE 0 PARA PARAR.
DIGITE UM NUMERO:? 2
DIGITE UM NUMERO:? 1
UM
DIGITE UM NUMERO:? 5
DIGITE UM NUMERO:? 0
DIGITE UM NUMERO:?
```

"Pare este loop!"



Como um aprendiz de feiticeiro, nós criamos um terrível problema: este programa não pára nunca! Este é um erro comum de programação chamado *loop sem fim*. O programa pode continuar executando para sempre. Não se preocupe. Isto não estraga o equipamento. Para pará-lo você deve pressionar a tecla designada pelo fornecedor do Intérprete para interromper um programa BASIC (tente CTRL C). O pior que pode acontecer é você não se lembrar do que fazer, então desligue seu computador e ligue-o novamente. Mas lembre-se, se você desligar seu computador, perderá tudo o que tiver sido digitado antes e não estiver guardado em cassete ou *diskette*. Nós pretendemos prevenir esta situação desagradável, provendo uma saída normal (programada) para cada programa daqui para a frente.

Introduzimos a instrução GOTO; agora, vamos voltar à nossa definição de instrução IF e simplificá-la.

Reavaliando a Instrução IF

Recorde que uma das formas da instrução IF é:

IF (expressão lógica) THEN (número de linha)

A outra é:

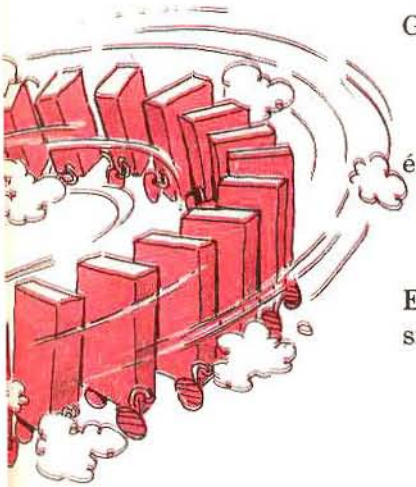
IF (expressão lógica) THEN (instrução executável)

Aqui está um exemplo da primeira maneira:

```
IF NUMERO = 0 THEN 60
```

Isto é o equivalente a:

```
IF NUMERO = 0 THEN GOTO 60
```



GOTO 60 é uma instrução executável e você reconhece que a forma

```
THEN 60
```

é simplesmente uma instrução encurtada de:

```
THEN GOTO 60
```

Então, na realidade, a forma geral da instrução IF é, de fato, mais simples que nossa definição anterior:

IF (expressão lógica) THEN (instrução executável)

Aqui está mais uma simplificação: o uso do THEN antes do GOTO é usualmente opcional. Na maioria dos BASICs as formas seguintes são equivalentes:

```
IF NUMERO = 0 THEN 100
IF NUMERO = 0 THEN GOTO 100
IF NUMERO = 0 GOTO 100
```

Demonstraremos agora o uso de IFs e GOTOs em exemplos de programa.

Contando os Números Um

No capítulo 5 introduzimos a técnica de contagem. Vamos agora usá-la para contar quantos números 1 foram digitados no último programa da sessão precedente. Aqui está o programa melhorado:

```
1   REM CONTAGEM DE 1
10  PRINT "QUERO CONTAR QUANTAS VEZES O NUMERO 1 SOZI-
          NHO FOI DIGITADO."
20  PRINT "DIGITE 0 PARA PARAR"
30  SOMA = 0
40  INPUT "DIGITE UM NUMERO: "; NUMERO
50  IF NUMERO = 0 THEN 100
60  IF NUMERO < > 1 THEN GOTO 40
70  SOMA = SOMA + 1
80  PRINT "UM. TOTAL ATE AGORA:"; SOMA
90  GOTO 40
100  END
```

Aqui está o resultado:

```
QUERO CONTAR QUANTAS VEZES O NUMERO 1 SOZINHO
FOI DIGITADO.
DIGITE UM NUMERO:? 10
DIGITE UM NUMERO:? 1
UM. TOTAL ATE AGORA: 1
DIGITE UM NUMERO:? 9
DIGITE UM NUMERO:? 5
DIGITE UM NUMERO:? 1
UM. TOTAL ATE AGORA:? 2
DIGITE UM NUMERO:? 2
DIGITE UM NUMERO:? 1
UM. TOTAL ATE AGORA: 3
DIGITE UM NUMERO:? 410
DIGITE UM NUMERO:?
```

Vamos examinar o programa. As instruções 10 e 20 executam mensagens:

```
10  PRINT "QUERO CONTAR QUANTAS VEZES O NUMERO 1 SOZI-
        NHO FOI DIGITADO."
20  PRINT "DIGITE 0 PARA PARAR"
```

A afirmação 30 inicia o contador variável SOMA em zero:

```
30  SOMA = 0
```

Então, o número é coletado pelo teclado.

```
40  INPUT "DIGITE UM NUMERO:"; NUMERO
```

Se o número é 0, então a instrução 50 é executada:

```
50  IF NUMERO = 0 THEN 100
```

onde 100 é a instrução: END. Vamos supor que o número era 10 e ver o que acontece:

```
60  IF NUMERO < > 1 THEN GOTO 40
```

Se o número não é 1, nós saltamos de volta para a linha 40 e requisitamos um novo número. Se o número é 1, fazemos:

```
70  SOMA = SOMA + 1
```

A variável contador SOMA é incrementada de um. Recorde o significado de uma instrução designativa. Você pode ler na linha 70:

SOMA recebe o novo valor de (valor antigo de SOMA) + 1

Até este ponto, SOMA recebe o valor 0 + 1 = 1. A instrução seguinte é:

```
80  PRINT "UM. TOTAL ATE AGORA:"; SOMA
```

Então o programa salta de volta para a linha 40, requisitando um novo número:

```
90  GOTO 40
```

Reavaliando o Exercício de Aritmética

Recorde que desenvolvemos um exercício aritmético no início deste capítulo. Lamentamos o fato de que ele era muito simples e não tenha podido reciclá-lo. No entanto, podemos agora fazê-lo.

Como o programa é bastante grande, vamos olhar apenas os segmentos relevantes. Primeiro, aqui está a seção que pede ao usuário para selecionar um número entre 1 e 4:

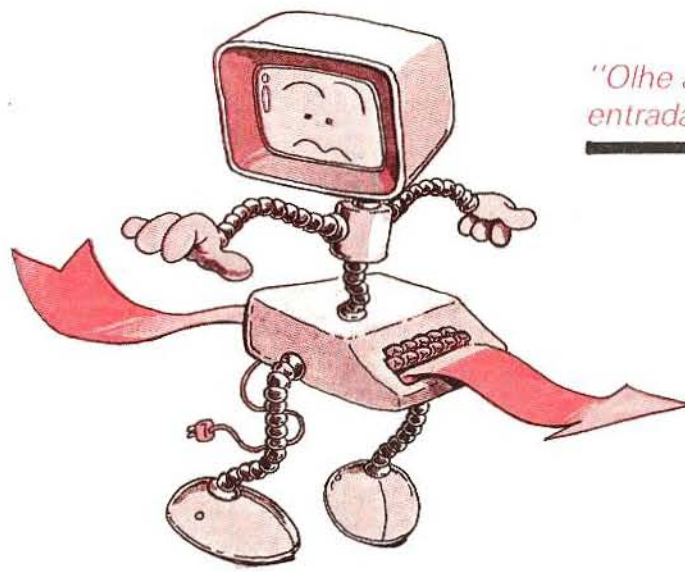
```
90   INPUT "QUAL E SUA ESCOLHA:"; ESCOLHA
100  IF (ESCOLHA = 1) THEN 200
110  IF (ESCOLHA = 2) THEN 300
120  IF (ESCOLHA = 3) THEN 400
130  IF (ESCOLHA = 4) THEN 500
140  PRINT "ESCOLHA INCORRETA. VOCE DEVE SELECIONAR UM
          NUMERO ENTRE 1 E 4"
150  PRINT "ATE LOGO": END
```

E aqui está o aperfeiçoamento:

```
150  GOTO 90
```

isto é tudo. Verifique:

Agora queremos que o programa faça mais do que uma pergunta sobre aritmética. Dizemos que queremos fazer a ele 10 perguntas diferentes. Podemos fazer isto adicionando GOTOs e um contador.



"Olhe a minha entrada, por favor."

Validação das Entradas

O exemplo que acabamos de examinar mostra uma regra importante de projeto de programa. Sempre que você pede dados ao teclado, não assuma que eles sejam sempre supridos corretamente. Um usuário pode digitar a tecla errada, deliberada ou acidentalmente. Para evitar erros de programa, faça sempre uma *validação das entradas*. Se a digitada num teclado não é válida, gere uma mensagem polida e requisite novamente o INPUT. Nós executaremos a validação das entradas na maioria de nossos exemplos. Vamos agora desenvolver dois programas completos, que tomem decisões.

A Conversão de Unidades de Medição

No capítulo 3 aprendemos a realizar uma simples conversão de milhas em quilômetros. Aqui está a maneira de automatizá-la:

```
10   REM * * * CONVERSAO DE MILHAS * * *
20   REM
30   PRINT "CONVERTO MILHAS EM QUILOMETROS"
40   PRINT "DIGITE 0 PARA PARAR"
50   INPUT "QUANTAS MILHAS"; MILHAS
60   IF MILHAS = 0 GOTO 100
70   KM = MILHAS * 1.6
80   PRINT MILHAS; "MILHAS = "; KM; "QUILOMETROS"
90   GOTO 50
100  END
```

Aqui está o resultado:

```
CONVERTO MILHAS EM QUILOMETROS
DIGITE 0 PARA PARAR
QUANTAS MILHAS? 7
    7 MILHAS = 11.2 QUILOMETROS
QUANTAS MILHAS? 10
    10 MILHAS = 16 QUILOMETROS
QUANTAS MILHAS? 0
```

O Cálculo da Idade

Aqui está mais um exemplo. Vamos melhorar nosso programa inicial que computava a idade de uma pessoa. Você suprirá a data de hoje e o dia, mês e ano em que você nasceu e o programa lhe dirá sua idade exata. Aqui está o programa.

```
10  REM *** CALCULO DE IDADE ***
20  INPUT "QUAL E SEU PRIMEIRO NOME?"; PRIMEIRO$
30  PRINT "ALO"; PRIMEIRO$; "EU CALCULAREI SUA IDADE"
40  PRINT "QUAL E A DATA DE HOJE? (AA/MM/DD)"
50  INPUT "O PRIMEIRO ANO (2 DIGITOS):"; AA
60  IF (AA<0 OR AA>99) THEN 50
70  INPUT "MES (DE 1 A 12):"; MM
80  IF (MM<1 OR MM>12) THEN 70
90  INPUT "O DIA:"; DD
100 IF (DD<1 OR DD>31) THEN 90
110                                                    REM
120 PRINT "AGORA DE ME SUA DATA DE NASCIMENTO"
130 INPUT "ANO (2 DIGITOS):"; ANASC
140 IF (ANASC< 0 OR ANASC> 99 ) THEN 130
150 INPUT "MES (DE 1 A 12):"; MNASC
160 IF (MNASC <1 OR MNASC>12) THEN 150
170 INPUT "DIA:"; DNASC
180 IF (DNASC <1 OR DNASC>31) THEN 170
190                                                    REM
200 REM-----CALCULO DA IDADE-----
210 IF MNASC<MM THEN 270
220 IF MNASC>MM THEN 320
230 REM ----- DATA DE ANIVERSARIO -----
240 IF DNASC<DD THEN 270
250 IF DNASC>DD THEN 320
260 PRINT "HOJE E SEU ANIVERSARIO. PARABENS"
270 IDADE = AA - ANASC
280 PRINT "VOCE TEM"; IDADE; "ANOS DE IDADE"
290 END
300                                                    REM
310 REM O DIA DO MEU ANIVERSARIO AINDA NAO CHEGOU ESTE
    ANO
320 IDADE = AA - ANASC - 1
330 GOTO 280
340 END
```

Apesar do tamanho, este programa é muito simples. Note como reafirmamos cada entrada. No entanto, para manter o programa curto, nossas validações são toscas. Nós não verificamos se cada número é

um número inteiro, assim como não verificamos o número de dias de cada mês. Este é um ótimo exercício para o leitor.

Aqui está uma maneira de verificar se M é igual a um número inteiro entre 1 e 12:

```
IF NOT (MM = 1 OR MM = 2 OR ... MM = 12) THEN 70
```

Sumário

Usando as instruções IF e GOTO, aprendemos a escrever programas para realizar testes sobre valores e tomar decisões. Nós aprendemos ainda como aperfeiçoar laços de programa (*loops*) onde cada parte do programa pode ser repetida indefinidamente. Além disso aprendemos a, sistematicamente, verificar e validar as entradas digitadas no teclado. Nós já aprendemos até aqui toda a experiência básica necessária para escrever programas comuns e examinamos vários exemplos significativos. Agora, escreveremos nossos programas de uma forma mais conveniente.

Por causa da freqüência e da importância dos *loops* (laços de programa) e da automação nos programas, o BASIC oferece facilidades adicionais, na forma de instruções adicionais. Discutiremos essas facilidades no próximo capítulo.

Exercícios

6-1: Qual é o uso da instrução IF?

6-2: Qual é o resultado do programa:

```
10  INPUT RESPOSTA$
20  IF (RESPOSTA$ = "SIM") THEN PRINT "OBRIGADO"
30  IF (RESPOSTAS = "NAO") THEN PRINT "MUITO MAL"
40  PRINT "SIM OU NÃO" : GOTO 10
```

Se o resultado acima é ilógico, sugira um programa melhor.

6-3: As expressões a seguir são válidas?

a. A = 4
b. B = 2 OR C = 3
c. A>5
d. 5>A
e. 1>2
f. SOMA>NUMERO
g. LETRA$ = "A"

6-4: É válido o que se segue?

```
10  IF A = 5 THEN IF B = 2 THEN 18
```

6-5: O que é um salto ou laço de programa (*loop*)?

Automatizando

7

Usando as instruções IF e GOTO, podemos executar um segmento de programa repetidamente. O segmento de programa correspondente é chamado *loop* e a maioria dos programas usam *loops*. Neste capítulo aprenderemos técnicas aperfeiçoadas para a criação de *loop*. Também desenvolveremos programas sofisticados que automatizam tarefas.

Começaremos esse capítulo com uma revisão da técnica IF/GOTO para gerar um *loop*. Em seguida introduziremos uma nova instrução, e a FOR ... NEXT, que se destina a facilitar a criação de *loops*. Usaremos essa importante instrução intensivamente em nossos programas.

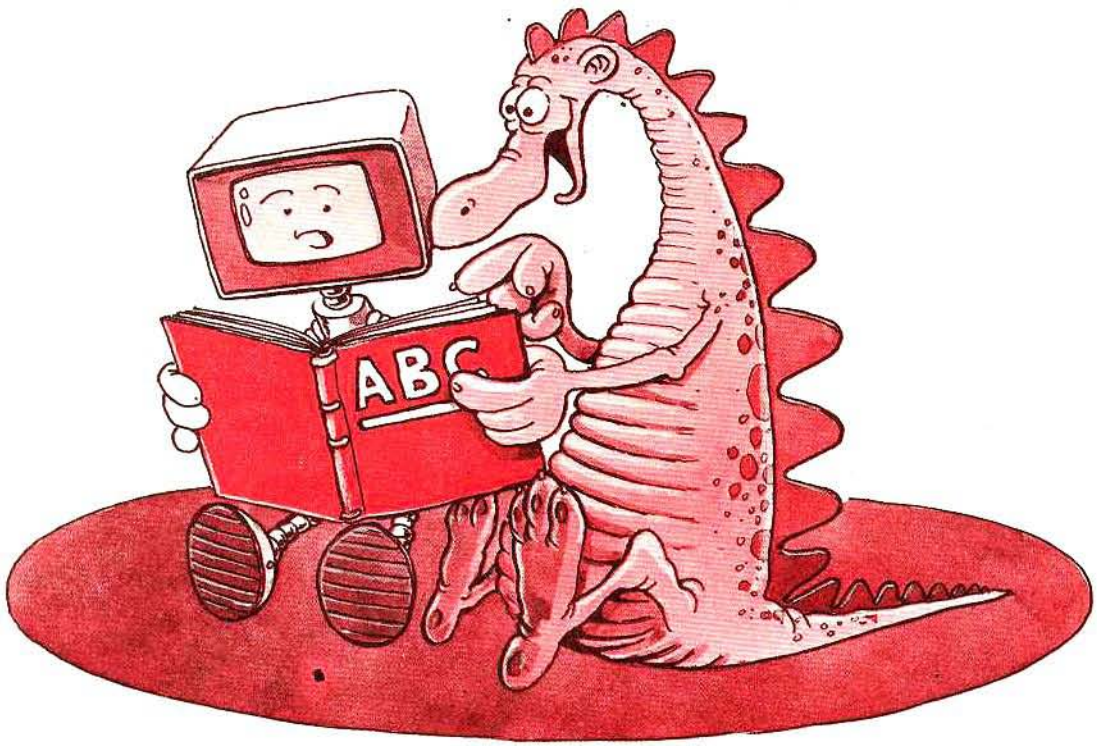
as Repetições

A Técnica IF/GOTO

Começaremos por examinar um programa que automatize um *loop*, usando a técnica IF/GOTO. À medida que examinarmos o programa identificaremos certas características que são comuns a todos os *loops*. Por exemplo, examinaremos o uso da variável de contagem, incrementação, iniciação e testes, antes da saída do *loop*. Aqui está o programa. Ele calcula a soma dos dez primeiros números inteiros.

```
1   REM *** SOMA DOS DEZ PRIMEIROS NUMEROS INTEIROS ***
10  SOMA = 0
20  I = 1
30  SOMA = SOMA + 1
40  I = I + 1
50  IF I = 11 THEN 70
60  GOTO 30
70  PRINT "A SOMA DOS DEZ PRIMEIROS NUMEROS INTEIROS E:";
        SOMA
80  END
```

"O que acha de fazermos alguma coisa mais complexa?"



Duas variáveis são utilizadas neste programa: SOMA e I. A variável SOMA acumula a soma dos 10 primeiros números inteiros, na medida em que nós os adicionamos — é o equivalente ao subtotal numa calculadora. I é o número inteiro que está sendo adicionado à SOMA.

Lembre-se de que uma variável deve ter um valor na primeira vez em que é usada. Então, antes de usarmos SOMA ou I numa fórmula, devemos estabelecer seus valores *iniciais* (0 e 1, respectivamente). Isto é obtido pelas instruções 10 e 20. Estas instruções são chamadas *instruções iniciais*.

A instrução seguinte é:

```
30  SOMA = SOMA + 1
```

"Lembra-se de mim?"

Esta instrução adiciona valor atual de I ao valor atual de SOMA. Quando esta instrução é executada pela primeira vez, o valor da SOMA é 0 e o valor de I é 1. Esta instrução determina o valor 0 + 1 = 1 para SOMA, como resultado. Após esta instrução, a variável SOMA contém o valor 1.

A instrução seguinte é:

```
40  I = I + 1
```

O valor atual de I é 1. O resultado desta instrução é dar a I o novo valor, 2. Esta é a técnica de contagem: I é incrementado por um, de forma a gerar o próximo número inteiro. Ao mesmo tempo, o valor de I indica quantos números inteiros foram somados até então. Em outras palavras, I é usado como o número inteiro atual e como um contador.

Assim, tudo o que temos a fazer é voltar à afirmação 30, e continuar a adicionar números inteiros:

"Peguei você novamente!"

```
50  GOTO 30
```

Errado. Este programa (em teoria) nunca pararia (realmente ele parará quando o valor de SOMA ficar maior que o máximo permitido por seu intérprete). Não é o que nós queremos. Queremos que o programa pare depois de executar o laço (*loop*) dez vezes. Devemos introduzir uma instrução *teste*. Aqui está ela:

```
50  IF I = 11 THEN 70
```

Quando I atingir o valor 11, a instrução 70 é executada e o programa pára. Isto se chama a *saída* do *loop*. Verifiquemos, agora, que o valor 11 (ao invés de 10) é realmente correto na instrução 50. Se escrevermos:

```
50  IF I = 10 THEN 70
```

Não vai funcionar. Quando I atingir o valor 10, SOMA contém a soma de 1 a 9 apenas. O *loop* deveria ser executado uma vez mais.

Lembre-se de que cada *loop* contém um contador. Você deve sempre verificar cuidadosamente o valor do contador que causa a saída do *loop*. No nosso exemplo, enquanto I não for igual a 11 o *loop* é mantido.

```
60  GOTO 30
```

Só quando I atingir o valor 11, é que os dez primeiros números terão sido adicionados. Tal fato ocorre porque em nosso programa, a adição (SOMA = SOMA + I) ocorre antes do incremento (I = I + 1). As duas instruções finais no programa permitem a *saída* do *loop:*

```
70  PRINT "A SOMA DOS DEZ PRIMEIROS NUMEROS INTEIROS E:";
    SOMA

80  END
```

Aqui está uma amostra da execução desse programa:

```
A SOMA DOS DEZ PRIMEIROS NUMEROS INTEIROS E: 55
```

A ilustração na figura 7.1 mostra o fluxo de controle no programa. Os números entre parênteses são os números das instruções:

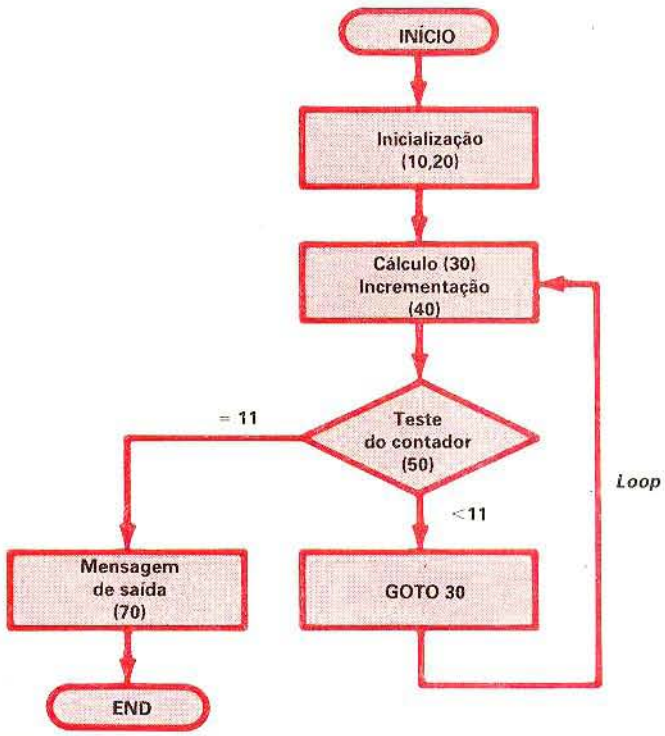


Figura 7.1 Fluxo de controle do programa da soma dos números inteiros

Este diagrama é chamado de *fluxograma*. Discutiremos o assunto dos fluxogramas em detalhes no capítulo 8. Por hora, simplesmente note a organização geral do programa: inicialização, cálculo mais incrementação, teste e saída. Todos os segmentos do programa com *loop* realizam essas funções.

Variações

Vamos agora brincar com nosso programa de soma de números inteiros e aprofundar nossos conhecimentos em programação. Isto demonstrará as várias alternativas que podem ser utilizadas para escrever um programa. Por exemplo, na linha 50 nós podíamos ter escrito:

```
50  IF I>10 THEN 70
```

e o resultado seria o mesmo (quando I chegasse ao valor 11 seria maior que 10). Também poderíamos ter escrito:

```
40  IF I = 10 THEN 70
50  I = I + 1
```

em lugar de:

```
40  I = I + 1
50  IF I = 11 THEN 70
```

Com esta mudança, I é testado primeiro e, depois, incrementado. Note que desta vez I é testado para o valor 10 (em vez de 11).

Poderíamos ainda ter escrito:

```
50 IF I<11 THEN 30
60  REM
```

Você pode verificar que estas versões são realmente corretas. Todas estas variações são válidas e equivalentes. Mesmo um programa curto como o programa SOMA pode ser escrito de várias maneiras equivalentes. Não há uma única maneira de escrever um programa. Assim como numa linguagem falada, você pode expressar o mesmo conceito de várias maneiras diferentes.

Este curto programa ilustrou o uso do *loop* e da variável de contagem. Nós também examinamos as fases típicas envolvidas neste programa: inicialização, cálculo, incrementação, teste e saída. Em vista do uso freqüente de *loops* em programas, uma instrução especial foi criada para facilitar seu uso em BASIC. É a instrução FOR ... NEXT.

A Instrução FOR ... NEXT

A instrução FOR ... NEXT automatiza grande parte da programação necessária ao *loop*. Explanaremos seu uso e operação, usando exemplos reais.

Aqui está uma maneira de reescrever nosso programa de adição, usando estas novas instruções:

```
 1  REM *** ADICAO DE NUMEROS INTEIROS - 2.ª VERSAO***
10  SOMA = 0
20  FOR I = 1 A 10
30  SOMA = SOMA + I
40  NEXT I
50  PRINT "A SOMA DOS DEZ PRIMEIROS NUMEROS INTEIROS E:";
        SOMA
60  END
```

Note que este programa tem duas instruções a menos que o primeiro. Ele é mais curto e mais legível. Vamos examiná-lo em detalhes. A primeira instrução executada inicializa SOMA em zero:

```
10  SOMA = 0
```

A instrução seguinte é a instrução FOR:

```
20  FOR I = 1 A 10
```

Esta instrução tem várias funções:

- Ela marca o início do *loop* automático (é onde ele começa)
- Ela especifica que I (a variável de contagem) começa com o valor inicial 1, quando a instrução é executada pela primeira vez. Tal fato elimina a necessidade de uma instrução de inicialização para I.
- I é incrementado de 1 (até o valor máximo de 10) cada vez que a instrução é executada, a partir da instrução NEXT. Um teste automático é realizado e, quando I excede o valor 10, o *loop* não mais é executado e a instrução seguinte ao NEXT é executada em seu lugar (é a saída do *loop*).

O corpo do *loop* contém, simplesmente, a acumulação da soma:

```
30  SOMA = SOMA + I
```

A instrução NEXT:

```
40  NEXT I
```

marca o final do *loop* e causa a reativação do FOR. Isto substitui duas instruções da versão precedente:

```
40 I = I + 1
60  GOTO 30
```

Cada vez que NEXT é executado, o programa volta para o início do *loop*, isto é, a instrução FOR. Quando FOR é ativado:

- I é incrementado de 1
- O novo valor de I é automaticamente comparado a 10.

Enquanto I não exceder 10, a execução prossegue. O *loop* pára quando I é igual a 10 e o NEXT é alcançado. Neste ponto, ocorre a saída e a

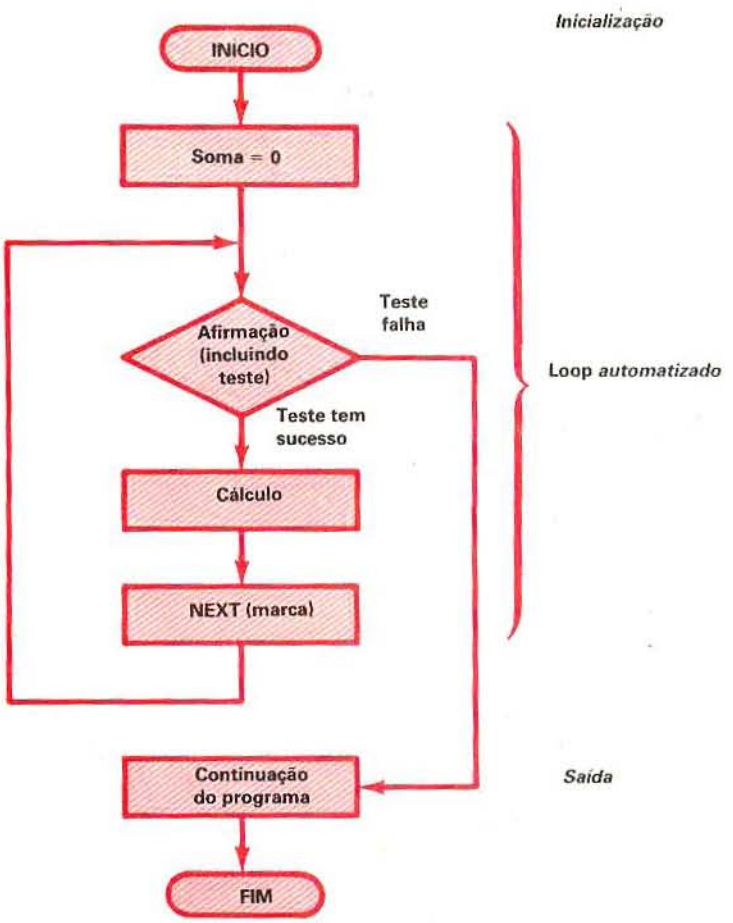


Figura 7.2 *Loop* automático com FOR ... NEXT

instrução 50 (seguinte ao NEXT) é executada. Esta seqüência é ilustrada na figura 7.2 (um fluxograma).

A figura 7.2 mostra que a instrução FOR automatiza três tarefas:

- Inicialização da variável de contagem (I é inicialmente alocado como 1)
- Incrementação da variável de contagem (I é incrementado de 1 a cada passagem do programa)
- Teste da variável e de contagem até o valor máximo (I é comparado a 10)

A instrução NEXT simplesmente marca o final do *loop* e cria uma instrução "GOTO para o FOR". Após usar a instrução FOR algumas vezes, você apreciará como ela simplifica o projeto do *loop* e clareia o programa.

FOR ... NEXT é uma instrução conveniente. Você não é obrigado a usá-la, mas provavelmente descobrirá seu valor. Geralmente, quanto mais claro um programa, menor será o risco de erros. Daremos agora exemplos práticos para ilustrar o uso da instrução FOR ... NEXT e o uso de *loops* automatizados.

Soma dos N Primeiros Números Inteiros

Nós calcularemos a soma dos N primeiros números inteiros. Desta vez, o usuário especifica o valor de N, no teclado. Aqui está o programa:

```
10  REM * SOMA DOS N PRIMEIROS NUMEROS INTEIROS *
20  SOMA = 0
30  INPUT "QUERO SOMAR OS N PRIMEIROS NUMEROS INTEIROS.
          DIGITE N:"; N
40  FOR I = 1 TO N
50  SOMA = SOMA + I
60  NEXT I
70  PRINT "A SOMA DOS PRIMEIROS"; N; "NUMEROS INTEIROS E";
          SOMA
80  END
```

Aqui está uma amostra da execução:

```
QUERO SOMAR OS N PRIMEIROS NUMEROS INTEIROS. DIGI-
TE N:? 5
A SOMA DOS 5 PRIMEIROS NUMEROS INTEIROS E 15
```

Você deve entender o programa facilmente. Desta vez nós pulamos de 1 para N onde N é fornecido pelo teclado (instrução 30). Você pode melhorar este programa validando a entrada: N deve ser maior que 1. O intérprete BASIC verificará automaticamente que N é um número inteiro, quando estiver executando a afirmação FOR. Tente enganá-lo.

Tabelas de Valores

Usando a poderosa instrução FOR ... NEXT nós mostraremos como é fácil automatizar cálculos e imprimir tabelas de valores. Aqui está uma tabela de quadrados (um número multiplicado por ele mesmo) e uma de cubos (um número multiplicado por ele mesmo e novamente multiplicado por ele mesmo):

```
10  REM TABELA DE QUADRADOS E CUBOS
20  REM PARA OS 10 PRIMEIROS NUMEROS INTEIROS
30  FOR I = 1 A 10
40  PRINT I, I ^ 2, I ^ 3
50  NEXT I
60  END
```

Aqui está o resultado:

```
1     1     1
2     4     8
3     9     27
4     16    64
5     25    125
6     36    216
7     49    343
8     64    512
9     81    729
10    100   1000
```

Vamos olhar mais de perto a instrução 40:

```
40  PRINT I, I ^ 2, I ^ 3
```

I ^ 2 representa I na potência 2, isto é, I * I. Por exemplo, se I = 2, então I ^ 2 = 2 × 2 = 4. Similarmente I ^ 3 representa I na potência 3, isto é, I*I*I. Se I = 4, então I ^ 3 = 4 × 4 × 4 = 64.

Note que usamos uma vírgula desta vez, na instrução PRINT, então os resultados são cuidadosamente alinhados, quando listados. A vírgula força um espaço automático (ou *tabulação*) na linha. O espaço exato entre as colunas depende do seu intérprete BASIC. Por exemplo: pode ser de 14 caracteres.

Como exercício, você pode reescrever este programa, para executar a soma dos quadrados e cubos dos N primeiros números inteiros, onde N é lido no teclado. Aprendemos a fazer isto na seção anterior.

Linhas de Asteriscos

Aqui está um programa simples, que imprime N linhas de asteriscos onde N é um número especificado no teclado:

```
10  REM * LINHAS DE ASTERISCOS *
20  PRINT "QUERO MOSTRAR LINHAS DE ASTERISCOS"
30  INPUT "DIGA ME QUANTAS LINHAS : " ; N
40  REM N E O NUMERO DE LINHAS A SER MOSTRADO
50  FOR I = 1 TO N
60  PRINT "******************************"
70  NEXT I
80  END
```

Aqui está uma amostra da execução:

```
QUERO MOSTRAR LINHAS DE ASTERISCOS
DIGA ME QUANTAS LINHAS: ? 6
**********
**********
**********
**********
**********
**********
```

Novamente, a cada vez que uma entrada é suprida pelo usuário, é uma boa idéia validá-la para evitar erros de programação. Esperamos que quem fornece os dados ao programa, forneça um número positivo. Vamos admitir que você não queira mais do que 20 linhas de asteriscos. Deveria então dizê-lo ao usuário numa instrução PRINT apropriada e usar uma instrução de validação como:

```
IF  (N < 1) OR (N > 20) GOTO 20
```

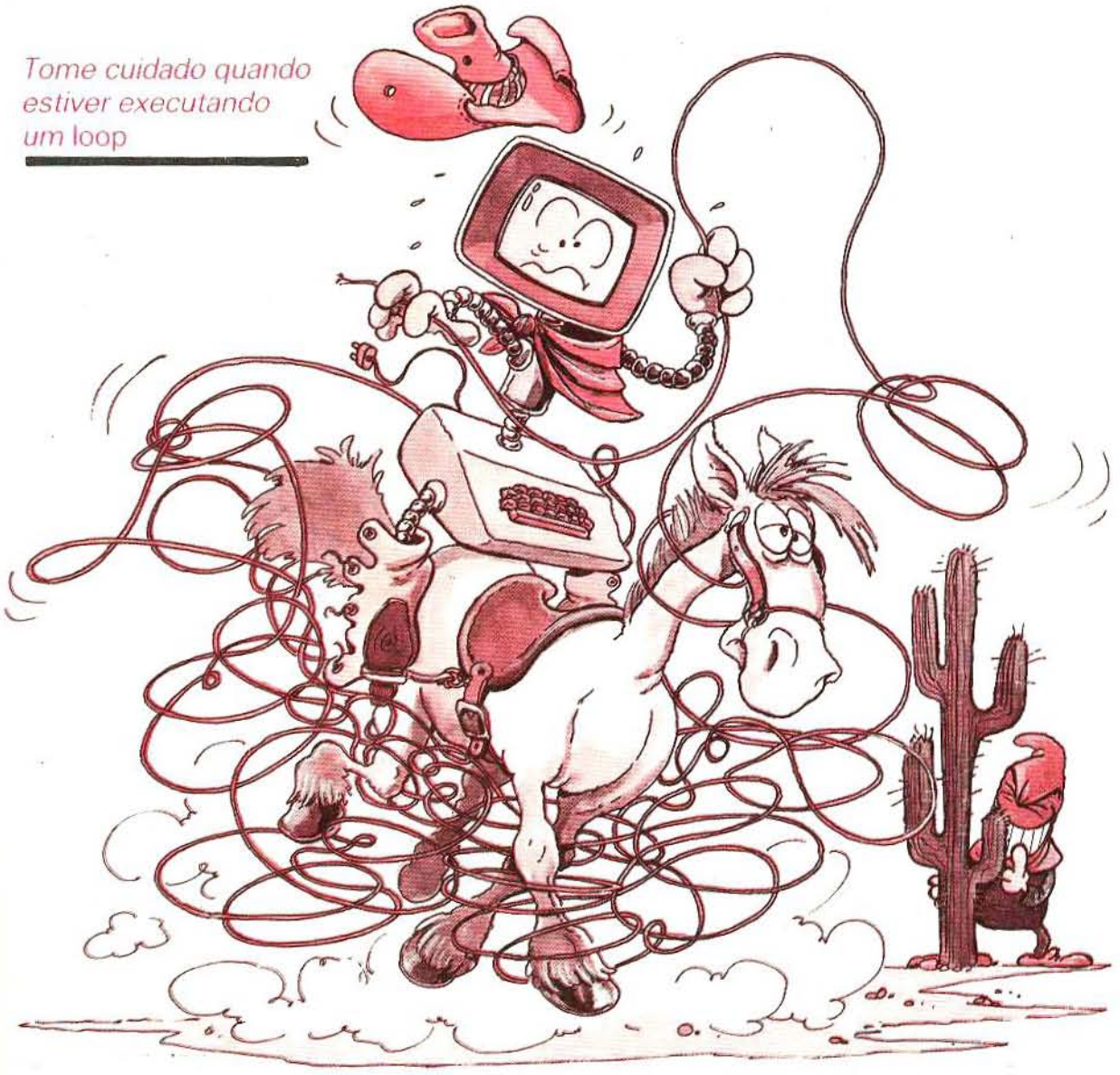
Laços de Programa mais Elaborados

A instrução FOR ... NEXT oferece dois recursos avançados que nós ainda não descrevemos:

- Você pode incrementar o contador em qualquer valor inteiro, como 2, 3, 4 ou até − 1 (ao invés de incrementar de 1). Isto é chamado de característica de *passo variável*.
- Você pode criar um *loop* dentro de outro *loop*. Estes *loops* são chamados de *loops encadeados* ou *loops seriados*.

Vamos examinar estas duas técnicas.

Tome cuidado quando estiver executando um loop



Passo Variável

Aqui está um exemplo de passo variável:

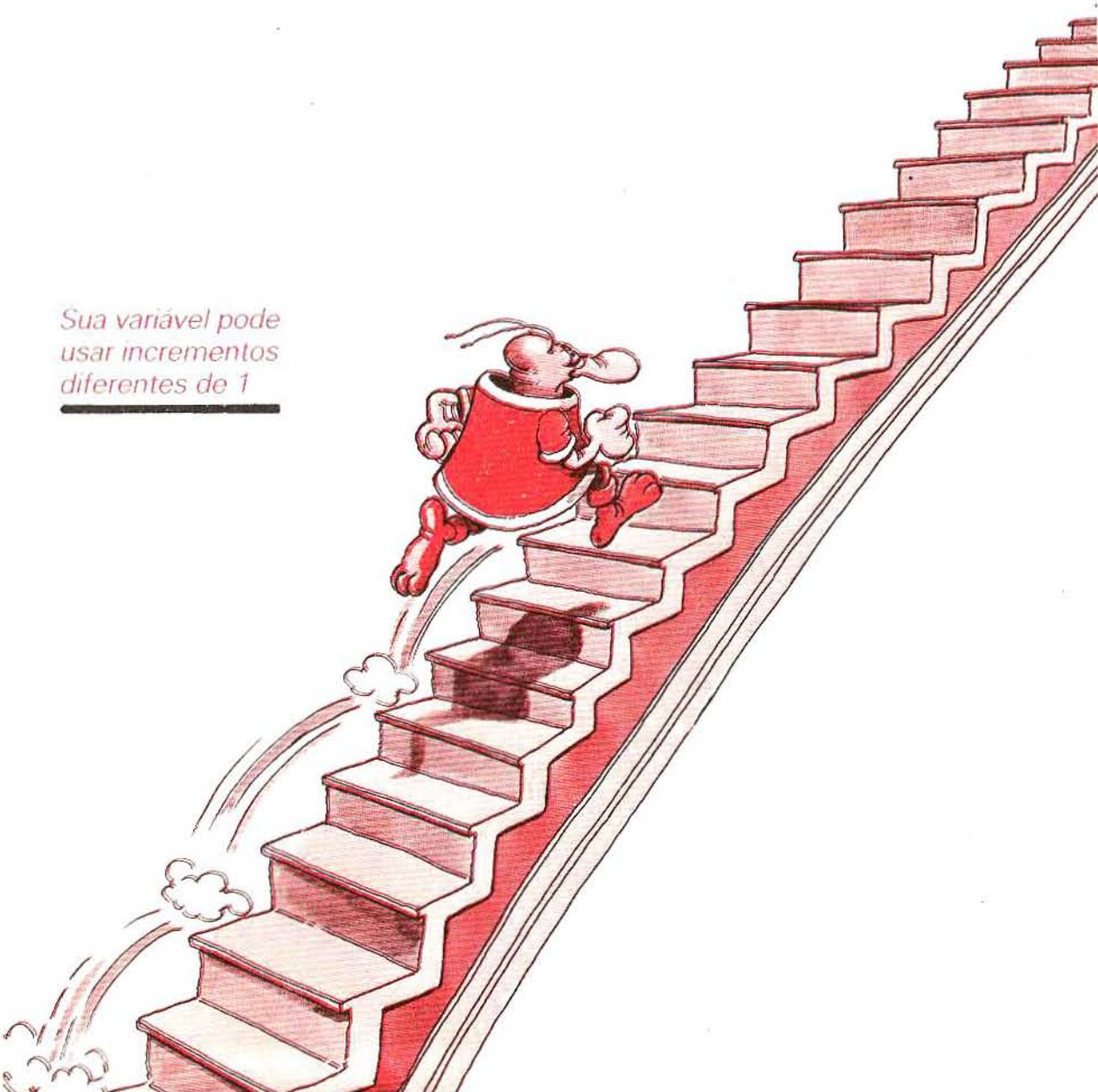
```
FOR I = 1 TO 5 STEP ?
```

Cada vez que o *loop* é executado I será incrementado de 2. Você poderá ainda escrever:

```
FOR I = 10 TO – 5 STEP – 1
```

usando um incremento de *passo negativo*. Pelo fato de o limite superior da variável de contagem (–5) ser menor do que o valor inicial (10), ele é olhado como um "passo negativo" pelo interprete. O valor de I será decrescido de 1 a cada vez. O primeiro valor de I será 10. O

Sua variável pode usar incrementos diferentes de 1



próximo será 9, o seguinte 8, etc. O último será −5. Em outras palavras, I tomará os seguintes valores em ordem: 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, −1, −2, −3, −4, −5. O passo negativo é uma outra característica que você pode querer usar.

Loops Encadeados (seriados)

A técnica de *loops* encadeados é um recurso importante e poderoso, usado para automatizar processamentos complexos. Um *loop* encadeado é criado sempre que você usa um grupo de instruções FOR ... NEXT dentro de um *loop*, isto é, sempre que você usa um outro grupo de instruções FOR ... NEXT.

Em geral, você pode usar quantas instruções quiser entre as instruções FOR ... NEXT. Especificamente, você pode ainda incluir um outro *loop* no limite destas instruções. Este conceito é ilustrado na figura 7.3.

Quando usar o *loop* encadeado, note como o programa se torna mais difícil de ler. Para remediar tal fato, você será encorajado a usar a *denteação*. Denteação é uma outra técnica de clarear o programa. A figura 7.4 mostra a versão denteada do programa da figura 7.3.

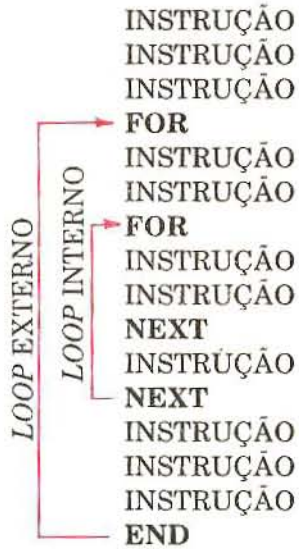
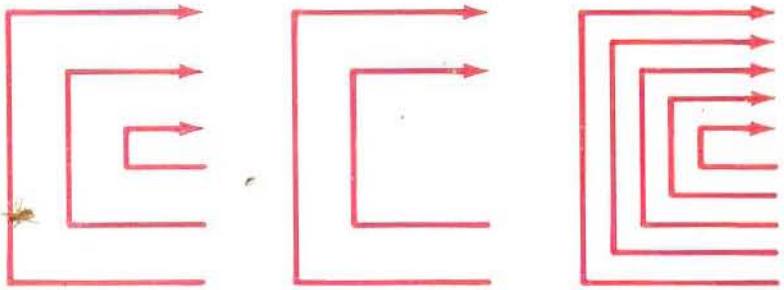


Figura 7.3 Um *loop* seriado

```
INSTRUÇÃO
INSTRUÇÃO
INSTRUÇÃO
FOR
  INSTRUÇÃO
  INSTRUÇÃO
  FOR
    INSTRUÇÃO
    INSTRUÇÃO
  NEXT
  INSTRUÇÃO
NEXT
INSTRUÇÃO
INSTRUÇÃO
INSTRUÇÃO
END
```

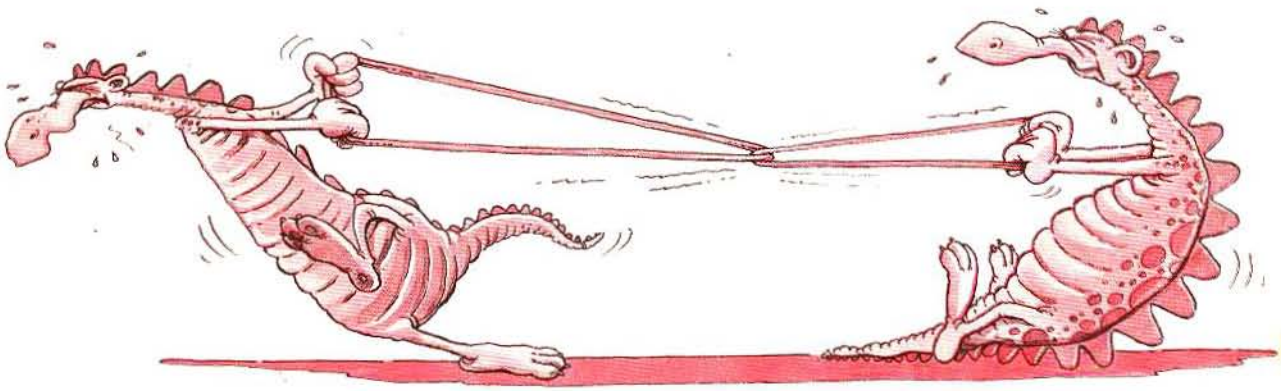
Figura 7.4 Um programa denteado

Você pode encadear *loops* a qualquer nível, até um número máximo, dependendo de seu intérprete ou da capacidade de memória disponível. No entanto, você não pode sobrepor *loops*. Os *loops* seguintes são legais:



Os seguintes não são:

Não sobreponha loops!

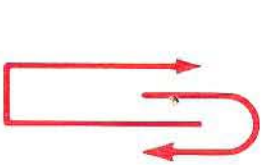


Em suma, você não pode saltar (isto é, especificar um GOTO) de um ponto *dentro* do *loop* externo para um ponto dentro do interno:

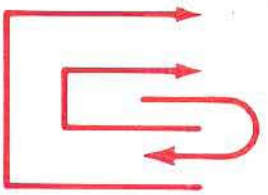
Encadeamento correto

Salto ilegal (com IF ou GOTO)

No entanto, você pode saltar para fora do *loop* interno:



Salto correto



Salto correto

Aqui está um exemplo de *loop* encadeado. Este programa mostra o tempo em minutos e horas:

```
10  REM *** RELOGIO SIMULADO ***
20  FOR HORA = 0 TO 23
30      FOR MINUTOS = 0 TO 59
40          PRINT "O TEMPO E"; HORA; " HORAS E';
            MINUTO; "MINUTOS"
50      NEXT MINUTO
60  NEXT HORA
70  PRINT "FINAL DO DIA"
80  END
```

Aqui está uma parte da execução:

```
O TEMPO E 0 HORA E 0 MINUTOS
O TEMPO E 0 HORA E 1 MINUTO
O TEMPO E 0 HORA E 2 MINUTOS
O TEMPO E 0 HORA E 3 MINUTOS
O TEMPO E 0 HORA E 4 MINUTOS



O TEMPO E 0 HORA E 59 MINUTOS
O TEMPO E 1 HORA E 0 MINUTOS
```

Características Adicionais

Como informação final, valores decimais e expressões são geralmente permitidas na instrução FOR. Por exemplo, o que se segue é válido:

```
FOR MEDIDA = 0.1 TO 13.5 STEP 0.2
FOR NUMERO = N TO (N * 2) STEP 1
```

Esta prática não é recomendada, e nós não usaremos aqui estas características avançadas.

Sumário

Loops são muito utilizados para automatizar repetições de um segmento de programa. A instrução FOR ... NEXT é utilizada para automatizar *loops* em BASIC. Na maioria dos casos, a instrução FOR ... NEXT pode substituir diversas outras instruções BASIC. Neste capítulo, examinamos usos típicos para a instrução FOR ... NEXT, incluindo *loops* encadeados, e desenvolvemos vários programas avançados. Agora que aprendemos todas as técnicas básicas de programação, você está quase pronto a começar a escrever seus próprios programas. No próximo capítulo, explicaremos como você pode começar.

Exercícios

7-1: Mostre os 15 primeiros números inteiros em uma linha (4 instruções).

7-2: Escreva um programa que leia o tempo em horas e minutos no teclado, e mostre-o como se segue:

INPUT:	3 *(horas)*,	*31 (minutos)*
Mostre:	H H H	*(3 letras)*
	M M M	*(3 letras)*
	M	*(1 letra)*

(Sugestão: Você pode executar PRINT LETRA$; repetidamente para imprimir vários caracteres).

7-3: Qual é a variável de contagem num *loop*?

7-4: Você pode saltar para o meio do *loop*?

7-5: Mostre uma tabela que converta onças em gramas (1 onça = 28 gramas).

7-6: Calcule a soma dos N primeiros números ímpares inteiros — onde N é fornecido pelo teclado — e mostre-o para cada número inteiro.

7-7: Leia as notas de 5 estudantes que fizeram 4 testes, com gradação de 0 a 10. Mostre as notas, depois o total e a média de cada um.

7-8: Mostre uma tabela de impostos de venda para preços de Cr$ 1 a Cr$ 100 com Cr$ 1 de incremento. Forneça a taxa do imposto pelo teclado.

Criando ur

Programa

8

Programar envolve escrever um programa que automatize uma tarefa. Até agora escrevemos vários programas curtos. Fizemos isto sem nenhuma etapa intermediária, escrevendo diretamente uma seqüência de instruções BASIC. Esta técnica é boa para programas muito simples, mas não funciona bem para os mais complexos.

Neste capítulo aprenderemos a maneira correta de criar um programa. Este é um processo de cinco etapas.

1. Especifique a seqüência de etapas envolvidas na solução do problema. Isto é chamado de *projeto de algoritmo.*
2. Desenhe um diagrama mostrando a seqüência de eventos e etapas lógicas. Isto é chamado de *desenho do fluxograma.*
3. Escreva o programa em BASIC. Isto é chamado de *codificação.*
4. Verifique e teste o programa. Isto é chamado de *correção de erros.*
5. Clareie e documente o programa. Isto é chamado de *documentação.*

Até agora aprendemos e praticamos apenas as etapas 2 e 5. Mas esta seqüência só funcionará para programas curtos. Antes de escrever programas longos, vamos estudar a seqüência completa envolvida no desenvolvimento de um programa.

Projeto do Algoritmo

Nós queremos escrever um programa que resolva um dado problema ou automatize uma tarefa. Até agora projetamos programas para resolver problemas simples. A seqüência de etapas requeridas para resolver cada problema foi, geralmente, óbvia, em que, praticamente, não havia uma etapa de projeto. De maneira geral, no entanto, dado um problema, devemos, antes de tudo, projetar uma solução. Para escrever programas, esta solução deve ser especificada como uma seqüência de etapas. Esta seqüência de etapas é chamada de *algoritmo*. Formalmente, um algoritmo é definido como uma especificação passo a passo, para a solução de um problema. Tecnicamente, um algoritmo também deve ter um fim, ou seja, ele não pode continuar indefinidamente. Um algoritmo que não pára é chamado de *um erro*!

Aqui está um problema simples. Vamos converter um peso medido em onças para seu equivalente em gramas. Lembre-se que 1 onça é equivalente a 28,35 gramas. A solução é óbvia: multiplicamos o peso em onças por 28,35 gramas. Este é um algoritmo de apenas um passo.

Vamos agora examinar um problema ligeiramente mais complexo: vamos ler um número no teclado e verificar se ele está dentro de determinados limites. Aceitaremos o número como válido se estiver entre 0 e 100. A seqüência de etapas envolvidas na solução deste problema é a seguinte:

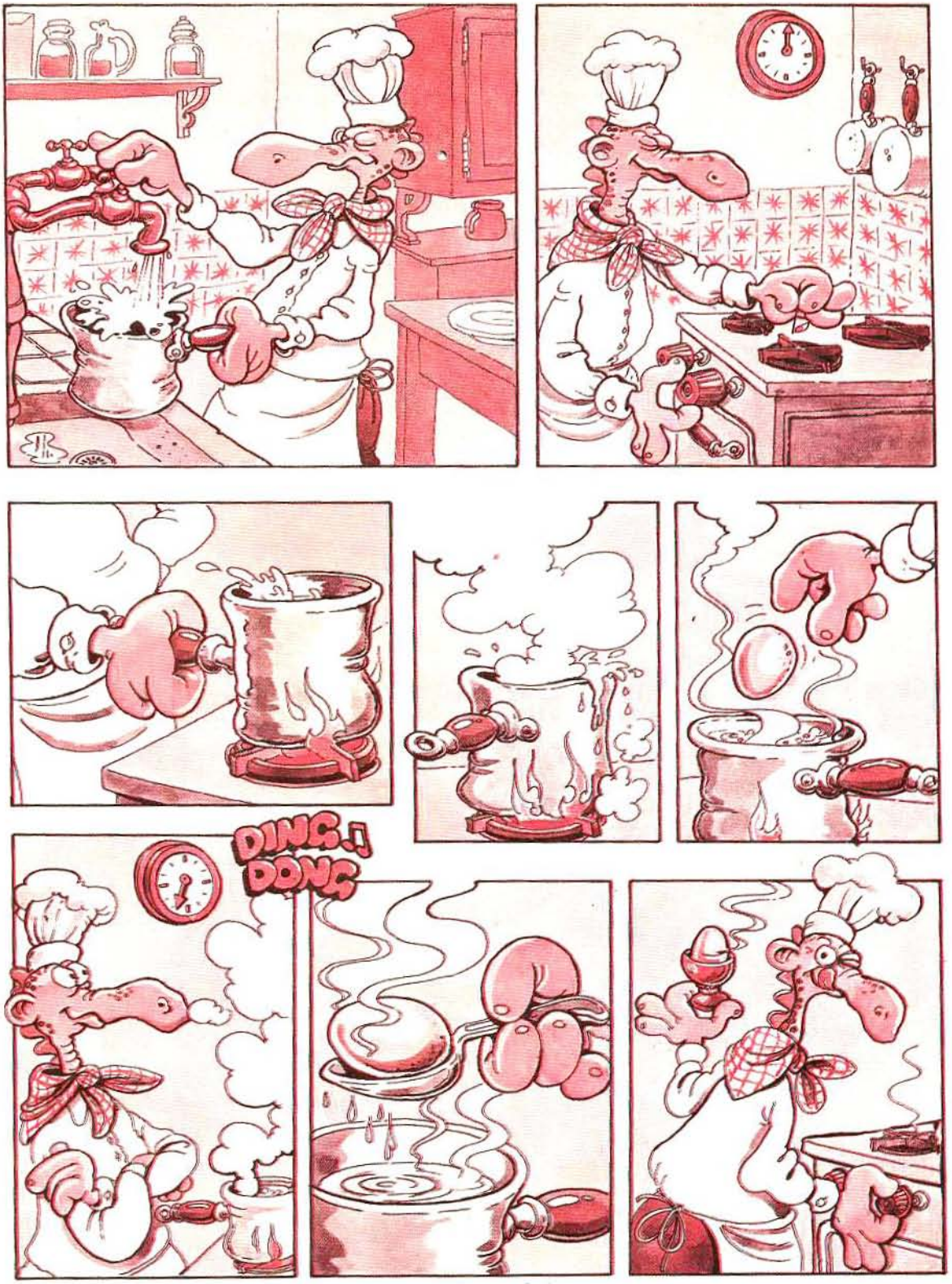
1. Ler o número.
2. Verificar se ele é maior do que zero. Se for, prosseguir; senão, rejeitar o número.
3. Verificar se ele é menor do que 100. Se for, aceitá-lo; senão, rejeitá-lo.

Este é um algoritmo de 3 etapas.

Na prática, a maioria dos problemas é mais complicada e sua solução requer algoritmo maiores e mais complexos. Aqui estão alguns exemplos de algoritmos do dia a dia. Você pode encontrar muito mais no seu livro de cozinha, no seu carro ou nos manuais dos seus eletrodomésticos.

Vamos examinar um algoritmo para cozinhar um ovo-de-três-minutos. Aqui estão as etapas:

1. Pegue uma panela
2. Encha-a de água
3. Ligue o fogão
4. Coloque a panela no fogão
5. Deixe a água ferver
6. Coloque um ovo na água fervendo
7. Marque 3 minutos no relógio
8. Quando o despertador tocar, retire o ovo
9. Desligue o fogão.



"Demonstrarei o algoritmo do ovo-de-três-minutos."

Este algoritmo parece direto, mas se ele fosse executado como um programa por um robô computadorizado, ele teria que ser muito mais preciso. Por exemplo, precisaríamos especificar exatamente qual panela usar, e a quantidade exata de água a ser colocada na panela.

A maioria dos algoritmos apresentados nos livros comuns assumem que o usuário tenha um conhecimento anterior técnico e cultural específico: e eles são, na maioria das vezes, incompletos. Em outras palavras, eles assumem que o usuário possa "ler nas entrelinhas". Por isso é que existem manuais tão difíceis de compreender!

Não cometeremos o mesmo erro aqui. Nossos algoritmos serão completamente especificados para tornarem-se programas utilizáveis.

Aqui está um último exemplo: um algoritmo para dar partida em um carro. Se assumirmos que o carro funciona perfeitamente, o algoritmo é muito simples:

1. Inserir a chave na ignição.
2. Girar totalmente a chave para a direita.
3. Soltar a chave enquanto aperta suavemente o pedal do acelerador.

No entanto, sabemos que é possível que o carro não ligue. Isto porque há outros fatores envolvidos, como temperatura ou as condições mecânicas da máquina. Preparar um algoritmo completo para ligar um carro sob qualquer condição exigirá várias páginas, caso se queira prever todas as coisas que podem sair erradas.

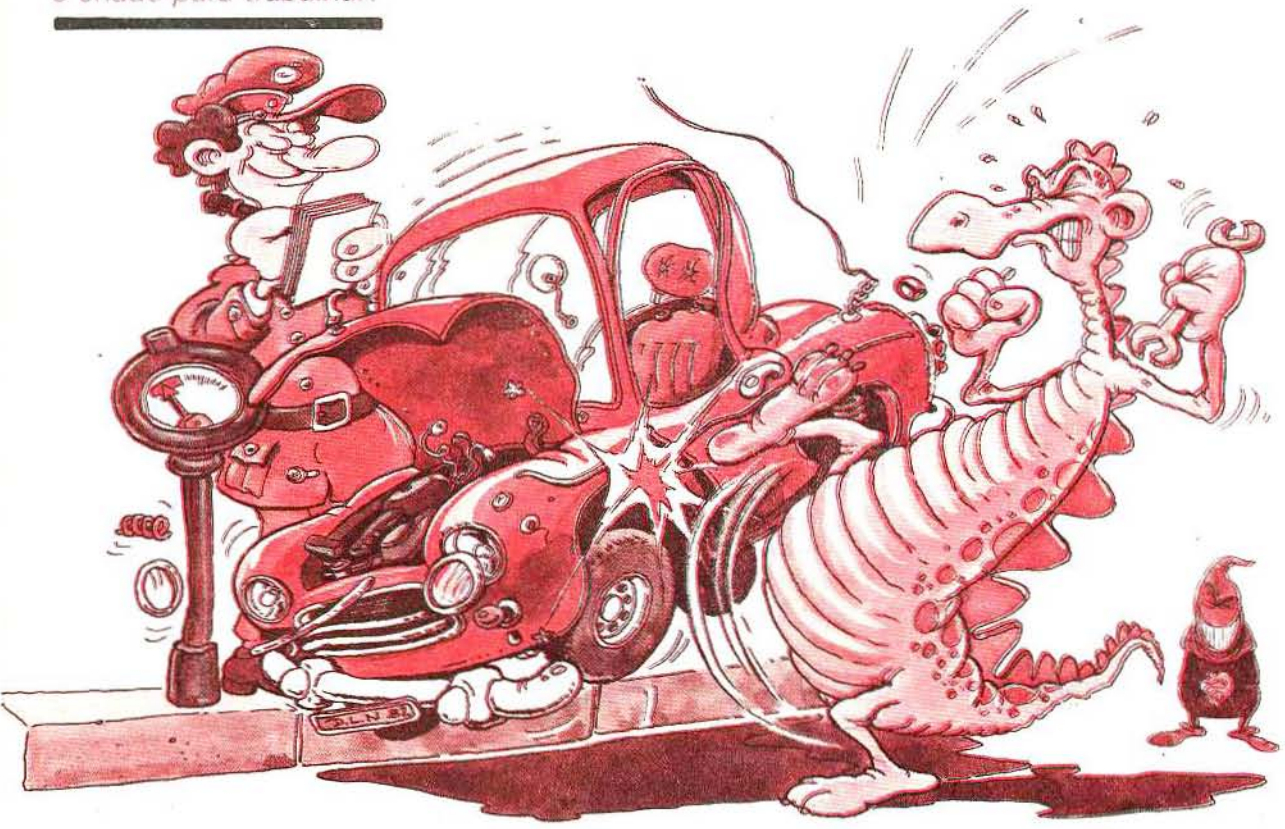
Ligar um carro é um interessante algoritmo

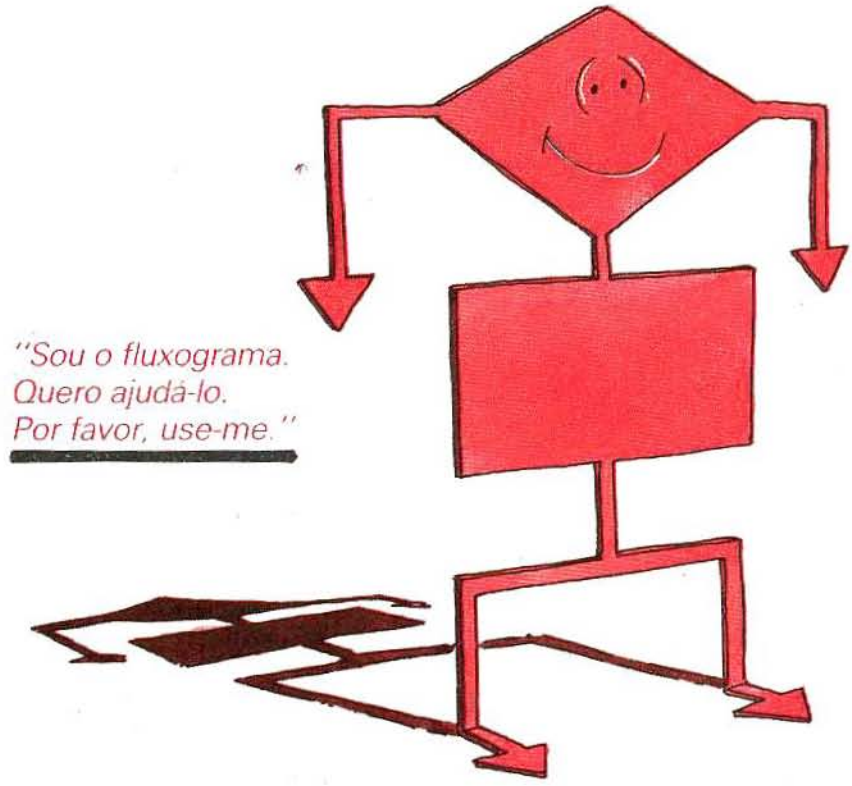
No dia a dia podemos, algumas vezes, simplificar as etapas de um algoritmo. Mas num programa de computador, isto é impossível. Um algoritmo tem que ser correto (exato) e completo. Quando projetamos um algoritmo para uma solução de computador, devemos ser completos e antecipar cada fato razoável que possa acontecer, ou nosso programa pode, eventualmente, falhar. O sucesso de um programa requer uma atitude especial: você deve continuamente duvidar do que faz, sempre supondo que possa estar errado ou incompleto.

Nunca assuma que uma entrada esteja certa. Cheque-a e verifique-a. Sempre admita a possibilidade de erros. Ilustraremos estas considerações mais tarde, neste capítulo, quando examinarmos um caso real.

Em suma, para automatizar a solução de um problema, escrevendo um programa de computador, comece preparando um algoritmo. O algoritmo final deve ser perfeito, apesar de que, no começo, ele raramente o é. De fato, no início, você provavelmente projetará uma solução aproximada (isto é, um algoritmo tosco), então continuará a aperfeiçoar o algoritmo o quanto quiser, até que ele alcance o estado em que você o considerará perfeito. Tenha sempre certeza de que seus algoritmos estão completos, antes de começar, efetivamente, a escrever as instruções.

Lembre-se, o algoritmo é criado para trabalhar!





Fluxograma

Assim, projetamos um algoritmo. O pensamento que agora vem à mente é: Vamos traduzi-lo rapidamente em um programa BASIC e executá-lo. Errado. Há ainda mais uma etapa que pode economizar horas do tempo de programação: é o chamado *fluxograma*. Se você omitir este passo, provavelmente escreverá um programa que não funcionará, e perderá muito mais tempo, mais tarde, tentando corrigi-lo — com pouca garantia de sucesso. Em contrapartida, se você fizer um bom fluxograma, escrever um programa é apenas um simples passo a mais.

Um fluxograma é simplesmente uma representação gráfica, mostrando uma seqüência de eventos. A figura 8.1 mostra um fluxograma das etapas envolvidas em "cozinhar um ovo-de-três-minutos".

Como você pode ver, este fluxograma é uma representação gráfica do algoritmo que já apresentamos. Neste caso, cada retângulo do fluxograma representa um passo do algoritmo. O propósito do fluxograma é mostrar toda a seqüência de etapas. Mais tarde, quando você se familiarizar com o fluxograma, você poderá omitir a etapa do projeto do algoritmo e começar diretamente com o fluxograma, já que o fluxograma é, exatamente, a representação do algoritmo.

Figura 8.1
Cozinhando um ovo-de-três-minutos

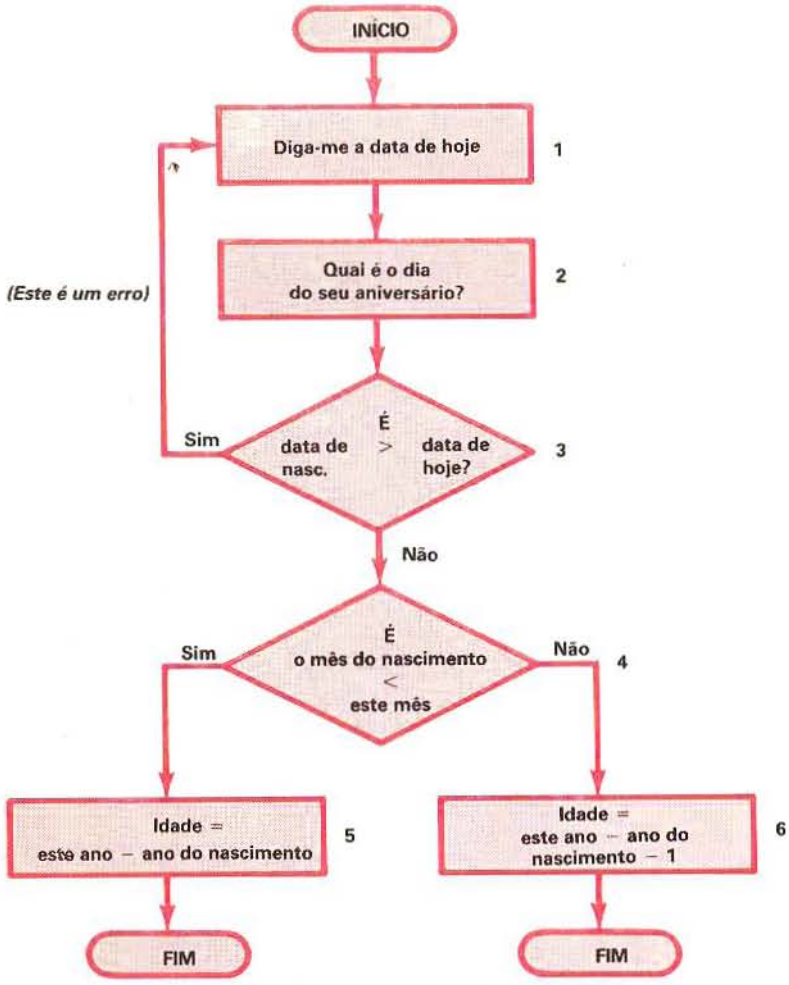


Figura 8.2 Fluxograma para cálculo da idade

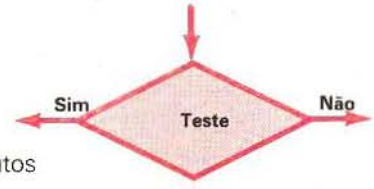


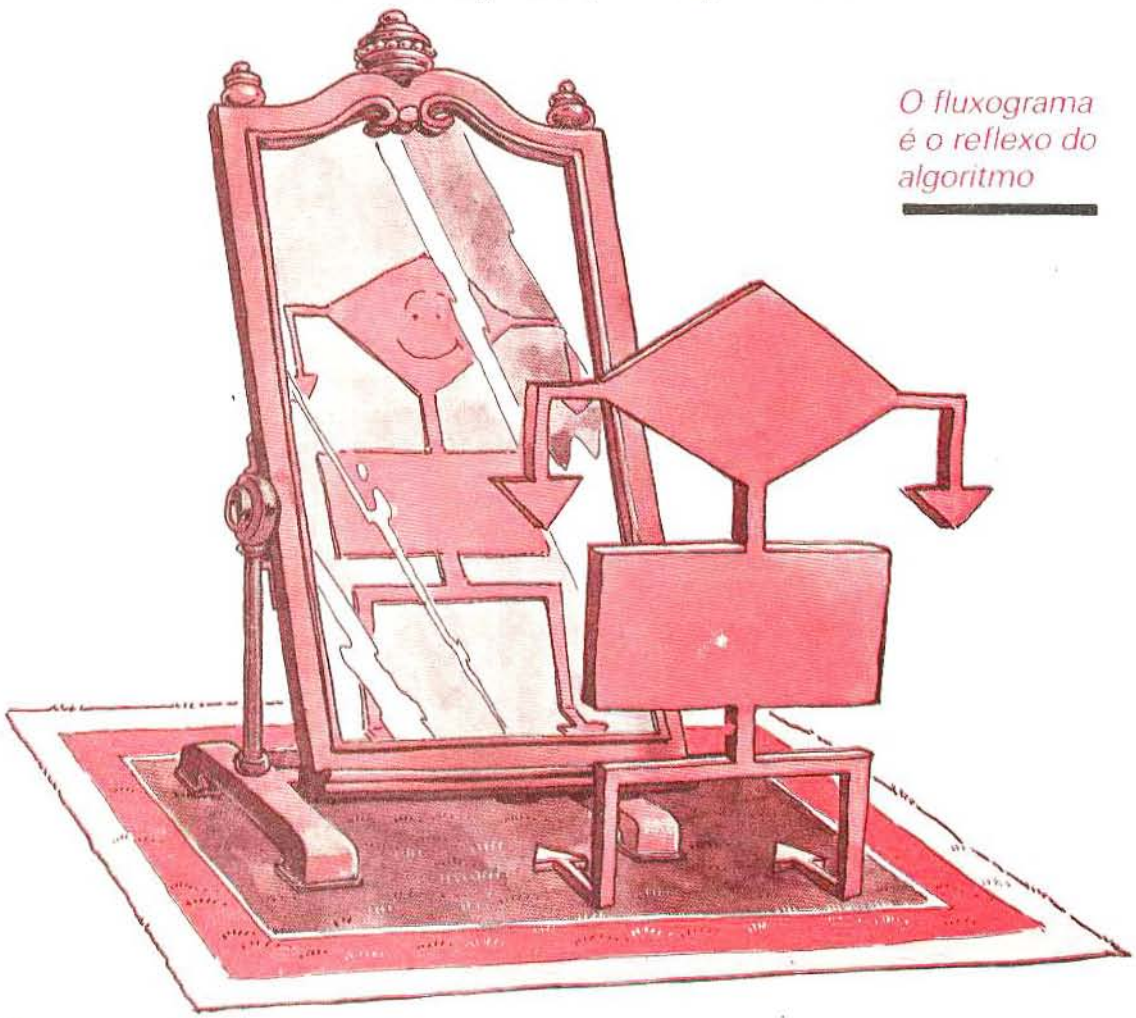
Figura 8.3 Losango (símbolo da decisão)

No caso de um algoritmo tão simples quanto o da preparação do ovo-de-três-minutos, o fluxograma não é muito útil, e pode ser dispensado. O verdadeiro valor do fluxograma aparece quando você começa a planejar algoritmos mais complexos que envolvem muitas escolhas e decisões.

Vamos agora projetar um novo programa que pergunte por sua data de nascimento, a data de hoje e então calcule a sua idade. O algoritmo é óbvio. O fluxograma é mostrado na figura 8.2. O losango no fluxograma indica um teste, isto é, uma escolha na seqüência. Neste ponto, para facilitar a conversão do fluxograma em um programa, certifique-se de que cada escolha tenha apenas *dois* resultados ou respostas: "sim" ou "não". Indique as setas adequadamente. Agora examine a figura 8.2 e verifique que cada seta saída do losango seja indicada com "sim" ou com "não", dependendo do resultado do teste (veja a figura 8.3)

Em geral, há três maneiras como as setas podem ser desenhadas, como mostra a figura 8.4. Você pode selecionar qualquer maneira que preferir. O propósito de sua seleção será simplesmente o de facilitar a leitura de seu fluxograma. A posição das setas, do "sim" e do "não", pode ser livremente trocada; em outras palavras, o "sim" pode ficar à direita, se você preferir.

Vamos voltar à figura 8.2, ao fluxograma que calcula a idade, e examinar o algoritmo que ele representa.



O fluxograma é o reflexo do algoritmo

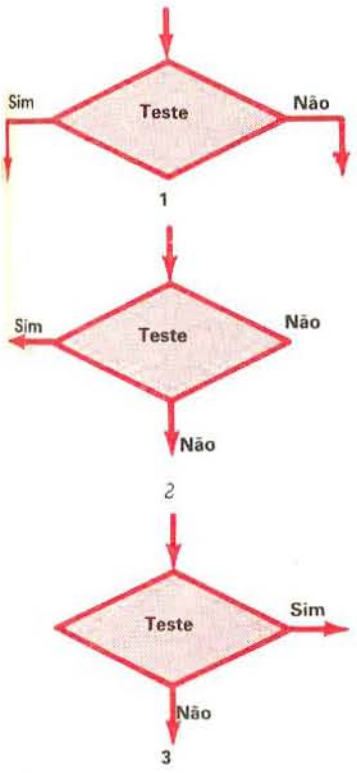


Figura 8.4 As três maneiras de saída às setas

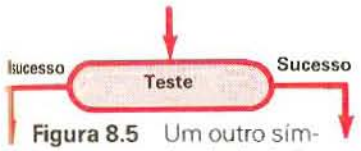


Figura 8.5 Um outro símbolo para a caixa de decisão

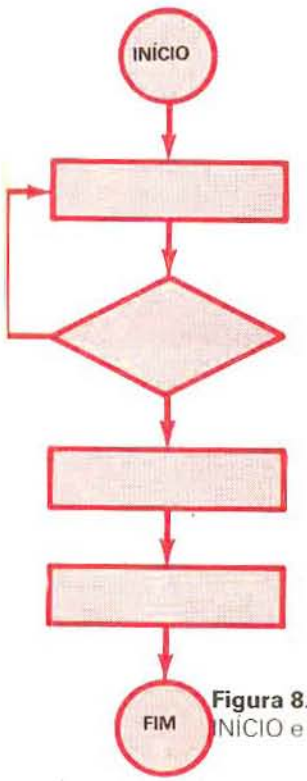


Figura 8.6 Símbolos para INÍCIO e FIM

Primeiro é requisitada a data de hoje. Este passo corresponde ao primeiro retângulo no fluxograma (indicado como 1). O segundo requisito é sua data de nascimento. Este é o retângulo indicado como 2 no fluxograma.

Em seguida devemos verificar se a data de nascimento é anterior à data de hoje. Se o valor da data de seu aniversário é maior do que o valor da data de hoje, um erro será reconhecido e o processo terá que ser recomeçado. Caso contrário, a data de aniversário será assumida como válida. Este passo corresponde ao losango (indicado como 3) no fluxograma. Para ser ainda mais aperfeiçoado, rejeitaríamos ainda idades que resultassem em 150 anos ou mais, desde que tal idade não possa ser válida. No entanto, aceitar esta idade não cria sérios efeitos adversos. Por conseqüência pode não valer a pena o trabalho de checá-la com tal objetivo.

No losango 4 do fluxograma, determinamos se o mês de sua data de nascimento é menor do que o mês corrente. Se for, seu aniversário neste ano já passou e sua idade pode ser calculada (no retângulo 5 do fluxograma) como a diferença entre ano corrente e o de seu nascimento. Por exemplo, se você nasceu em fevereiro de 1946 e nós estamos em março de 1984, sua idade é: 1984 − 1946 = 38.

Caso contrário (este é o retângulo 6 do fluxograma), sua idade é calculada como: o ano corrente menos o ano de seu nascimento, menos 1. Por exemplo, se você tiver nascido em junho de 1942 e estivermos em março de 1984, sua idade seria 1984 − 1942 − 1 = 41. Para manter simples o exemplo, nós não checamos o dia do mês. Adicionaremos este aperfeiçoamento mais tarde.

Os passos para o algoritmo devem estar agora bem mais claros. Vamos examinar os símbolos do fluxograma.

Os Símbolos do Fluxograma

No fluxograma os retângulos são usados para cálculos e ações diretas que não envolvem uma decisão como uma entrada ou saída. Os losangos são usados para testes ou decisões. Eles devem sempre ter ao menos duas setas saindo. Finalmente, cada algoritmo deve ter um começo e um fim. Isto está definido nas indicações: INÍCIO e FIM.

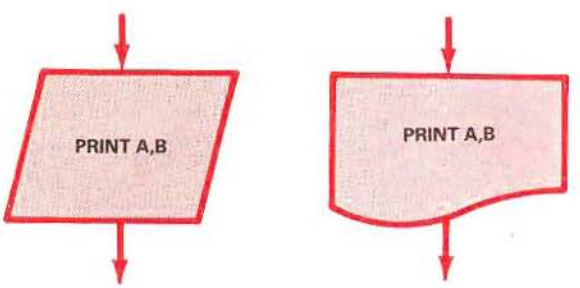


Figura 8.7 Símbolos para PRINT

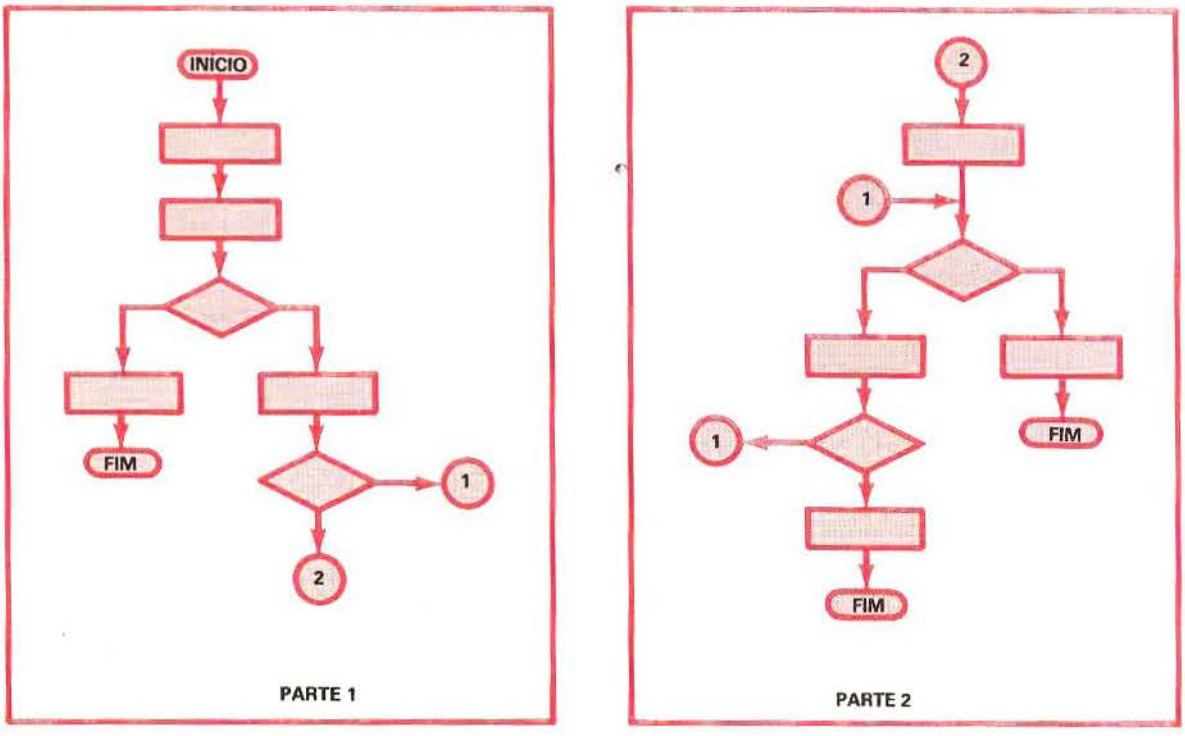


Figura 8.8 Cortando um fluxograma

Os símbolos usados no fluxograma não são uniformemente padronizados. Muitos foram propostos, mas nenhum ganhou aceitação universal. O retângulo é sempre usado. O losango, no entanto, pode ser substituído por outro sem arestas como é mostrado na Figura 8.5. Além disso, os símbolos INÍCIO e FIM podem ser colocados em um círculo pequeno como é mostrado na Figura 8.6. Finalmente, alguns símbolos especiais podem ser usados para indicar o uso de periféricos específicos. Por exemplo, uma operação PRINT pode aparecer de duas maneiras como é mostrado na Figura 8.7.

Na prática, você não precisa se assustar com esses símbolos. Um fluxograma é, simplesmente, uma maneira de visualizar, convenientemente, um algoritmo (especialmente quando ele contém várias decisões). Você pode ainda usar símbolos diferentes à sua escolha. No entanto, é melhor usar os símbolos comuns, para que seus programas possam ser mais facilmente comparados com outros, de forma que você possa ler e facilmente seguir qualquer outro fluxograma.

Dividindo o Fluxograma

Aqui está mais uma convenção muito usada. Se um fluxograma tiver de ser estendido por muitas páginas, você pode cortá-lo em pedaços. Classifique cada seta de conexão com um número ou nome, garantindo indicações de entrada para os fluxogramas das outras páginas. A figura 8.8 mostra um exemplo do uso de números para conectar setas.

Aperfeiçoando o Fluxograma

As instruções colocadas nos retângulos do fluxograma podem ser escritas como lhe aprouver. Elas não são instruções BASIC. Quando es-

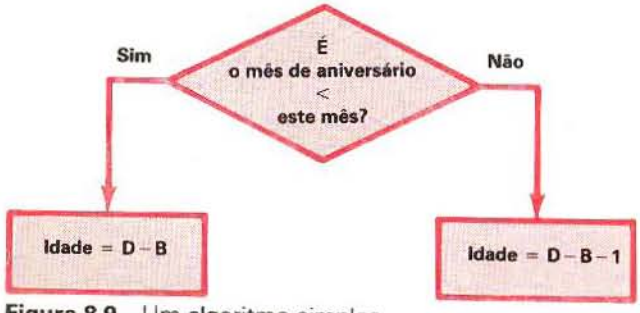


Figura 8.9 Um algoritmo simples

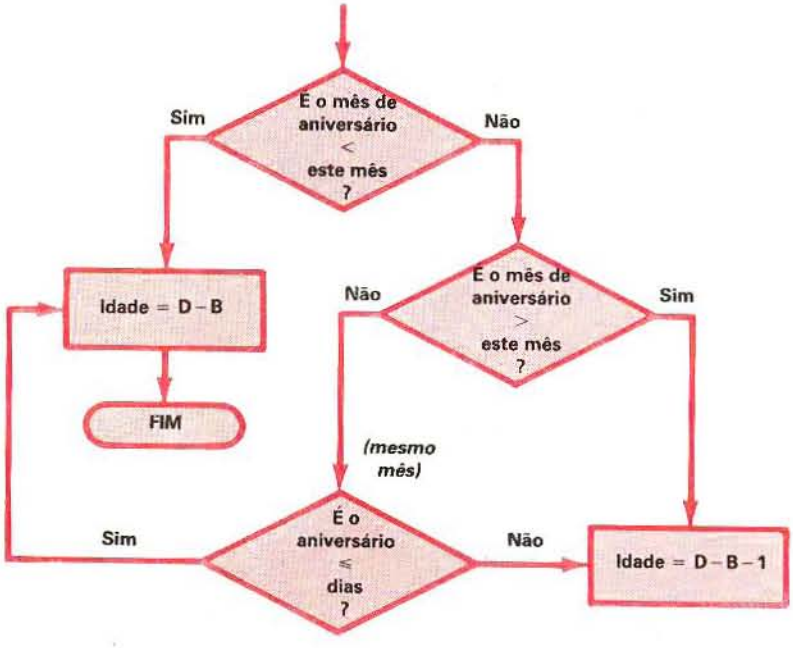


Figura 8.10 Um fluxograma aperfeiçoado

crever seu fluxograma pela primeira vez, você pode escrever instruções vagas como "diga-me sua data de nascimento". Mais tarde, você pode aperfeiçoar as instruções contidas nos retângulos e desenhar um fluxograma mais detalhado, que seja mais facilmente traduzido para um programa.

Se você sentir que as instruções nos retângulos são suficientemente precisas para escrever um programa equivalente, não será necessário fazer nenhuma mudança adicional. Se sentir, no entanto, que elas são vagas ou complexas para serem transcritas diretamente para um programa, então você pode substituí-la por uma seqüência de instruções mais detalhadas.

Como exemplo, recorde o fluxograma da figura 8.2, que checa o mês de seu aniversário, mas não o dia, ao verificar se sua data de nascimento já aconteceu no mês em curso. Vamos aperfeiçoar, para que seja verificado tanto o mês quanto o dia. O segmento que corresponde ao fluxograma inicial aparece na figura 8.9. O novo, ou o fluxograma aperfeiçoado, é mostrado na figura 8.10.

Na prática, a maioria das pessoas não escreve o algoritmo, elas vão diretamente à fase do fluxograma. Isto é ótimo. No entanto, programadores com alguma experiência (ou com quase nenhuma) também omitem a fase do fluxograma, e começam escrevendo o *programa* em uma folha de papel. Isto é altamente perigoso e um caminho para o erro. Recomendo que você sempre desenhe um fluxograma antes de escrever qualquer programa. Mais tarde, quando se tornar um programador experiente, poderá estar apto a dispensar um fluxograma detalhado e desenhar apenas um esboço de fluxograma.

Teste Manual

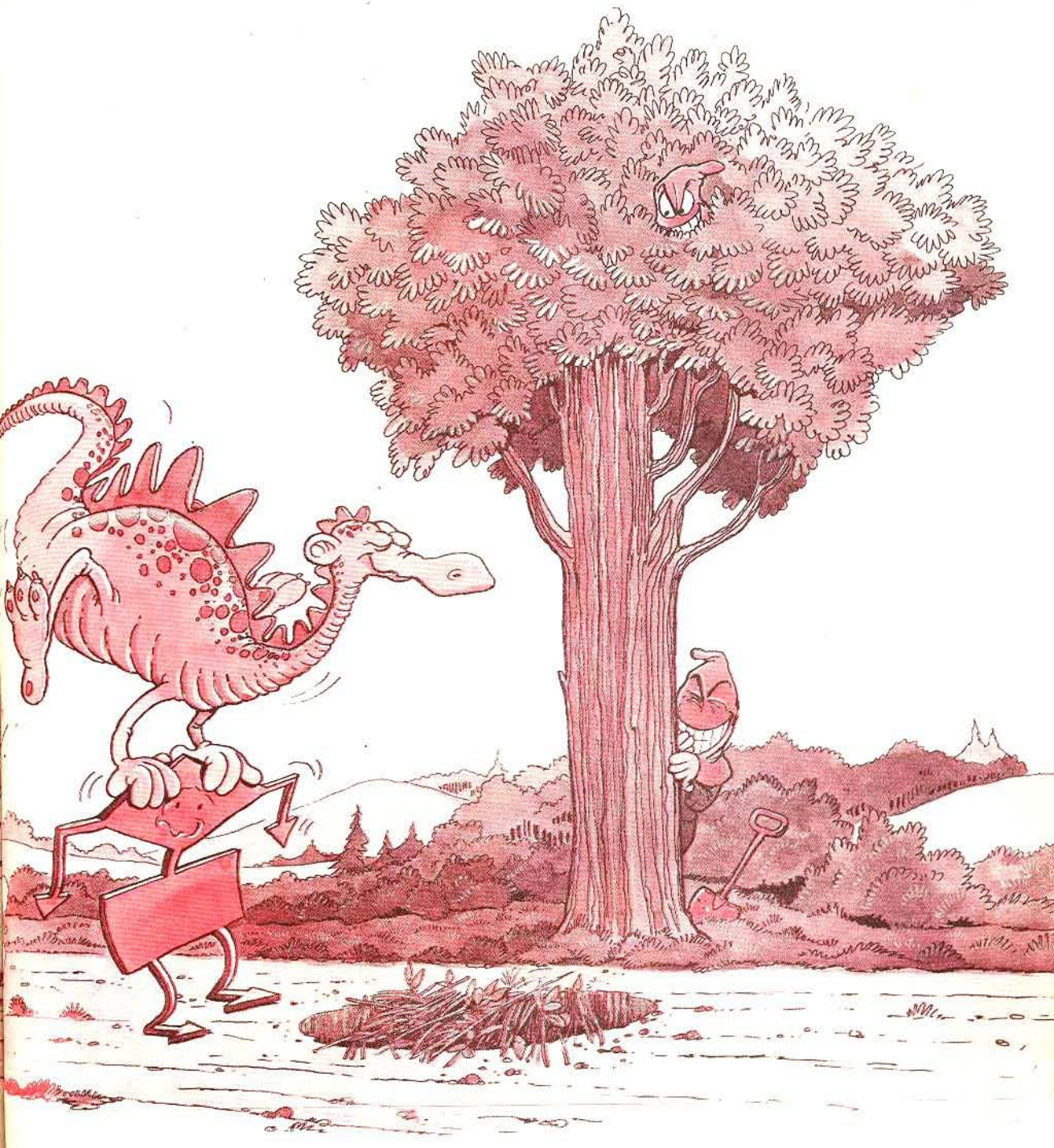
Se você já escreveu um fluxograma, teste-o com exemplos reais. Tenha certeza de que o resultado é correto ou, pelo menos, aparenta ser correto. É chamado *teste manual*, em oposição ao uso do computador.

Por exemplo, volte ao fluxograma que calcula a idade na figura 8.2 e forneça-lhe sua própria data de nascimento e a data de hoje, como indicado. Ele aponta sua idade correta? Se afirmativo, as coisas vão

Se você pulou o fluxograma, provavelmente vai ter problemas

bem. Se não, há algum erro. Agora tente novamente, com valores diferentes, usando datas de aniversários que ocorram antes e depois da data de hoje. Funcionou? Se afirmativo, o algoritmo provavelmente está certo. Se não, há um erro. O teste manual é a maneira mais rápida de se assegurar de que não há nenhum erro primário.

Tendo escrito um fluxograma que parece funcionar, teremos alcançado todos os passos previamente necessários para se escrever um programa real. Vamos agora escrever o programa.



A Codificação

Escrever instruções de programas é chamado de *codificação*. A *programação* se refere normalmente à seqüência total necessária para criar um programa, projetar o algoritmo, desenhar o fluxograma, codificar, corrigir erros e testar. Codificar envolve a tradução do conteúdo do fluxograma para instruções de programa, expressas numa linguagem de programação — neste caso, expressa em BASIC.

Isto foi o que aprendemos até agora: aprendemos a traduzir instruções, fórmulas, testes e condições em instruções BASIC.

A chave para codificar bem é escrever um fluxograma detalhado o bastante para poder facilmente codificar cada etapa do fluxograma em algumas poucas instruções BASIC. Geralmente, nos primeiros passos da programação, cada elemento do fluxograma é traduzido em algumas instruções BASIC, uma ou duas, isto é, há uma equivalência direta entre os elementos do fluxograma e as instruções. Mais tarde, quando sua habilidade e experiência para programar tiver crescido, você estará apto a escrever fluxogramas mais "descontraídos", onde cada elemento é codificado em várias instruções BASIC.

Apesar da experiência, a fase de codificação é, na maioria das vezes, aquela que requer o menor tempo na seqüência de desenvolvimento de um programa. Testar um programa requer usualmente muito mais tempo do que a codificação. Daí a importância de desenhar um bom fluxograma para poder minimizar erros e tempo de teste.

Quando codificar um programa, seja preciso, claro e legível, para que ele possa trabalhar rapidamente e possa ser facilmente testado ou modificado.

"Codificar é fácil, desde que você tenha um bom fluxograma!"

Seja Preciso

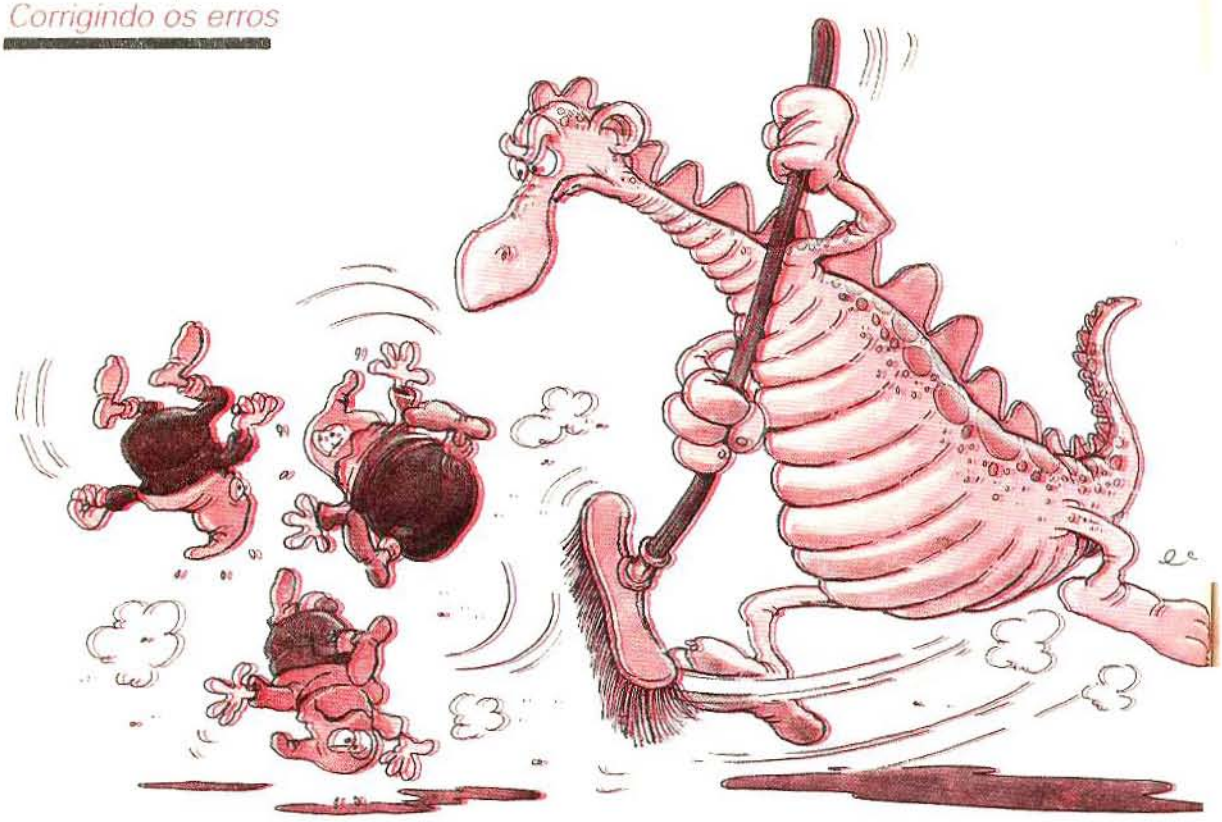
Escreva suas instruções de programa com o máximo cuidado; qualquer erro de colocação de sinais de pontuação ou símbolos, provavelmente ocasionará erros no programa.

Seja Claro

Use nomes de variáveis que sejam fáceis de lembrar. Deixe intervalos ou espaços entre os números das instruções para o caso de querer inserir outras instruções entre elas. Use anotações (instrução REM) livremente, durante o programa, para dar maior clareza ao que ele faz.

Então, até aqui projetamos um algoritmo, desenhamos um fluxograma e escrevemos o programa correspondente. E você espera seriamente que ele funcione. Não. Desculpe. Na maioria dos casos, *programas não funcionam na primeira vez*. Usualmente exigem várias tentativas e muita experiência, independentemente de seu tamanho, para funcionar corretamente. Este é o tópico da próxima fase.

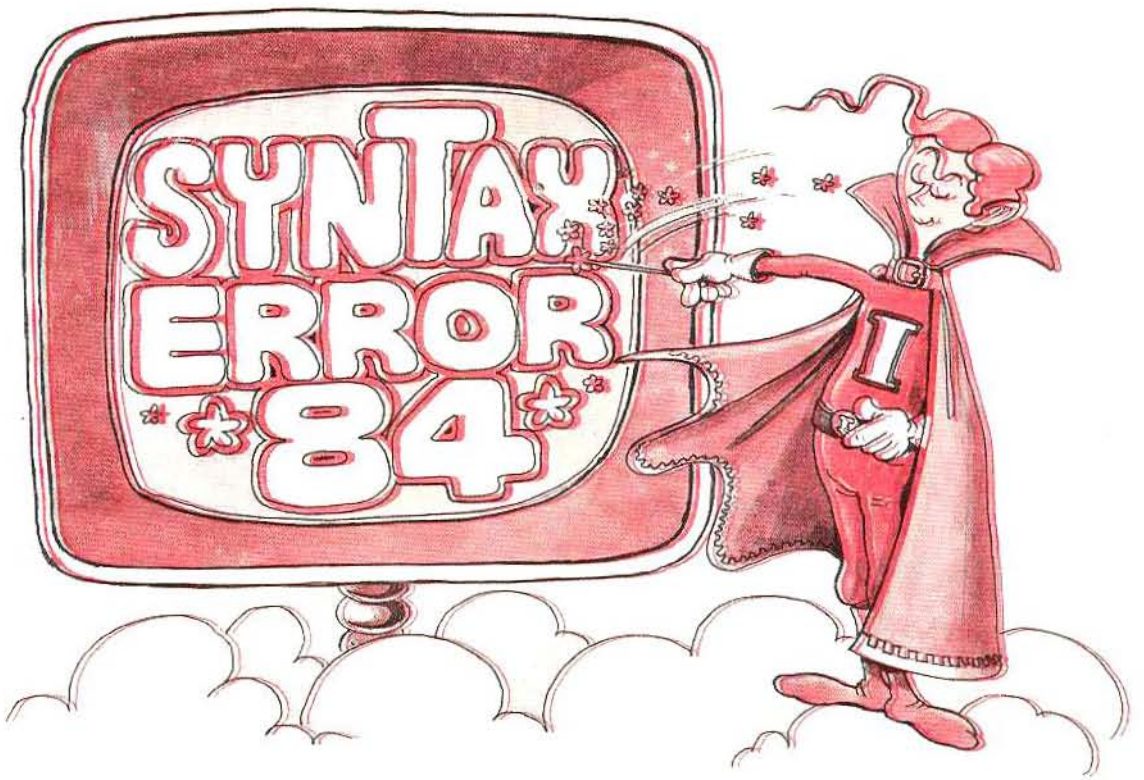
Fora com os erros! Seja preciso; seja claro.



A Correção dos Erros

Você codificou seu fluxograma num programa BASIC. Seu programa está ainda no papel. Você agora o colocaria na memória de seu computador, digitando "RUN", fazendo tudo para que ele funcione. Isto é chamado de *teste* e *correção de erros*. Em inglês, *debugging*. Os erros são conhecidos em inglês como *bugs*. Em português, usa-se o termo *debugar* um programa, ou seja, corrigir-lhe os erros. Cada vez que você encontrar um erro, deve corrigi-lo e então "RUN" novamente o programa. Mesmo se tiver sido cuidadoso na codificação do programa, um longo programa raramente funciona corretamente.

Felizmente, seu intérprete BASIC ajudará você a diagnosticar alguns problemas. Se seu programa contém um tipo de erro que pode ser detectado pelo intérprete, a execução do programa será interrompida logo depois que você digitar RUN e o intérprete dará a você um diagnóstico como "ERRO DE SINTAXE na linha 84". O intérprete, antes de tudo, ajudará a detectar *erros de sintaxe* (o uso ilegal de símbolos ou operações). Infelizmente, ele não ajudará a detectar os erros mais perigosos: os *erros de lógica* e *de projeto*. Esta responsabilidade é sua. Esta é a razão para você investir tempo em desenhar cuidadosamente seus fluxogramas. Também é por isso que cada vez



O intérprete ajuda na diagnose dos erros de sintaxe

que um número é fornecido a um programa, ou gerado por ele, deve ser validado o seu alcance. Na eventualidade de seu programa conter um erro lógico, esta técnica o ajudará a isolar a seção do programa que contém o erro.

Usualmente, no caso de um programa simples, alguns erros tipográficos serão detectados pelo intérprete BASIC. Você os corrigirá e seu programa funcionará. Deverá ser assegurado que ele funcione corretamente, testando-o para vários casos ou valores; seu programa pode conter erros lógicos. Na maioria dos casos, no entanto, você fará com que seu programa funcione corretamente.

Sugestões Práticas

Aqui está uma sugestão prática: você deveria inserir instruções PRINT adicionais durante o programa, para verificar os valores-chave ao longo da execução. Isto o ajudará a detectar valores estranhos e a isolar as instruções que os causaram. Aqui está um exemplo de PRINT que você pode inserir:

```
1235  PRINT "TESTE PARA VALOR MEDIO E"; MEDIA
```

Assim, se o programa funcionar você pode remover essas informações PRINT extras. Esta técnica é chamada de *rastreio de uma variável.*

Como outra sugestão prática, a qualquer momento que seu programa seja interrompido automaticamente ou por intervenção do intérprete, você deve usar o *modo de execução imediata* para verificar o valor das diferentes variáveis do seu programa. Por exemplo, você pode digitar:

```
> PRINT MEDIA
```

e, então:

```
> PRINT SOMA
```

para checar o valor corrente dessas duas variáveis.

A chave de uma correção de erros bem-sucedida é experiência e muita previdência. Da próxima vez, gaste muito mais tempo projetando o programa e o fluxograma e você gastará menos tempo corrigindo erros.

Então, seu programa funcionou corretamente. Você acha que ele está correto e não quer mais mexer nele. No entanto, pode-se encontrar um erro mais tarde, ou você pode usá-lo novamente ou reparti-lo com outra pessoa. Para fazer um programa reutilizável será necessário mais um passo: você deve documentar o programa para que sempre se lembre daquilo que ele faz.

Documentação

Você terminou de projetar e codificar um programa. Você já está familiarizado com sua operação e com o que cada instrução faz. Você ficará surpreendido em descobrir o quão rapidamente você esquecerá sua função e como é difícil lê-lo ou entendê-lo mais tarde. Se você pretende utilizá-lo de novo ou corrigir erros encontrados mais tarde, é vital que o clarifique pronta e completamente. Isto não só significa limpá-lo, como documentar tudo que possa requerer uma explanação em manuais ou em instruções REM dentro do programa.

Aqui está um sumário das técnicas descritas para clarificar o programa.

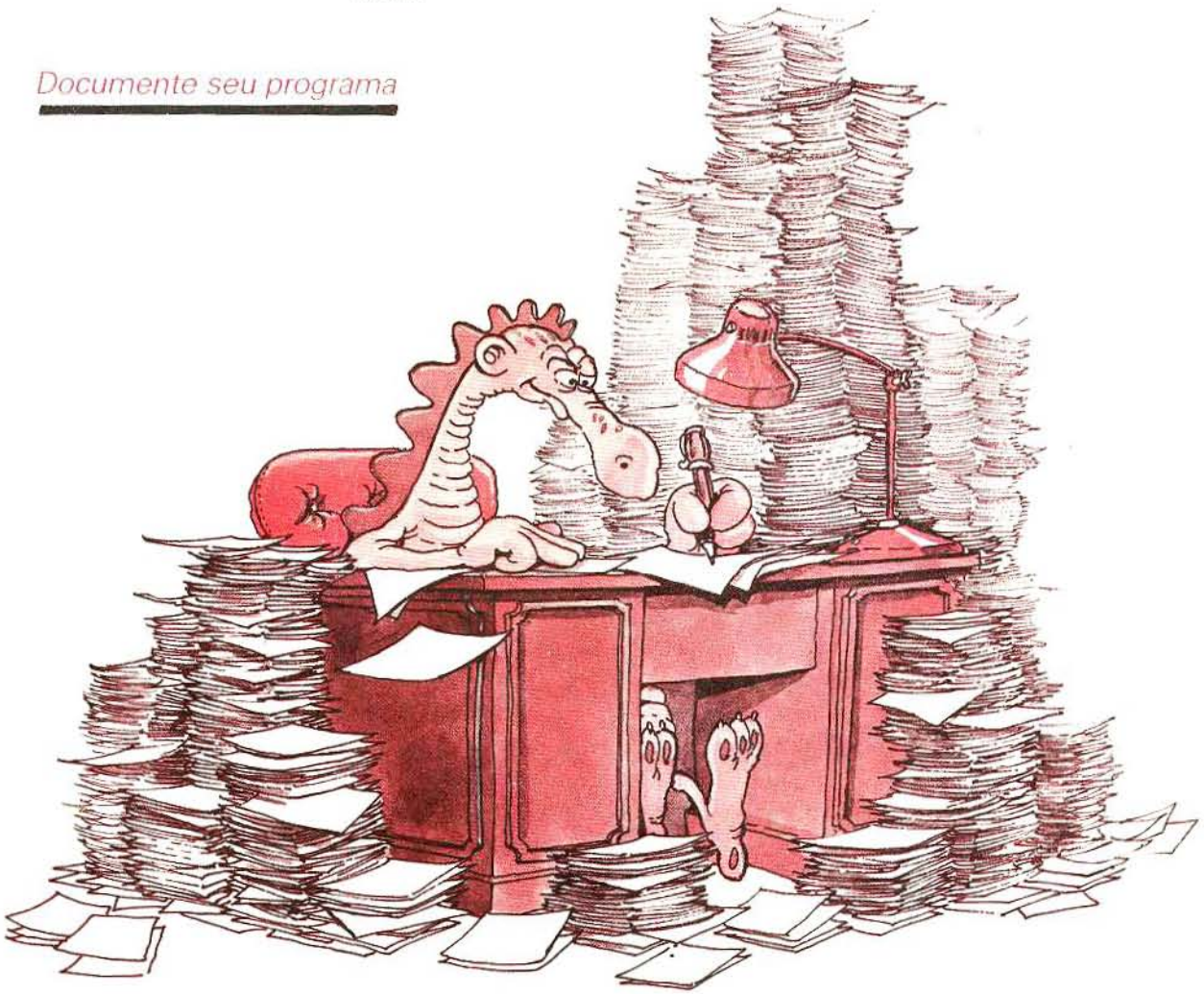
Use um layout **claro:** Seções separadas por linhas em branco ou instruções vazias. Use, para maior clareza, alinhamentos ou denteações. Você pode querer usar números de linha que tenham todos o mesmo comprimento. Desta maneira, todas as instruções do programa serão alinhadas verticalmente. Adicionalmente, cada vez que você usa a instrução FOR ... NEXT será uma boa idéia dentear o bloco de instruções que estão incluídas entre o FOR e o NEXT.

Use também espaços livremente, para clarificar instruções complexas, em particular aquelas que contêm expressões matemáticas. Use parênteses para clarificar o resultado de um cálculo.

Explique o que você faz: Use a instrução REM para explicar fórmulas, testes, nomes ou convenções. É também uma boa idéia providenciar uma pequena explanação escrita para quaisquer métodos ou técnicas que você esteja usando, que não sejam óbvios ou que não sejam descritos pela instrução PRINT no programa.

Limpe o fluxograma: Produzir um fluxograma limpo ou uma série de fluxogramas que correspondam exatamente ao seu programa. Às vezes, durante o processo de correção de erros, você pode querer mudar no último minuto e fazê-lo diretamente no programa. Assegure-se de que isto refletiu-se no fluxograma original ou você encontrará muitas dificuldades para mudar ou corrigir seu programa mais tarde.

Documente seu programa



Renumere suas linhas: Às vezes, no processo de correção do programa, isto é, na fase de remoção de erros, você pode precisar inserir instruções adicionais. Se você acha que seu programa está correto, é uma boa idéia renumerar as linhas para que todos os números de linhas estejam regularmente espaçados. Tal procedimento facilitará, mais tarde, mudanças no programa. Programas especiais que renumeram um programa inteiro estão disponíveis no mercado. Se tal disponibilidade não existe, pode ser uma boa idéia perder algum tempo renumerando instruções para obter uma seqüência clara. Isto é uma conveniência, não uma obrigação.

Sumário

Neste capítulo, descrevemos os cinco passos de um programa acabado: projeto, fluxograma, codificação, correção de erros e documentação. Vamos revê-los.

Cada programa requer o projeto de um algoritmo. O algoritmo deve ser projetado no mínimo mentalmente, se não no papel. O algoritmo deve ser esboçado como uma série de fórmulas ou notas descrevendo os passos essenciais.

O passo seguinte é desenhar o fluxograma que descreva a seqüência completa de eventos. Considerá-lo como um passo obrigatório para qualquer programa que envolva mais do que algumas linhas. Lembre-se: quanto mais cuidadosamente você projetar seu fluxograma, maior será sua chance de ter um programa correto.

O passo seguinte é codificar. O fluxograma é traduzido em instruções BASIC. Praticar, acelerará consideravelmente esta fase. De fato, a fase de codificação rapidamente se transforma na fase mais curta.

Em seguida, vem o teste e a correção de erros. Esta fase é sempre necessária, muitas vezes é a fase mais longa. Cada programa deve ser cuidadosamente checado.

Finalmente, a qualidade da documentação facilitará ou impedirá o uso futuro ou as mudanças do programa.

Exercícios

8-1: Descreva as cinco fases de desenvolvimento do programa.

8-2: Qual é a diferença entre codificar e programar?

8-3: Qual é o propósito da correção de erros?

8-4: Como você rastreia uma variável?

8-5: Por que renumerar um programa após fazer muitas mudanças?

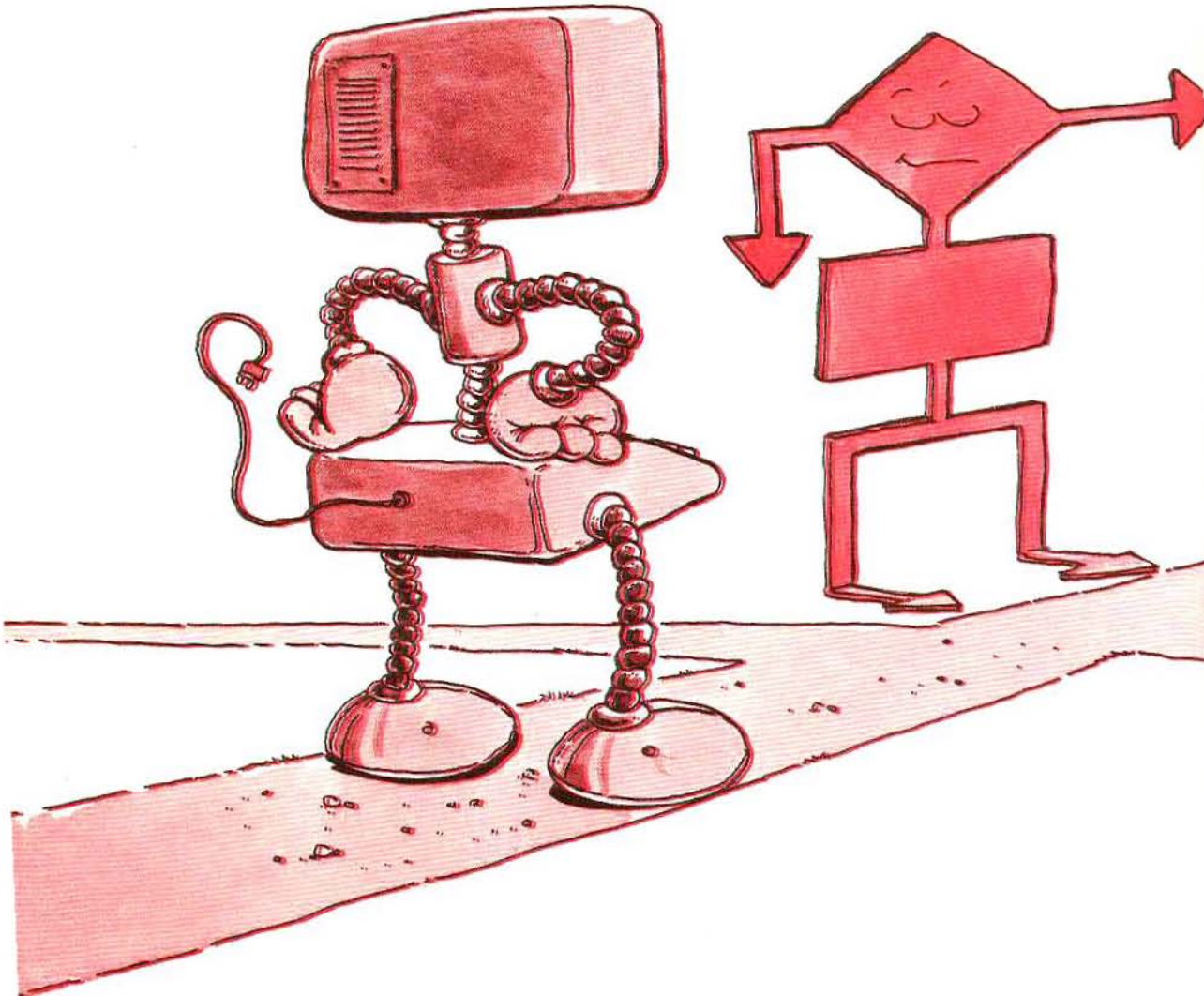
8-6: O que é um fluxograma?

8-7: Escreva um fluxograma para ligar seu carro ou operar um eletrodoméstico.

8-8: Quais são as vantagens de um programa escrito com clareza?

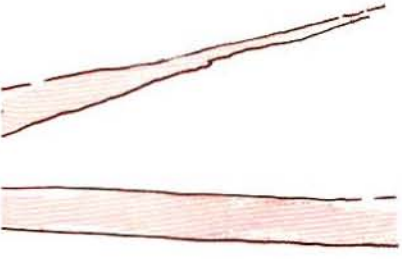
8-9: Descreva as técnicas que podem ser usadas para clarear um programa.

Estud
Conversã



de Caso:
o Métrica

9



Agora desenvolveremos um programa completo e o descreveremos a cada passo. Aqui está um problema para ser resolvido. *Precisamos escrever um programa que possa converter automaticamente um peso expresso em onças em seu valor equivalente em gramas. Este programa tanto pode converter um número digitado no teclado, como imprimir uma tabela de conversão de pesos para os números existentes entre dois valores especificados.*

O Projeto do Algoritmo

A seqüência simples de passos que seguiremos para solucionar este problema é bastante direta. Perguntaremos ao usuário o que ele (ou ela) quer (uma simples conversão ou uma tabela de valores) e, então, procederemos à ação solicitada. Este é o nosso algoritmo simples. Vamos aperfeiçoá-lo.

Uma onça é igual a 28.35 gramas. A conversão de onças em gramas é geralmente realizada pela seguinte fórmula:

$$P_{gramas} = P_{onças} \times 28.35$$

ou, em resumo:

$$P_{g} = P_{on} \times 28.35$$

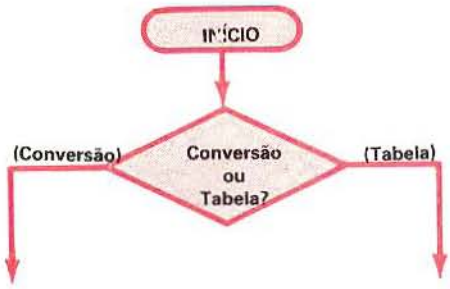
Aqui está o algoritmo básico:

- ▶ Especificar se é conversão simples ou tabela
- ▶ Se for conversão, requisitar peso em onças
- ▶ Converter em gramas (usando a fórmula acima) e mostrar o resultado
- ▶ FIM
- ▶ Se tabela, pedir o peso máximo em onças
- ▶ Converter em gramas e mostrar os resultados até o limite de tabela
- ▶ FIM

Na prática, não há necessidade de escrever o algoritmo, desde que você prepare um fluxograma.

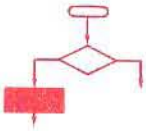
O Fluxograma

Na preparação do fluxograma, precisamos, em primeiro lugar, saber o que o usuário quer que o programa execute: uma conversão simples ou uma tabela de valores. Aqui está o elemento correspondente ao fluxograma:

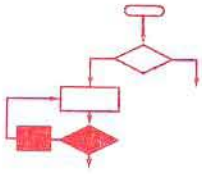


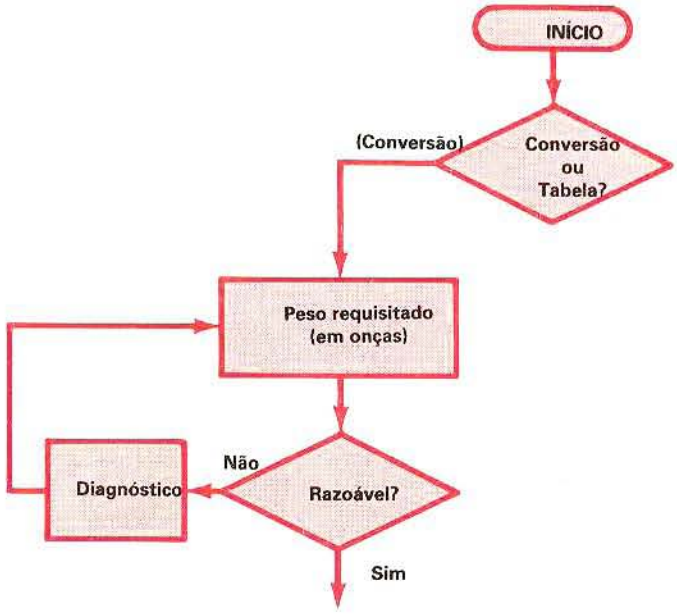
Este é um elemento de decisão com duas saídas possíveis (isto é, ramos): CONVERSÃO ou TABELA.

Vamos primeiro examinar a CONVERSÃO. O usuário quer converter um peso simples de onças em gramas. Devemos requisitar o valor do peso. Aqui está, no fluxograma, a entrada correspondente:



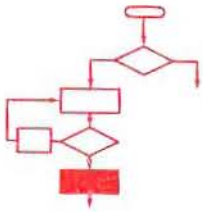
Podemos, agora, converter este valor em gramas. No entanto, vamos aperfeiçoar o fluxograma, imediatamente. Como precaução, validaremos o valor suprido pelo usuário, checando o quanto ele é razoável ou não. Aqui está a maneira como nosso fluxograma aparece com essa validade adicional.

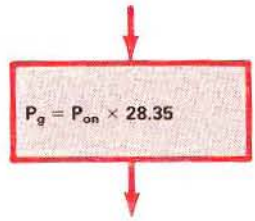




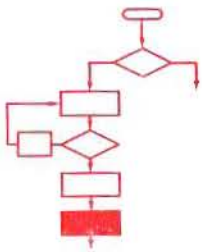
Note que adicionamos um elemento para checar se a entrada é razoável. Se for razoável, prosseguiremos. Se não, um diagnóstico como "entrada não razoável, tente novamente" é acionado, e o programa requisita um novo valor para o peso.

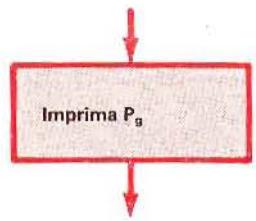
Agora nós temos um valor válido para o peso. Vamos convertê-lo em gramas. Isto é obtido como se segue:



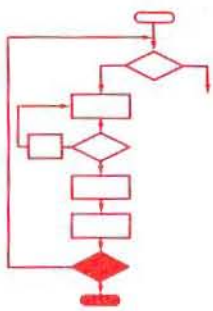


Podemos agora mostrar o resultado. Isto é executado pelo elemento seguinte:

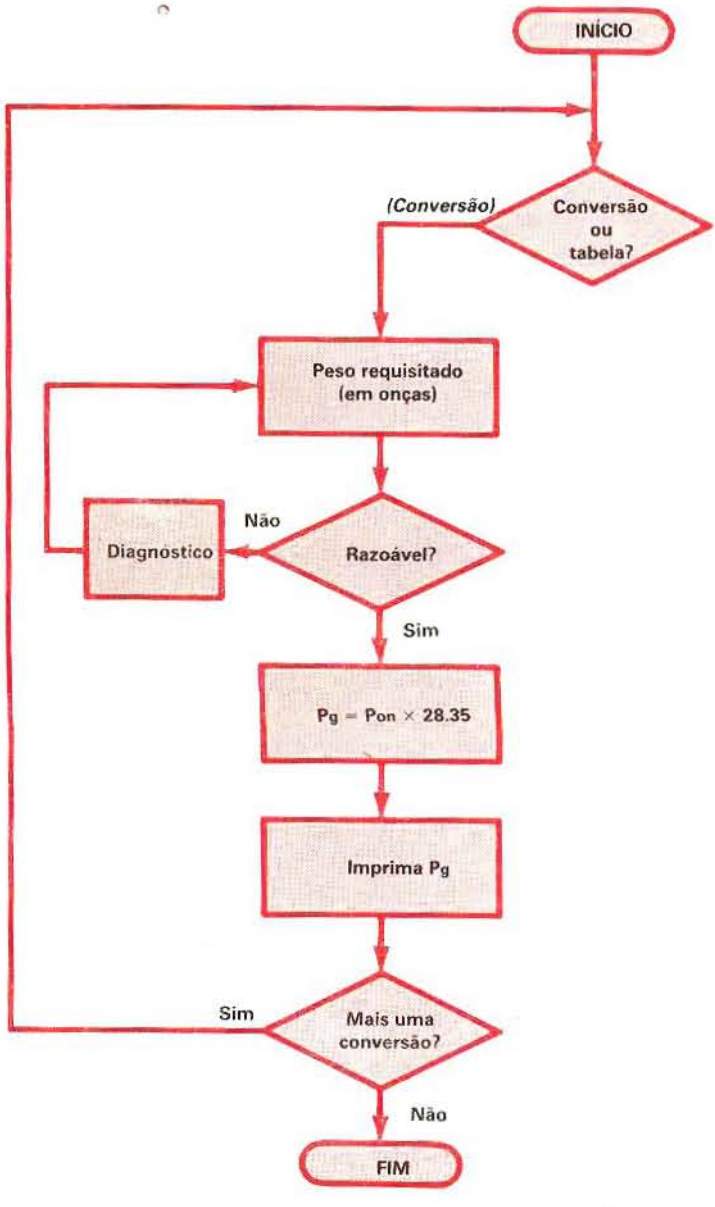




A conversão simples agora está feita. Terminaríamos aqui esta parte do fluxograma. No entanto, vamos acrescentar uma característica conveniente e perguntar ao usuário se ele quer fazer uma outra conversão. Isto é obtido pelo elemento seguinte:

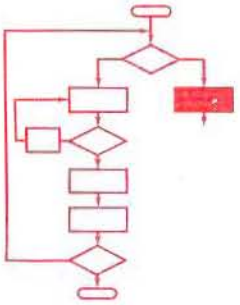


O símbolo de decisão com a seta à esquerda indica que esta seta será conectada ao começo do fluxograma. Até esse ponto, nosso fluxograma está assim:

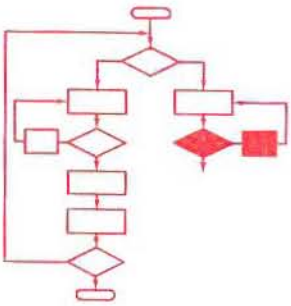


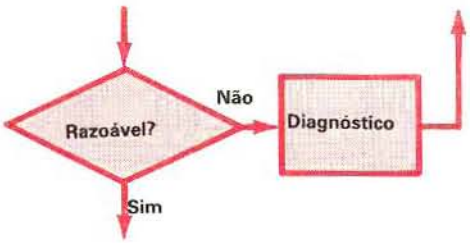
Vamos agora voltar ao nosso primeiro elemento de decisão e examinar o que acontece quando o usuário quer executar uma tabela ou tábua de valores. Esta é a opção "tabela" no alto do fluxograma.

Devemos conhecer o valor máximo a ser convertido. Isto é obtido pelo elemento seguinte:



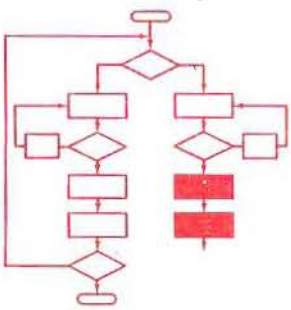
Novamente, por segurança, checaremos se este número é razoável:



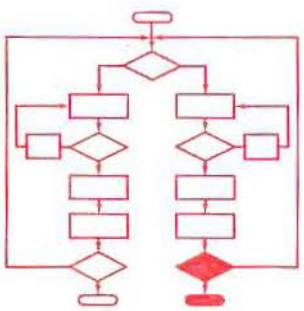


Como antes, o programa não prosseguirá, até que um valor razoável para MAX seja fornecido pelo usuário.

Quando um valor razoável for fornecido, podemos prosseguir e mostrar uma tabela que converta onças em gramas até o limite desejado.



Finalmente, vamos acrescentar a mesma característica de conveniência que usamos no caso de conversão única, ou seja, perguntar ao usuário se ele gostaria de executar mais uma conversão.



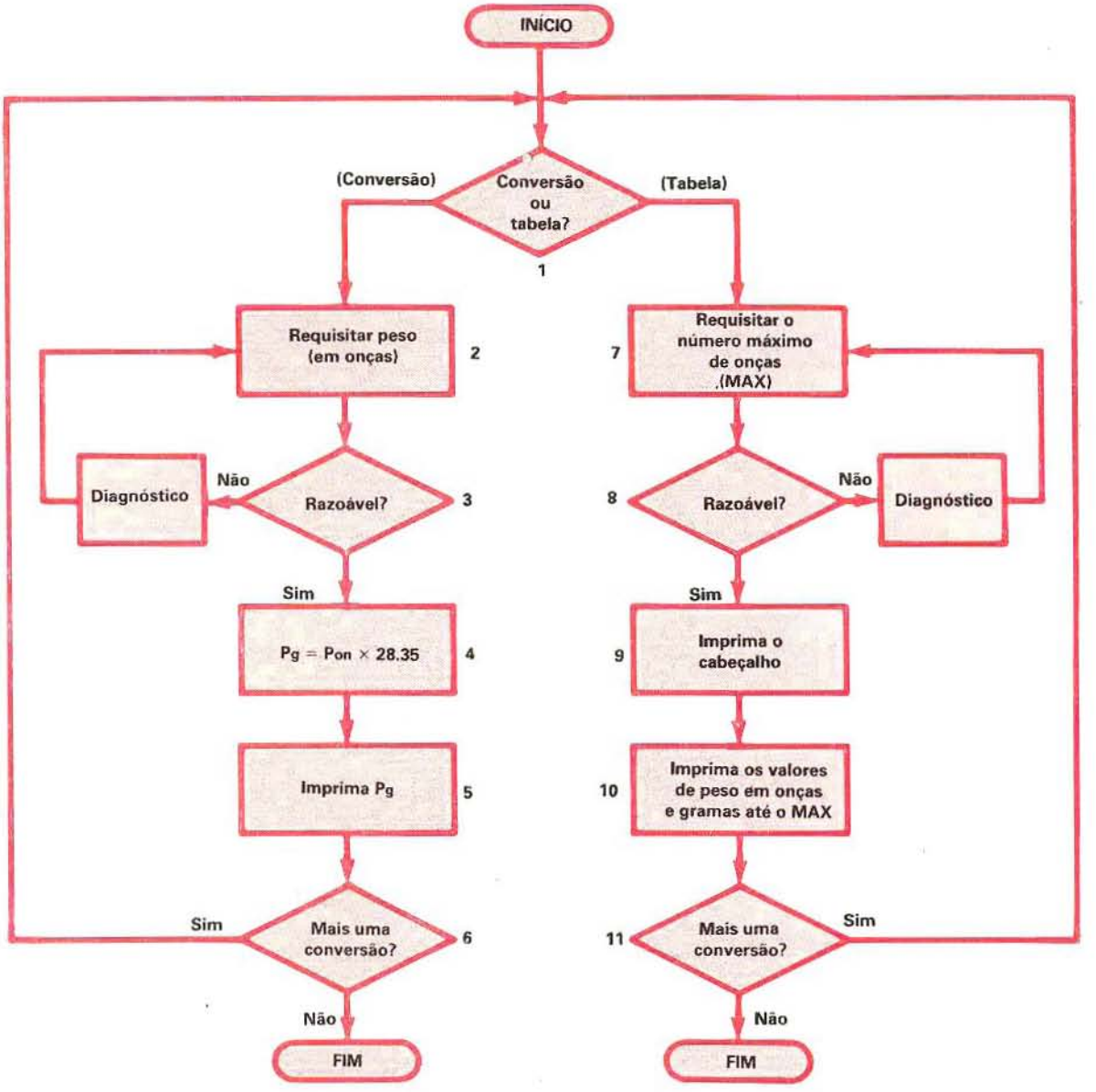


Figura 9.1 Fluxograma de conversão de peso

A figura 9.1 mostra o fluxograma completo. Este fluxograma é típico. Os conteúdos dos elementos são, de alguma forma, escritos "livremente". Alguns dos elementos no fluxograma foram codificados em apenas uma ou duas instruções BASIC, enquanto outras reque-

rem muito mais. No entanto, é a seqüência que conta. O fato de que alguns elementos podem ser mais detalhados do que outros não tem importância. Lembre-se de que a seqüência do fluxograma deve ser exata, mas os detalhes podem ser escritos da maneira que for mais conveniente para você. Não há necessidade de perder uma porção de tempo dissecando o conteúdo dos elementos, desde que você sinta que pode codificá-lo de uma maneira direta e objetiva.

O fluxograma deve simplesmente ser claro e fácil de ler. Um fluxograma claro e bem organizado aumenta as suas chances de escrever um programa correto.

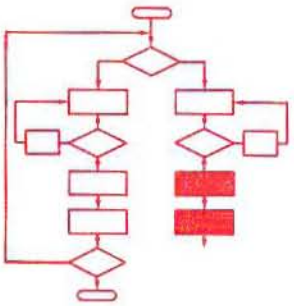
Para completar, aqui estão alguns refinamentos ou alternativas, que você pode considerar:

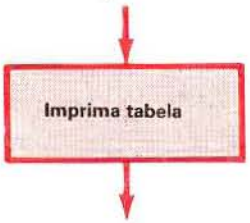
- Você pode explanar em detalhes a maneira como você checará o peso para a "razoabilidade". (Aqui simplesmente checaremos se Pon é um número positivo.)
- Você pode ser mais explícito sobre o diagnóstico e sobre os diálogos.

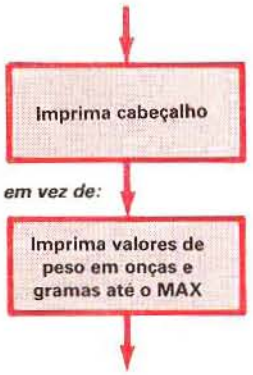
Em geral, a advertência é: *simplifique*. Faça exatamente o suficiente para que:

1. A seqüência de passos seja correta e completa.
2. Você entenda cada elemento e saiba facilmente como convertê-lo em instruções de programas.

Quanto mais simples o conteúdo dos elementos mais claro será o fluxograma. Aplicando essas advertências, podemos simplificar o fluxograma, escrevendo:







Ambas as opções são corretas. Use aquela em que você se sinta mais confortável. Aqui decidi sugerir os passos dos programas mais desenvolvidos e conseqüentemente fiz o conteúdo dos elementos do fluxograma mais explícito.

Você pode sempre reescrever o conteúdo de um elemento ou qualquer parte do fluxograma que queira, em um pedaço de papel separado, para facilitar a codificação ou para tentar uma outra alternativa.

Agora que temos um fluxograma, vamos testá-lo manualmente com números reais, para ter certeza de que ele funciona. Este processo de checagem à mão é óbvio, portanto, vamos adiante.

A Codificação

Agora escreveremos um programa que corresponde ao nosso fluxograma. Para facilitar a leitura do programa, assumiremos que o seu intérprete BASIC permite nomes de variáveis longos (nomes que usam mais do que uma letra) e operações com ponto flutuante (isto é, permite o uso de números decimais).

Se não for este o caso, você deve simplesmente encurtar os nomes das variáveis. Se o seu BASIC é um BASIC de números inteiros (não números fracionados), o programa funcionará, mas a conversão será apenas aproximada.

Vamos agora codificar cada elemento do fluxograma em instruções BASIC correspondentes.

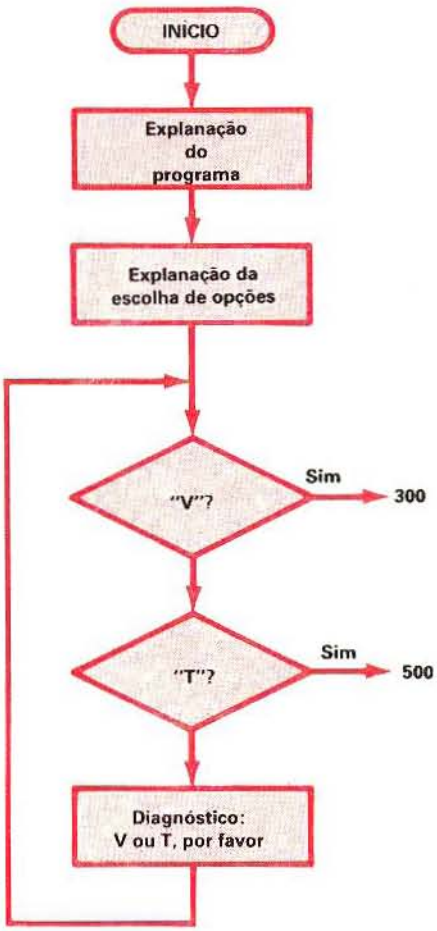
Aqui está o elemento 1 do fluxograma.

Aqui estão as instruções correspondentes do programa:

```
100  REM *** CONVERSAO DE ONCAS EM GRAMAS ***
110  REM ESTE PROGRAMA EXECUTA OU UMA CONVERSAO
         DIRETA
120  REM OU IMPRIME UMA TABELA DE VALORES
130  REM PRIMEIRO ESPECIFIQUE O MODO DE CONVERSAO: DIRE-
         TA OU TABELA.
140  PRINT "QUERO CONVERTER ONÇAS EM GRAMAS"
150  PRINT "SE VOCE QUER CONVERTER UM VALOR DIRETAMENTE
         DIGITE V."
160  PRINT "SE VOCE QUER IMPRIMIR UMA TABELA DE VALORES
         DIGITE T."
170  PRINT "SUA ESCOLHA (V OU T)";
180  INPUT ESCOLHA$
190  IF ESCOLHA$ = "V" THEN GOTO 300
200  REM ROTULO 300 E A CONVERSAO DE VALOR
210  IF ESCOLHA$ = "T" THEN GOTO 500
```

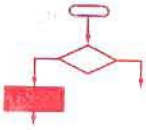
```
220  REM ROTULO 500 E A TABELA DE CONVERSAO
230  REM SE O CARACTERE NAO FOR V OU T A ENTRADA NAO E
         VALIDA.
240  PRINT "V OU T POR FAVOR:"
250  GOTO 170
```

Como você já está experiente, já é capaz de ir diretamente do elemento 1 do fluxograma para a instrução do programa correspondente. No entanto, no começo, você precisará escrever embaixo uma versão detalhada do primeiro fluxograma. Aqui está a versão detalhada do elemento 1.



Olhando para essa versão detalhada, repare como proximamente, o fluxograma corresponde às instruções BASIC. Em suma, note que introduzimos um *teste de validade* para "ESCOLHA$". Nós não vamos simplesmente assumir que o usuário coopere e digite "V" ou "T". Cheque-o. **Lembre-se** de que cada vez que você requisitar uma entrada (INPUT) você deve validá-la.

O resto é mais simples. Vamos converter o elemento 2:



As instruções correspondentes são:

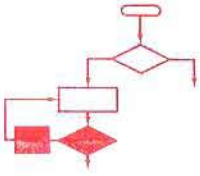
```
300  REM *** CONVERSAO DE VALOR ***
310  PRINT "DIGITE O PESO EM ONCAS...";
320  INPUT PON
```

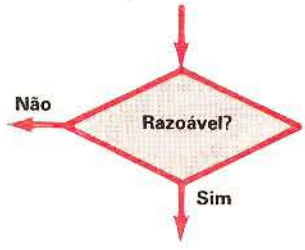
Note que poderíamos também escrever:

```
310  INPUT "DIGITE O PESO EM ONCAS..."; PON
```

No caso, decidi separar o PRINT do INPUT para mostrar como as linhas do programa correspondem ao fluxograma. Você pode, no entanto, usar qualquer uma das formas.

Vamos prosseguir. Aqui está o elemento 3:





Aqui está seu programa equivalente:

```
330  IF PON < 0 THEN GOTO 310
340  REM PESO DEVE SER POSITIVO
350  REM PODEMOS ACRESCENTAR AQUI UMA MENSAGEM EXPLI-
          CITA
```

O equivalente para o elemento 4 é:

```
360  PG = PON * 28.35
```

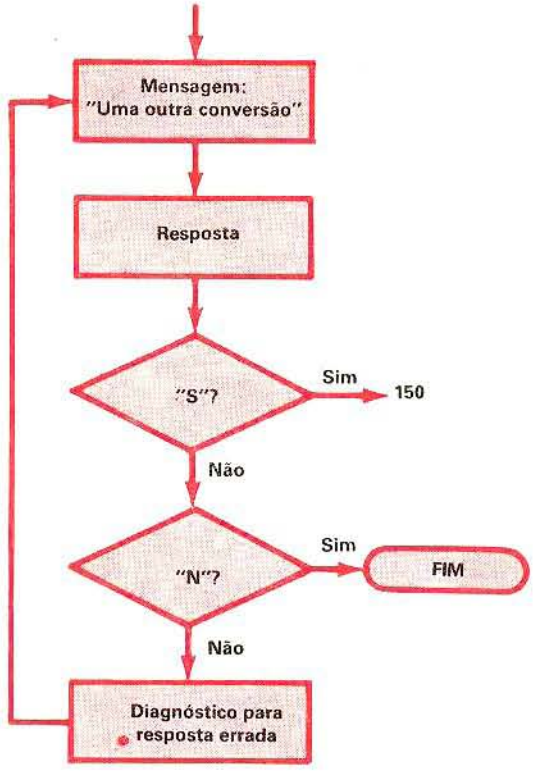
e para o elemento 5:

```
370  PRINT PON; "ONCAS SAO EQUIVALENTES A"; PG; "GRAMAS"
```

e para o elemento 6:

```
410  PRINT "VOCE QUER CONVERSAO NOVAMENTE? S PARA SIM,
          N PARA NAO;"
420  INPUT NOVAMENTE$
430  IF NOVAMENTE$ = "S" THEN GOTO 150
440  IF NOVAMENTE$ = "N" THEN END
450  REM ENTRADA NAO FOI SEM S OU N..DIGA AO USUARIO
460  PRINT "S OU N, POR FAVOR"
470  GOTO 380
```

Se o que está acima não está claro para você, aqui está o fluxograma equivalente detalhado:



Vamos agora olhar para o lado direito do fluxograma na figura 9.1. Aqui está o equivalente ao elemento 7:

```
520  PRINT "EXECUTAREI UMA TABELA DE CONVERSAO"
530  PRINT "DE ONCAS PARA GRAMAS"
540  PRINT "DIGA ME O NUMERO MAXIMO PARA ONCAS:";
550  INPUT MAX
```

E para o elemento 8:

```
560  REM MAX DEVE SER IGUAL OU MAIOR DO QUE 1.
          SE NAO, ELE E REJEITADO
570  IF MAX<1 THEN GOTO 540
```

Para maior clareza, vamos omitir uma linha na execução, antes de imprimirmos a tabela:

```
580  PRINT
```

Lembre-se: é um PRINT vazio. Ele executa uma linha em branco. Agora, aqui está o equivalente ao elemento 9:

```
590  PRINT "ONCAS", "GRAMAS"
```

Note que usamos uma vírgula, ao invés de um ponto e vírgula, para deixar um espaço agradável entre as colunas.

E aqui está o elemento 10:

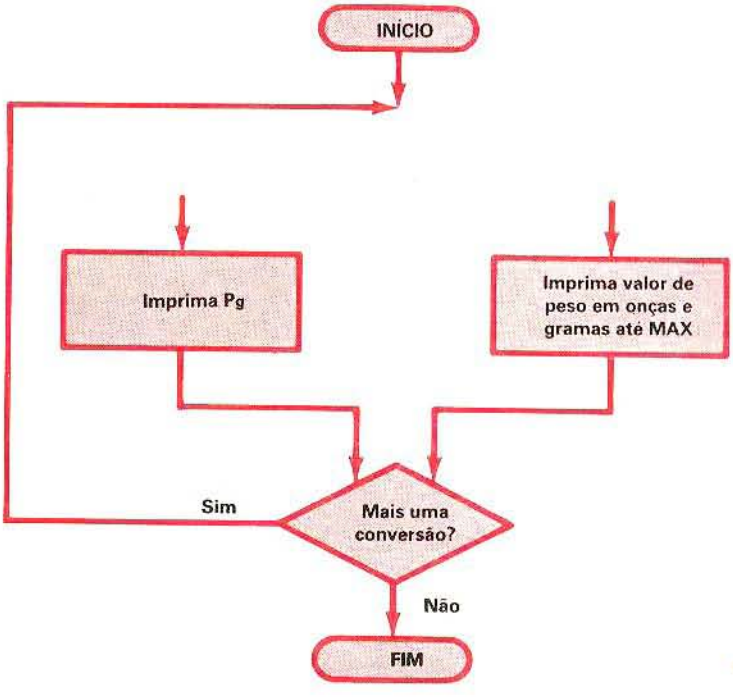
```
600  I = 1
610  PRINT I, I * 28.35
620  I = I + 1
630  IF I<= MAX GOTO 610
```

Finalmente, aqui está o elemento 11:

```
650  GOTO 380
```

O elemento 11 é o mesmo do elemento 6; então podemos simplificar nosso programa pulando (GOTO) para instrução do elemento 6.

Aqui está o fluxograma (corrigido) correspondente:



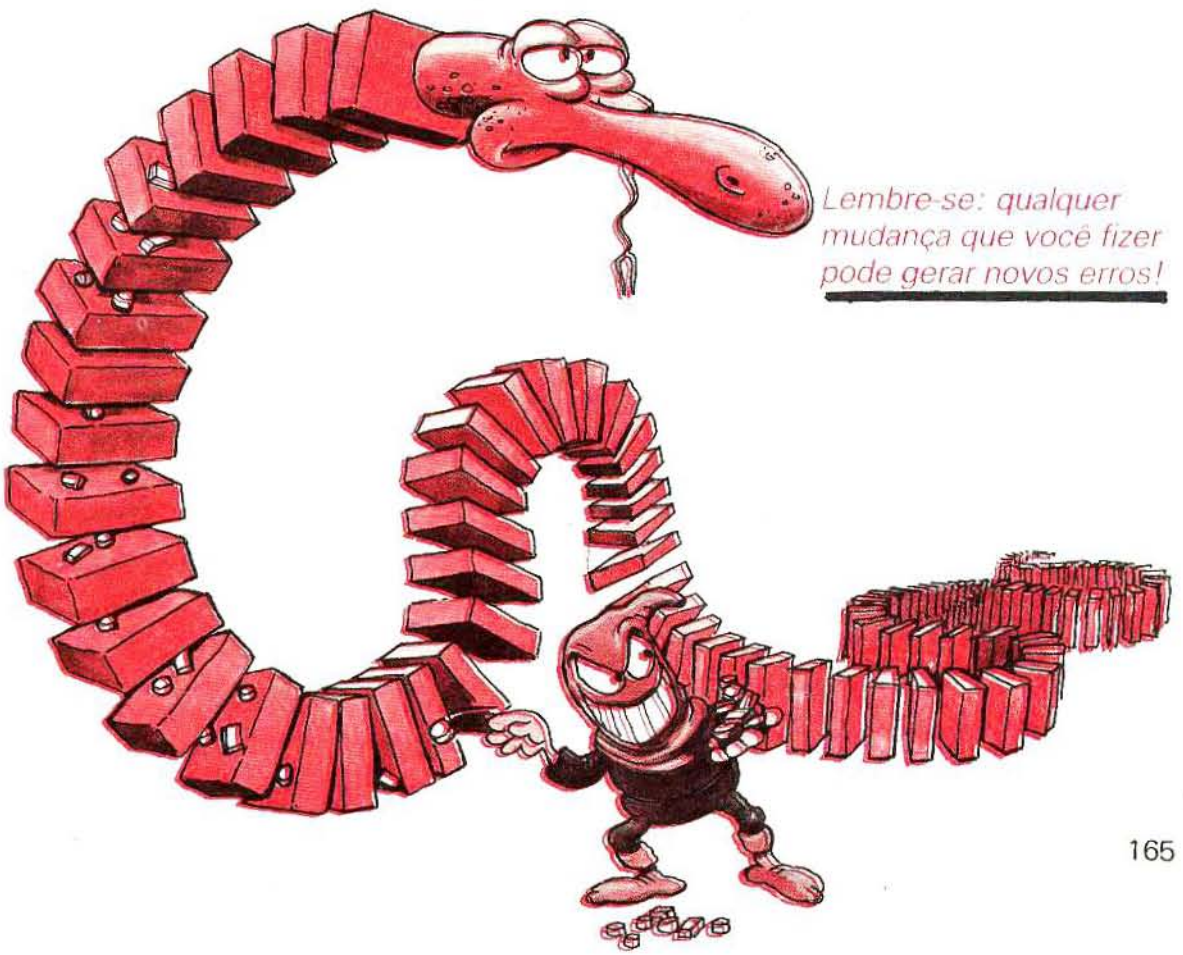
O programa completo é mostrado a seguir:

```
100 REM *** CONVERSAO DE ONCAS E GRAMAS ***
110 REM ESTE PROGRAMA REALIZA UMA CONVERSAO DIRETA
120 REM OU IMPRIME UMA TABELA DE VALORES
130 REM PRIMEIRO, ESPECIFIQUE O MODO DE CONVERSAO: DIRE-
        TA OU TABELA
140 PRINT "QUERO CONVERTER ONCAS EM GRAMAS"
150 PRINT "SE VOCE QUER CONVERTER UM VALOR DIRETA-
        MENTE, DIGITE V"
160 PRINT "SE VOCE QUER IMPRIMIR UMA TABELA DE CONVER-
        SAO, DIGITE T"
170 PRINT "SUA ESCOLHA [V OU T]";
180 INPUT ESCOLHA$
190 IF ESCOLHA$ = "V" THEN GOTO 300
200 REM ROTULO 300 E VALOR DA CONVERSAO
210 IF ESCOLHA$ = "T" THEN GOTO 500
220 REM ROTULO 500 E TABELA DE VALORES
230 REM SE O CARACTERE NAO FOR V OU T, A ENTRADA NAO E
        VALIDA
240 PRINT "V OU T, POR FAVOR:";
250 GOTO 170
260 REM
270 REM
300 REM *** CONVERSAO DE VALOR ***
310 PRINT "DIGITE O PESO EM ONCAS ...";
320 INPUT PON
330 IF PON<0 THEN GOTO 310
340 REM PESO DEVE SER POSITIVO
350 REM PODEMOS ACRESCENTAR AQUI MENSAGEM EXPLICITA
360 PG = PON * 28.35
370 PRINT PON; "ONCAS SAO EQUIVALENTES A ";P; "GRAMAS"
380 REM
390 REM *** MODULO DE SAIDA ***
400 PRINT
410 PRINT "VOCE QUER UMA OUTRA CONVERSAO? S PARA SIM, N
        PARA NAO:"
420 INPUT NOVAMENTE$
430 IF NOVAMENTE$ = "S" THEN GOTO 150
440 IF NOVAMENTE$ = "N" THEN END
450 REM ENTRADA NAO E S NEM N. DIGA AO USUARIO
460 PRINT "S OU N, POR FAVOR"
470 GOTO 380
480 REM
490 REM
500 REM *** TABELA DE CONVERSAO ***
510 REM REQUER O LIMITE MAXIMO
```

```
520  PRINT "QUERO EXECUTAR UMA TABELA DE CONVERSAO"
530  PRINT "DE ONCAS PARA GRAMAS"
540  PRINT "DIGA ME O NUMERO MAXIMO DE ONCAS:";
550  INPUT MAX
560  REM   MAX DEVE SER IGUAL OU MAIOR QUE 1. SE NAO, E RE-
           JEITADO
570  IF MAX<1 THEN GOTO 540
580  PRINT
590  PRINT "ONCAS", "GRAMAS"
600  I = 1
610  PRINT I, I * 28.35
620  I = I + 1
630  IF I<= MAX GOTO 610
640  REM I E AGORA>MAX. ESTE E O FIM DA TABELA
650  GOTO 380
660  END
```

Algumas melhorias foram acrescentadas. Por exemplo: vários REMs foram acrescentados para maior clareza (veja instruções 260, 270, 300, 380, 390, 480, 490, 500).

Outras melhorias podem ser feitas. Por exemplo: as instruções 600 e 630 podem ser substituídas por uma instrução FOR ... NEXT. Isto pode melhorar a legibilidade, mas à custa de um duro trabalho.



Lembre-se: qualquer mudança que você fizer pode gerar novos erros!

Lembre-se: qualquer mudança que você introduza para fazer funcionar um programa pode levar a novos erros. Mude seu programa apenas se houver um claro benefício.

Agora temos um programa completo. Vamos tentar usá-lo.

O Teste

Vamos executar (RUN) o programa. Aqui está uma amostra:

```
RUN
QUERO CONVERTER ONCAS EM GRAMAS
SE VOCE QUER CONVERTER UM VALOR DIRETAMENTE,
DIGITE V
SE VOCE QUER IMPRIMIR UMA TABELA DE VALORES, DIGITE T
SUA ESCOLHA [V OU T]? V
DIGITE O PESO EM ONCAS ...? 3
3 ONCAS SAO EQUIVALENTES A 85.05 GRAMAS
```

Aqui está uma outra:

```
VOCE QUER UMA OUTRA CONVERSAO? S PARA SIM, N PARA
NAO:? S
SE VOCE QUER CONVERTER UM VALOR DIRETAMENTE,
DIGITE V
SE VOCE QUER IMPRIMIR UMA TABELA DE VALORES, DIGITE T
SUA ESCOLHA [V OU T]? T
QUERO EXECUTAR UMA TABELA DE CONVERSAO
DE ONCAS EM GRAMAS
DIGA ME O NUMERO MAXIMO PARA ONCAS:? 4
      ONCAS          GRAMAS
        1             28.35
        2             56.7
        3             85.05
        4            113.4
```

Nosso programa parece funcionar tanto para conversão simples como para tabela.
Vamos tentar enganá-lo:

```
VOCE QUER UMA CONVERSAO? S PARA SIM, N PARA NAO:? S
SE VOCE QUER CONVERTER UM VALOR DIRETAMENTE,
DIGITE V
SE VOCE QUER IMPRIMIR UMA TABELA DE VALORES, DIGITE T
SUA ESCOLHA [V OU T]? V
V OU T POR FAVOR: SUA ESCOLHA [V OU T]? V
DIGITE O PESO EM ONCAS ...? 7
7 ONCAS SAO EQUIVALENTES A 198.45 GRAMAS
```

Vamos tentar a opção repetida:

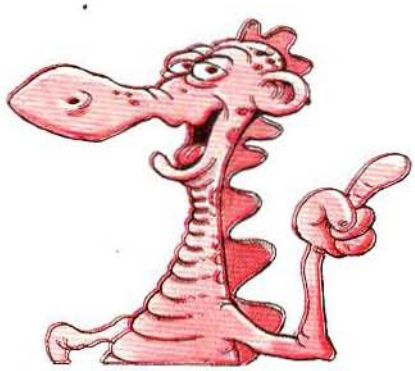
```
VOCE QUER UMA OUTRA CONVERSAO? S PARA SIM, N PARA
NAO:? S
SE VOCE QUER CONVERTER UM VALOR DIRETAMENTE,
DIGITE V
SE VOCE QUER IMPRIMIR UMA TABELA DE VALORES, DIGITE T
SUA ESCOLHA [V OU T]? V
DIGITE O PESO EM ONCAS ...? 85
85 ONCAS SAO EQUIVALENTES A 2409.75 GRAMAS

VOCE QUER UMA OUTRA CONVERSAO? S PARA SIM, N PARA
NAO:? S
SE VOCE QUER CONVERTER UM VALOR DIRETAMENTE,
DIGITE V
SE VOCE QUER IMPRIMIR UMA TABELA DE VALORES, DIGITE T
SUA ESCOLHA [V OU T]? V
DIGITE O PESO EM ONCAS ...? 2.5
2.5 ONCAS SAO EQUIVALENTES A 70.875 GRAMAS

VOCE QUER UMA OUTRA CONVERSAO? S PARA SIM, N PARA
NAO:? S
SE VOCE QUER CONVERTER UM VALOR DIRETAMENTE,
DIGITE V
SE VOCE QUER IMPRIMIR UMA TABELA DE VALORES, DIGITE T
SUA ESCOLHA [V OU T]? V
DIGITE O PESO EM ONCAS ...? 3.1
3.1 ONCAS SAO EQUIVALENTES A 87.885 GRAMAS
```

Vamos tentar enganá-lo de novo:

```
VOCE QUER UMA OUTRA CONVERSAO? S PARA SIM, N PARA
NAO:? S
SE VOCE QUER CONVERTER UM VALOR DIRETAMENTE,
DIGITE V
SE VOCE QUER IMPRIMIR UMA TABELA DE VALORES, DIGITE T
SUA ESCOLHA [V OU T]? V
DIGITE O PESO EM ONCAS ...? -5
DIGITE O PESO EM ONCAS ...? □
```

Bem, ele parece funcionar. Mas você deve realmente testá-lo muitas vezes mais, antes de ficar completamente satisfeito.

Neste caso, fomos bastante cuidadosos — e muito felizes. Nosso programa funcionou direto na primeira vez.



Sumário

Neste capítulo ilustramos a seqüência completa necessária para escrever um programa que resolva um problema dado. Você agora pode fechar este livro; desenhar seu próprio fluxograma e convertê-lo em um programa que funcione.

A chave do sucesso em programação é praticar.

Exercícios

9.1: Acrescente neste programa a opção para converter gramas em onças.

9.2: Desenvolva o programa incluindo a conversão de distância.

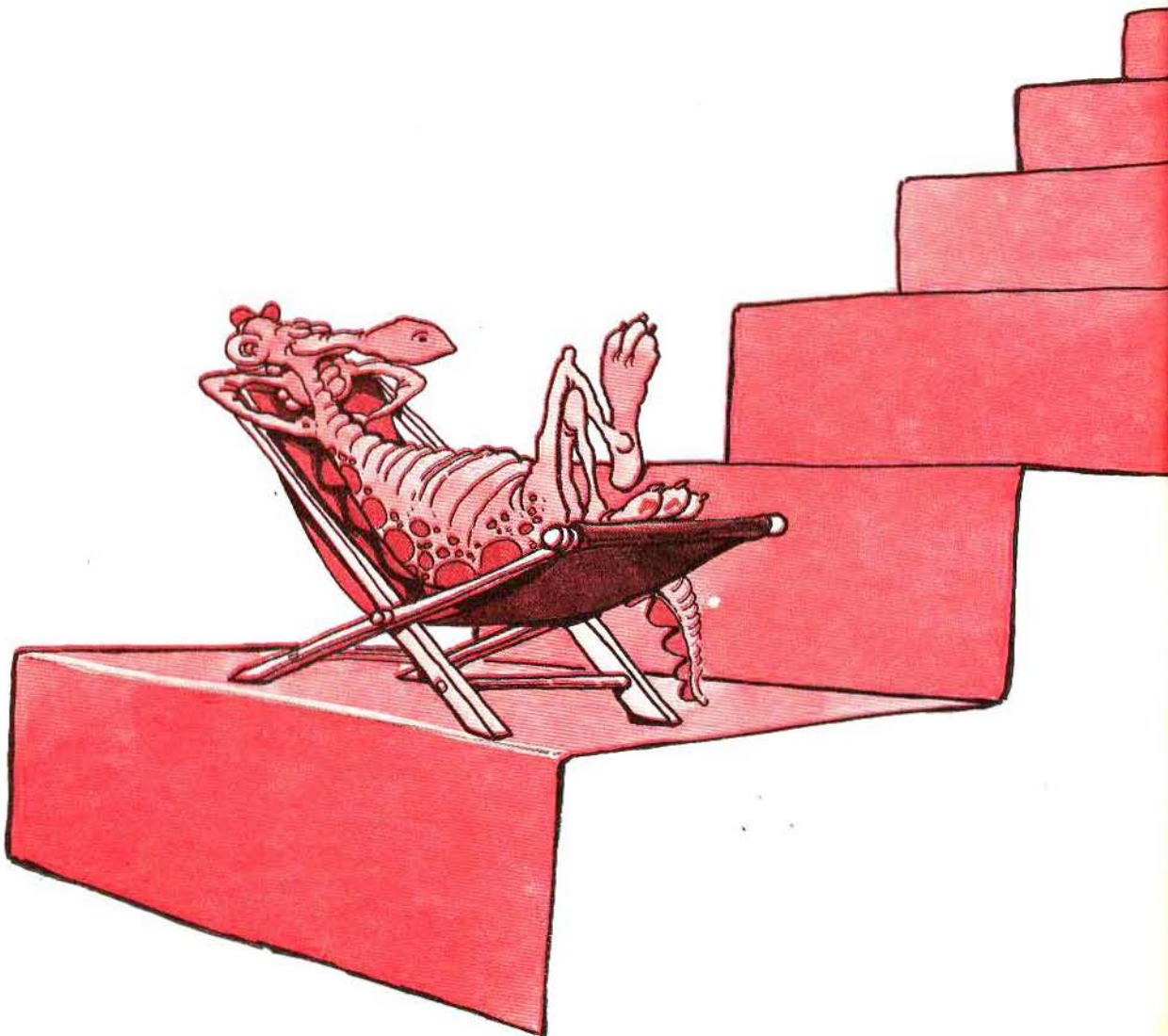
1 metro = 39.37079 polegadas
1 km = 0.62138 milhas
1 polegada = 25.3995 mm
1 pé = 30.479 cm
1 jarda = 0.91438 m
1 milha = 1609.3149 m

9.3: Desenvolva o programa, incluindo a conversão de temperatura:

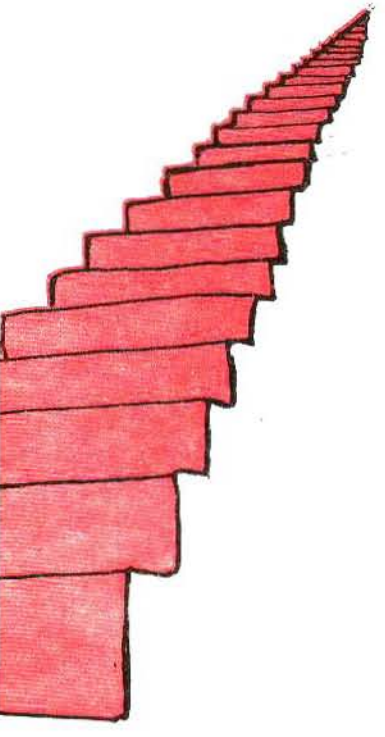
$C = (F - 32) \times 5/9$
$F = (9/5) \times C - 32$

9.4: Sugira meios adicionais para aperfeiçoar ou clarificar seu programa final.

O Próxin



ıo Passo



10

Você agora aprendeu a escrever seu próprio programa em BASIC. Neste capítulo examinaremos o próximo passo que você deve dar no sentido de melhorar suas habilidades de programação. Reveremos o que você pode fazer com o BASIC e, então, descreveremos experiências e técnicas adicionais que podem ajudá-lo a escrever programas complexos, mais facilmente.

O que Você Pode Fazer com o BASIC

Você pode escrever um programa BASIC para automatizar muitas tarefas, a não ser que elas requeiram cálculos matemáticos muito precisos, tomadas de decisões complexas ou uma resposta muito rápida (como o controle do processo em tempo real). Você descobrirá que o BASIC está bem preparado para aplicações simples de negócios; como processamento de dados, mala-direta e cálculos financeiros comuns. Com extensões de linguagem, o BASIC ainda se presta bem para gráficos e jogos.

Outras áreas de aplicação típicas para o BASIC incluem educação computadorizada, arquivo pessoal e de nogócios, cálculos matemáticos e técnicos com um grau limitado de precisão, e muito mais. As aplicações são geralmente limitadas, principalmente pelas habilidades ou técnicas de programação.

Com o conhecimento que adquiriu até agora, você está apto a escrever uma grande variedade de programas BASIC. No entanto, na medida do seu progresso, você vai querer melhorar suas habilidades e usar ferramentas de programação mais convenientes e poderosas. Este é o tópico das próximas seções.

Melhore suas habilidades

Aperfeiçoando suas Habilidades

Você pode dar três passos essenciais para incrementar suas habilidades em BASIC:

1. Ganhar mais prática
2. Obter um conhecimento melhor dos recursos completos de sua versão particular de BASIC
3. Aprender técnicas adicionais de programação

Vamos rever um passo de cada vez.

Pratique mais

Mais Prática

A chave do sucesso efetivo em programação é a prática: escreva tantos programas quanto possível e coloque-os para trabalhar. Desenvolva bons hábitos de programação, seguindo todas as recomendações apresentadas nesse livro. Se seus programas funcionam consistentemente, após apenas algumas tentativas, você pode, por si mesmo, tornar-se um efetivo e disciplinado programador. Se não, reveja seus hábitos, ou talvez releia partes desse livro, e então pratique um pouco mais. **Lembre-se:** não há outros meios para se tornar um bom programador, a não ser escrevendo muitos programas. Esse livro lhe deu um começo, mas nunca poderá ser um substituto para a experiência real.

Características Específicas do BASIC

Cada intérprete BASIC provê capacidades específicas, incluindo instruções, comandos, facilidades de simplificação, extensões para o BASIC padrão, (como gráficos e som), e um ambiente operacional (incluindo comandos para armazenamento em disco ou cassete, facilidades para aperfeiçoamento, etc...). Você pode intensificar sua habilidade de programação, aprendendo essas características e facilidades adicionais. Na seção seguinte, descreveremos as instruções adicionais BASIC encontradas na maioria dos intérpretes. Você ganhará, aprendendo sobre elas.

Técnicas Adicionais

Desde que tenha desenvolvido programas mais longos (digamos, maiores do que uma página), você vai querer aprender as técnicas usuais para resolver problemas comuns tais como: ordenar itens,

classificá-los, formatar dados, arquivamento e criação de estruturas de dados. Estes tópicos estão abordados em livros de programação.

Estude mais

Mais BASIC

Cada intérprete BASIC oferece uma longa série de facilidades "padronizadas", além de muitas extensões que são específicas do intérprete. As facilidades comuns oferecidas pela maioria dos intérpretes BASIC incluem tudo que já estudamos, além de seis tipos de instrução. Apresentaremos uma breve visão de cada um desses tipos adicionais. Você os estudará por si mesmo, ou com um livro mais avançado. Eles são:

1. Funções: Funções são *expressões embutidas* no BASIC (*built-in*) ou *criadas pelo usuário* (*user-defined*), que operam sobre uma variável dada, e realizam um cálculo ou ação específica. O BASIC sempre provê algumas funções embutidas tais como: ABS, COS, EXP, IN, FIX, RND, SGN, SIN, SQR ou TAN. Estas funções automaticamente realizarão tarefas comuns.

Funções adicionais podem ser definidas pelo usuário. Vamos examinar algumas das funções embutidas mais comuns:

- A função INT calcula a parte inteira de um número decimal pela eliminação da parte fracionada do número. Por exemplo, INT (1.234) produz o valor 1.
- Similarmente, ABS calcula o valor absoluto. Por exemplo, ABS (−5, 2) é 5.
- A função SQR calcula a raiz quadrada de um número. Por exemplo, SQR (4) produz o valor 2.

Funções podem ainda ser definidas pelo usuário. Uma função definida pelo usuário é essencialmente uma fórmula escrita pelo usuário, que tem um nome, opera sobre uma variável, e pode ser usada no programa, várias vezes. Assim, toda vez que o nome da função é usado, a fórmula é calculada com o valor corrente da variável. Esta é uma forma de simplificar um programa.

Aqui está um exemplo de uma função definida pelo usuário que calcula 2% de X:

```
10  DEF FNA (X) = X * 2 / 100
```

E aqui está uma maneira de usar esta função:

```
20  PRINT FNA (50)
```

A sua execução imprimirá o valor 1.

Vamos olhar estas instruções mais detalhadamente. FN quer dizer função, FNA é função A, isto é, o nome da função. X é a variável "postiça". X não tem um valor quando a DEFinição é escrita. O valor real ou variável é substituído por X ao mesmo tempo em que a função é utilizada; por exemplo: ele pode aparecer como FNA (50).

2. Sub-rotinas: Uma sub-rotina é um grupo de instruções ao alcance de um programa BASIC, que tem seu próprio nome e que pode ser usado repetidamente, simplesmente escrevendo este nome. Assim, cada vez que o nome da sub-rotina é usado no corpo do programa, todas as instruções incluídas na sub-rotina são executadas. Sub-rotinas são convenientes para executar repetidamente um segmento do programa sem ter que repetir as instruções do programa cada vez que são requeridas. Aqui está um exemplo de sub-rotina:

```
500  REM ESTE E UMA SUB-ROTINA DE MEDIA
510  PRINT "QUERO CALCULAR A MEDIA DE A E B"
520  PRINT "A = "; A; "B = "; B
530  MEDIA = (A + B) / 2
540  PRINT "A MEDIA E"; MEDIA
550  RETURN
```

Desde que A e B tenham recebido um valor, esta sub-rotina pode ser chamada por uma instrução como:

```
50  GOSUB 500
```

Mais tarde, ela pode ser utilizada, de novo, no programa com uma instrução do tipo:

```
170  GOSUB 500
```

e produzirá um valor diferente, se A e B forem diferentes. A utilização de sub-rotinas clarifica seu programa, encurta-o e poupa tempo. Você pode ainda construir uma biblioteca de sub-rotinas e usá-las em vários programas. No entanto, seja cuidadoso com nomes de variáveis e números de instrução.

3. Operações com cadeia de caracteres: (*strings*): operações de *strings* permitem-lhe manipular textos convenientemente, operando sobre cadeia de caracteres. Tais operações podem incluir modificação, medição, conexão de cadeias, secção e inserção de textos à esquerda, à direita ou em qualquer posição dada, substituição do caractere ou ainda comparação de cadeias de caracteres (*strings*). Estes operadores são úteis para aplicações em processadores de textos e de palavras.

4. Estruturas de dados: o BASIC permite variáveis subscritas, com um ou dois subscritos. Estas variáveis correspondem às matrizes ou vetores matemáticos ou tabelas. O uso de variáveis permite a você criar estruturas como listas e, então, convenientemente se referenciar a qualquer elemento dentro da lista. Por exemplo: os elementos de uma lista chamados CLIENTE podem ser referenciados como CLIENTE (1), CLIENTE (2) etc. e você pode imprimir todos os 10 valores escrevendo:

```
50  FOR N = 1 A 10
60  PRINT CLIENTE [N]
70  NEXT N
```

Você pode comparar 2 elementos consecutivos escrevendo:

```
100  IF CLIENTE [K] < CLIENTE [K + 1] THEN 500
```

Isto é uma grande conveniência.

5. Arquivos: Cada BASIC de disco provê facilidades para armazenar e/ou recuperar dados de/para os arquivos em disco. Estas facilidades são específicas de seu computador e do seu intérprete, e são descritas nos documentos do fabricante.

6. Recursos adicionais: Recursos adicionais típicos, que estão geralmente disponíveis em bons intérpretes BASIC, incluem as seguintes instruções: ON ... GOTO, READ, ... DATA, e PRINT USING. Você deve explorar estas facilidades com o seu manual do intérprete, ou um livro, pois eles tornarão mais fácil o ato de programar.

Também outras facilidades mais simples podem ser utilizadas; por exemplo: você pode ter permissão para digitar:

```
?25 * 32
```

obtendo dessa forma um modo conveniente de usar seu computador como uma calculadora de bolso.

Além do mais, o modo de precisão dupla pode ser fornecido para melhorar a precisão de cálculos numéricos e as facilidades de tabulação (TAB) de forma a facilitar a impressão de tabelas. Finalmente lembre-se de que há comandos específicos para o intérprete, para gerar gráficos, produzir efeitos sonoros e facilitar a edição — por exemplo, existem comandos que permitem-lhe modificar seu programa e os arquivos por ele criados — como auxílio na execução e na correção dos erros de um programa. Tais facilidades incluem controle da tela, estabelecimentos de pontos de interrupção nos quais o programa parará automaticamente, para mostrar os valores de variáveis sele-

cionadas durante a execução e, dessa forma, acelerando ou reduzindo a velocidade das mostras da tela.

Conclusão

O propósito deste livro foi ensinar-lhe rápida e efetivamente como escrever seu primeiro programa BASIC. Se você realmente se interessou pelo BASIC, vai querer aprender mais. Seu próximo passo deve ser trabalhar em todos os exercícios e desenvolver alguns de seus próprios programas.

Conforme progrida, você vai querer aprender técnicas mais avançadas.

Espero que você concorde que aprender BASIC pode ser fácil e agradável e gostaria que seu desejo agora seja fazer e aprender mais. Apreciaria ouvi-lo se você tem alguma sugestão para possíveis melhoramentos neste livro.

Apêndice A

Respostas aos Exercícios Selecionados

2

2-2:

```
10  PRINT "AAAAA"
20  PRINT "BBBB"
30  PRINT "CCC"
40  PRINT "DD"
50  PRINT "E"
```

2-4: **a**. Em BASIC, um rótulo é uma linha numerada, e deve preceder cada instrução que é parte de um programa.

b. O modo de execução programada é a maneira normal, na qual um programa é introduzido. As informações BASIC são digitadas em linhas numeradas e memorizadas pelo computador para execução posterior.

c. Execução imediata é a maneira pela qual uma instrução é impressa sem número de linha e executada imediatamente. Isto é também chamado de *modo de calculadora*.

d. Uma instrução vazia é uma instrução que nada faz. Tipicamente é um número de linha ou um rótulo que aparece sozinho — sem mais nada na mesma linha, ou com REM.

e. Um cursor é um símbolo visual especial na tela (assim como um quadrado ou uma linha sublinhar) que mostra a posição corrente na tela. Ele geralmente brilha para realçar sua visibilidade.

f. A tecla de controle CTRL, quando pressionada ao mesmo tempo que uma tecla alfabética, produzirá um comando específico. A tecla de controle torna mais fácil emitir comandos freqüentemente usados pelo computador, visto que apenas duas teclas têm que ser pressionadas.

g. O "KEY PAD" é um teclado. É geralmente um pequeno conjunto de teclas com propósitos especiais, posicionadas à direita do teclado principal, que são usadas para cálculos numéricos e posicionamento do cursor.

h. Uma palavra reservada é um nome que tem um significado específico para o BASIC. Não pode ser usado como nome de variável pelo programador.

i. O símbolo > é um caractere ou mensagem gerada por um programa, que indica que o programa espera que o usuário introduza a informação. A maioria dos BASIC usa o símbolo ">", que significa "digite sua próxima instrução". O símbolo "?" significa "dê-me um INPUT".

2-6: Sim, mas esta parece uma forma deselegante de fazê-lo, pois se você desejar executar (RUN) um programa novamente, você deve digitar novamente as instruções. Além disso, ramos condicionais (a serem cobertos mais tarde) não funcionam corretamente quando digitados dessa maneira.

2-8: Sim, o BASIC inserirá cada instrução em seu lugar apropriado, dentro do programa, de forma a que todas as instruções estejam em ordem numérica.

2-10: Não. Você deve digitar:

```
PRINT  "EXEMPLO"
```

2-12: Para eliminar a instrução 20 em um programa, digite uma instrução vazia com o rótulo 20.

3

3-2:

```
PRINT  1 + (1/2) * (1/(1 + (1/2)))
```

ou

```
PRINT  1 +.5/(1 +.5)
```

3-4:

```
PRINT  100 * 1.6
```

3-6:

```
PRINT  350 / 55
```

4

4-1:

```
10  INPUT A, B, C, D
20  SOMA = A + B + C + D
30  MEDIA = SOMA / 4
40  PRODUTO = A * B * C * D
50  PRINT SOMA, MEDIA, PRODUTO
60  END
```

4-2:

a = não	e = sim	i = sim
b = sim	f = sim	j = não
c = não	g = não	k = sim
d = sim	h = não	l = sim

4-4:

```
10  PRINT "DE ME O NOME DE UM OBJETO";
20  INPUT O$
30  PRINT "DE ME O NOME DE UMA PECA DE MOBILIA";
40  INPUT M$
50  PRINT "DE ME O NOME DE UM AMIGO";
60  INPUT A$
70  PRINT
80  PRINT "POR ACASO SEU AMIGO "; A$; " TEM UMA "; O$;
        " SOBRE A "; F$; "?"
90  END
```

4-6: b, c, f são válidos.

5

5-2:

a = sim	d = sim
b = sim	e = não
c = sim	f = não

5-4:

```
INPUT "SEU NOME: "; NOME$
INPUT "O QUE E 2 + 3?"; C
INPUT "OS DOIS ULTIMOS NUMEROS SAO... "; A, B
```

5-6: A = 3

6

6-1: A instrução IF permite que o programa tome decisões, mudando dessa forma seu comportamento, de acordo com as variações dos dados introduzidos ou dos valores calculados.

6-3:

a = sim
b = sim
c = sim
d = sim
e = não
f = sim
g = sim

6-4: Sim

6-5: Um laço no programa *loop* executa repetidamente uma parte do programa. De forma a impedir um *loop* infinito (aquele que não pára de se repetir), um teste deve ser sempre feito dentro do *loop*, o qual, quando bem-sucedido, permite que o programa saia fora do *loop*.

7

7-2:

```
10  INPUT "HORAS, MINUTOS :"; HORAS, MINUTOS
20  REM VALIDACAO DA ENTRADA
30  IF HORAS > = 0 AND HORAS < 72 AND
        MINUTOS > = 0 AND MINUTOS < 60 THEN 70
40  PRINT "INCORRETO, TENTE NOVAMENTE"
50  GOTO 10
60  REM IMPRIMA LINHAS DE H
70  IF HORAS = 0 THEN 110
80  FOR A = 1 TO HORAS
90  PRINT "H";
100 NEXT A
110 PRINT
120 REM IMPRIMA LINHAS DE M
130 IF MINUTOS = 0 THEN 170
140 FOR A = 1 TO MINUTOS
150 PRINT "M";
160 NEXT A
170 PRINT
180 END
```

7-4: Um salto pode ser feito para dentro de um laço que não seja executado por uma instrução FOR ... NEXT.

7-6:

```
10  PRINT "PROGRAMA PARA SOMAR TODOS OS NUMEROS INTEI-
        ROS PARES"
```

```
20   PRINT "PARA UM VALOR INTRODUZIDO PELO USUARIO"
30   INPUT "O MAIOR NUMERO INTEIRO PAR PARA SOMAR"; NUM
40   REM TESTE PARA VALIDAÇÃO DE ENTRADA
50   REM INT (NUM / 2) <> NUM / 2 CHECA PARA NUMERO PAR —
          VEJA CH. 10
60   IF NUM > 9 AND NUM < 10000 AND INT (NUM / 2) < NUM / 2 THEN 90
70   PRINT "NUMERO ERRADO ... TENTE NOVAMENTE"
80   GOTO 30
90   REM FACA TABELA
100  PRINT "NUMERO", "SOMA"
110  PRINT
120  FOR N = 1 TO NUM STEP 2
130  SOMA = SOMA + N
140  PRINT N, SOMA
150  NEXT N
160  END
```

7-8:

```
10  INPUT "TAXA DO IMPOSTO EM PERCENTUAL"; TAXA
20  IF TAXA < 1 OR TAXA > 100 THEN 10
30  PRINT "PRECO", "TAXA", "PRECO + TAXA"
40  FOR PRECO = 1 TO 100
50  PRINT PRECO, PRECO * TAXA / 100, PRECO + PRECO * TAXA / 100
60  NEXT PRECO
70  END
```

8

8-2: Codificar é um passo de programação. Programar envolve projetar o algoritmo, fluxograma, codificar, corrigir erros e documentar.

8-4: Uma variável é rastreada pela inserção da instrução PRINT que mostra o valor da variável em pontos críticos de um programa. Algumas vezes em comando TRACE é fornecido pelo intérprete para facilitar.

8-6: Um fluxograma é um diagrama simbólico, mostrando a seqüência de eventos que ocorrem durante a execução de um programa.

8-8: Programas claramente escritos e fáceis de entender são, portanto, fáceis de modificar. Isto permite não só ao programador que criou o programa, mas a outros programadores, mudá-lo facilmente.

Apêndice B

Palavras Reservadas mais Comuns em BASIC

Esta lista ajudará você a evitar o uso de nomes ilegais para variáveis, mesmo que algumas delas não sejam reservadas pelo seu intérprete.

ABS
AND
ARCCOS
ARCSIN
ARCTAN
AT
AUTO
BREAK
CALL
CHR$
CLEAR
CLOCK
CLOSE
CLS
COLOR
CON
CONT
COPY
COS
DATA
DATE
DEFDBL
DEFFN
DEFOMT
DEFSNG
DEFSTR
DEFUSR
DELETE
DIM
DSP
EDIT
ELSE
END
ENTER
EOF
ERR
ERROR
EXP
FIELD
FIX
FN
FOR
FORMAT
FREE
FUNCTION
GET
GOSUB
GOTO
GRAPHICS
HUN
IF
INKEY$
INP
INPUT
INSTR
INT
KILL
LEFT$
LET
LSET
LEN
LINE
LIST
LN
LOAD
LOC
LOG
LPRINT
MEM
MERGE
MID$
MKD$
MKI$
MKS$
NEW
NEXT
NOT
ON
OPEN
OR
OUT
PAUSE
PEEK
PLOT
POINT
POKE
POP
POS
POSN
POWER
PRINT
PUT
RANDOM
READ
REM
RENAME
RESET
RESTORE
RESUME
RETURN
RIGHT$
RND
RSET
RUN
SAVE
SET
SGN
SIN
SLOW
SQR
STEP
STOP
STRING$
STR$
TAB
TAN
TEXT
THEN
TIME$
TO
TRACE
TROFF
TRON
UNPLOT
USING
USR
VAL
VARPTR
VERIFY
VLIN
VTAB

Apêndice C

Glossário BASIC

algorithm (*algoritmo*) Uma seqüência de passos que especificam a solução para um problema dado.

alphanumeric (*alfanumérico*) Um conjunto de caracteres alfabéticos e numéricos.

assignment (*designação*) A operação de dar um valor para uma variável, que é indicado pelo símbolo "=" em BASIC

BASIC (*Beginners All-Purpose Symbolic Instruction Code*) Uma linguagem para programação de alto nível destinada ao fácil aprendizado.

binary (*binário*) Um sistema numérico que usa apenas dois dígitos: 0 e 1.

bit Contração das palavras binário e dígito. Um bit pode ter valor 0 ou 1.

bug (*erro*) Um erro no programa. Erros podem ser prevenidos, o que é melhor do que saná-los.

byte Um grupo de oito bits.

chip Um circuito integrado em um pequeno quadrado de silicone montado em um invólucro de plástico ou de cerâmica.

coding (*codificação*) O ato de converter um algoritmo ou um fluxograma numa seqüência de instruções para o programa. Este é um dos estágios do processo de programação.

command (*comando*) Uma palavra reservada, usada para tarefas específicas, como: limpeza da tela, dar início a um programa, ou pesquisar os arquivos. Especificamente, um comando ativa um programa especializado dentro do computador para executar a tarefa.

computer (*computador*) Uma caixa fechada contendo ao menos uma unidade central de processamento, uma memória, interfaces-padrão que permitam-no comunicar-se com o mundo exterior, e uma fonte de energia. A caixa pode ainda incluir um teclado, uma tela e uma unidade de disco. Um computador é capaz de armazenar programas e executá-los. Ele se comunica com o mundo exterior através dos dispositivos de entrada e saída. O dispositivo de entrada usual é o teclado. O dispositivo de saída usual é a tela que também pode ser a impressora. Uma memória adicional é, usualmente, fornecida na forma de unidades de disco ou de cassetes.

CPU CENTRAL PROCESSING UNIT (*Unidade Central de Processamento*) Um módulo eletrônico com a responsabilidade de buscar, decodificar e executar uma seqüência própria, as instruções armazenadas na memoria. Na maioria dos pequenos computadores, a CPU e a memória residem em um único cartão de circuito ou *card*. A CPU normalmente usa um *chip* microprocessador e alguns outros componentes.

CRT CATHODE RAY TUBE (*Tubo de Raios Catódicos*) Uma tela de televisão.

Cursor Um símbolo usado para indicar a posição corrente de um caractere que está sendo executado na tela. Na maioria das vezes um quadrado luminoso ou um traço de sublinhar. Teclas especiais geralmente estão disponíveis para posicionar, convenientemente, o cursor na tela.

data Qualquer texto ou número que o programa pode operar.

debugging (*eliminação de erros*) Ação de remoção dos erros (bugs) de um programa. Esta remoção pode ser árdua e todos os esforços devem ser usados para projetar um programa correto, para que se tenha o menor número de erros possível.

disk (*disco*) Meio magnético, no qual os dados e os programas podem ser armazenados. No disco, a informação é organizada em arquivos que podem ser recuperados pelo nome. Os discos podem armazenar uma grande quantidade de dados e são uns dispositivos comuns de memória de massa usados em pequenos computadores.

disk-drive (*unidade de disco*) O mecanismo usado para ler e escrever em um *disk*.

diskette Um disco maleável, isto é, um disco leve de 8" ou 5-1/4" destinado a fornecer um meio de armazenagem barata para programas e dados.

double precision (*precisão dupla*) Número que tem duas vezes mais dígitos que uma representação normal. Cada intérprete representa números com um conjunto de dígitos. A precisão dupla é usualmente exigida para cálculos científicos e cálculos de negócios que envolvem números longos e cálculos extensos.

editor Programa projetado para facilitar a entrada e modificação do texto. Usado para remover erros ou fazer mudanças enquanto se digita um programa.

empty statement (*instrução vazia*) Uma instrução que não faz nada. Por exemplo, a instrução:

```
10 REM
```

pode ser usada para prover um espaço no programa e facilitar a leitura.

endless loop (loop *sem fim*) Um *loop* que não tem limite e se executa eternamente. Este é um erro comum cometido pelos programadores quando não há um teste de condição incluído no *loop* ou quando o teste é sempre bem-sucedido ou sempre falha. Interromper um *loop* sem fim exige o uso de um caractere especial de interrupção — quase sempre o CTRL C.

expression (*expressão*) Uma combinação de operandos ou variáveis separados pelos operadores. Uma expressão representa uma fórmula e realiza um cálculo específico. Quando uma expressão é avaliada pelo intérprete em tempo de execução, ela resulta em um valor.

file (*arquivo*) Uma coleção de informações a qual se deu um nome. Um programa é usualmente armazenado num disco *disk* como um arquivo.

fixed point number (*número de ponto fixo*) Um número inteiro, isto é, o número todo, onde o ponto decimal está numa posição fixada à direita do último dígito.

floating point number (*número de ponto flutuante*) Um número decimal. Um número fixo de dígitos é usado internamente para representar qualquer número; e à medida que as operações são realizadas sobre o número, a posição do ponto decimal (vírgula) flutua à esquerda ou à direita.

flowchart (*fluxograma*) Representação simbólica visual de um algoritmo.

graphics (*gráfico*) Desenhos, figuras ou símbolos mostrados na tela, geralmente usando um padrão de pequenos pontos uns ao lado dos outros. O uso de cor realça a aparência dos gráficos.

hardware O equipamento, incluindo o computador, os discos, e qualquer outro item, que componha um sistema de computador. *Contrasta com: software* (isto é, a instrução ou os programas).

high-level language (*linguagem de alto nível*) Uma linguagem de programa destinada a facilitar o ato de dar instruções ao computador. O BASIC é uma linguagem de alto nível.

IC Um *circuito integrado*.

initialization (*inicialização*) A fase (num programa) durante a qual os valores iniciais são designados para as variáveis. Qualquer *loop* requer uma fase de inicialização.

instruction (*instrução*) Uma ordem válida dada ao intérprete que afetará os dados a serem operados. Cada instrução precedida de um número de linha é parte de um programa. (Comandos não afetam os valores dos dados. Eles realizam tarefas banais ou facilitam o projeto ou o uso de um programa.)

integrated circuit (*circuito integrado*) Também chamado IC. Um circuito eletrônico com vários transistores e funções lógicas colocadas em uma pequena peça de silício.

interface Circuitos eletrônicos que permitem a conexão de um dispositivo específico com o computador. Um disco, uma impressora e um gravador requerem circuitos eletrônicos específicos.

interpreter (*intérprete*) É o programa, responsável pela tradução das instruções de uma linguagem de programação (como BASIC) para a linguagem binária do computador e por sua execução. A partir do momento em que se instala o intérprete no computador, ele passa a entender as instruções BASIC.

I/O *Input/Output (E/S Entrada/Saída)* Isto é, as comunicações de e para o computador.

jump (*salto*) Um desvio ou ramificação, isto é, executar uma instrução fora de seqüência.

K 1024 É lido como K ou Kilo. K é usado para determinar o tamanho da memória, usualmente em *bytes*.

label (*rótulo*) Um número de linha em BASIC.

loading (*carregamento*) A transferência de dados ou de um programa para a memória interna do computador.

logical Uma variável ou expressão que assume o valor verdadeiro ou falso.

logical expression Uma combinação de operandos ou variáveis separadas por operadores relacionais (=, >, etc.). O valor de uma expressão lógica é verdadeira ou falsa.

loop Uma seqüência de instruções de programa que se executa repetidamente, até que ocorra um evento específico, usualmente, até que uma variável alcance um determinado valor.

machine language (*linguagem de máquina*) A linguagem binária que o computador entende diretamente, isto é, uma série limitada de instruções que manipulam a informação binária dentro da CPU e da memória.

memory (*memória*) Meio de armazenar informação. A memória interna usualmente reside ao lado da CPU, e armazena programas e dados como *bytes*. As unidades de memória de massa são os cassetes e os discos.

microcomputer (*microcomputador*) Um computador que usa um microcomputador como sua unidade central de processamento.

microprocessor (*microprocessador*) Um circuito integrado que realiza a maioria das funções da CPU, em um único *chip*. Um microprocessador moderno incorpora às vezes dezenas de milhares de transistores dentro de um único *chip* e pode até incorporar a memória.

monitor Um programa mínimo requerido para operar um sistema de computador. O monitor lê os caracteres do teclado, executa-os na tela e realiza a transferência básica de dados entre o teclado, a tela e os periféricos comuns.

MPU Unidade microprocessadora. Um *chip*.

nested loop (*laços encadeados*) Um *loop* embutido dentro de outro *loop*.

non-resident (*não-residente*) Um programa que normalmente não está na memória permanente (ROM) de computador. Um programa não-residente pode ser armazenado em cassete ou *diskette*.

operating system (*sistemas operacionais*) Um programa que realiza tarefas básicas, que fornece extensas facilidades para realizar todas as transferências e processamento comuns de dados necessários para o uso conveniente de recursos de um sistema de computador. O sistema operacional maneja arquivos em disco, realiza conversões de formato, começa e pára programas.

operator (*operador*) Um símbolo que representa qualquer operação válida sobre valores (isto é, + (soma) * (multiplicação), etc.).

peripheral (*periférico*) Dispositivo conectado ao computador, como uma impressora, uma unidade de disco ou um terminal.

program (*programa*) Seqüência de instruções a ser executada pelo computador. Cada programa é escrito numa linguagem específica de computador e deve ser carregado na memória do computador na ordem em que deva ser executado.

RAM RANDOM ACESS MEMORY (*Memória de Acesso Direto*) A parte (modificável) da memória do computador que permite ler e escrever (sendo ROM o restante).

relational operator (*operador relacional*) Um operador lógico que estabelece uma relação entre os valores.

reserved word (*palavra reservada*) Uma palavra que tem um significado pré-definido para o intérprete BASIC. Não pode ser usada pelo programador como um nome de variável.

resident (*residente*) Um programa que é armazenado permanentemente na memória do computador, isto é, em ROM.

ROM READ-ONLY MEMORY (*Memória que Apenas Lê*) Esta memória não pode ser mudada pelo programador. Ela geralmente contém parte ou tudo do monitor e, algumas vezes, um intérprete BASIC simples.

RUN (*Comando de Execução*) Comando que começa com a execução de um programa BASIC.

software Os programas.

statement (*instrução*) Uma instrução BASIC ou um comando, que é parte de um programa.

string (*cadeia de caracteres*) Uma seqüência de caracteres (contrasta com: um número). Em BASIC, o nome de uma variável é diferente, dependendo se ela contém uma cadeia de caracteres ou um número. Por exemplo, WORD$ é uma cadeia de caracteres enquanto que NUMBER é uma variável numérica.

syntax (*sintaxe*) Um conjunto de regras que definem as instruções aceitáveis para uma linguagem de computador. O BASIC tem uma sintaxe simples. O intérprete sempre verifica e indica os erros de sintaxe.

terminal Combinação de uma tela e um teclado ou uma impressora e um teclado, usado para se comunicar convenientemente com um computador.

variable (*variável*) Locação de memória que tem um nome e pode assumir sucessivos valores a qualquer tempo. O BASIC distingue entre variáveis alfanuméricas e variáveis numéricas. O nome de uma variável alfanumérica deve terminar com um $.

Índice