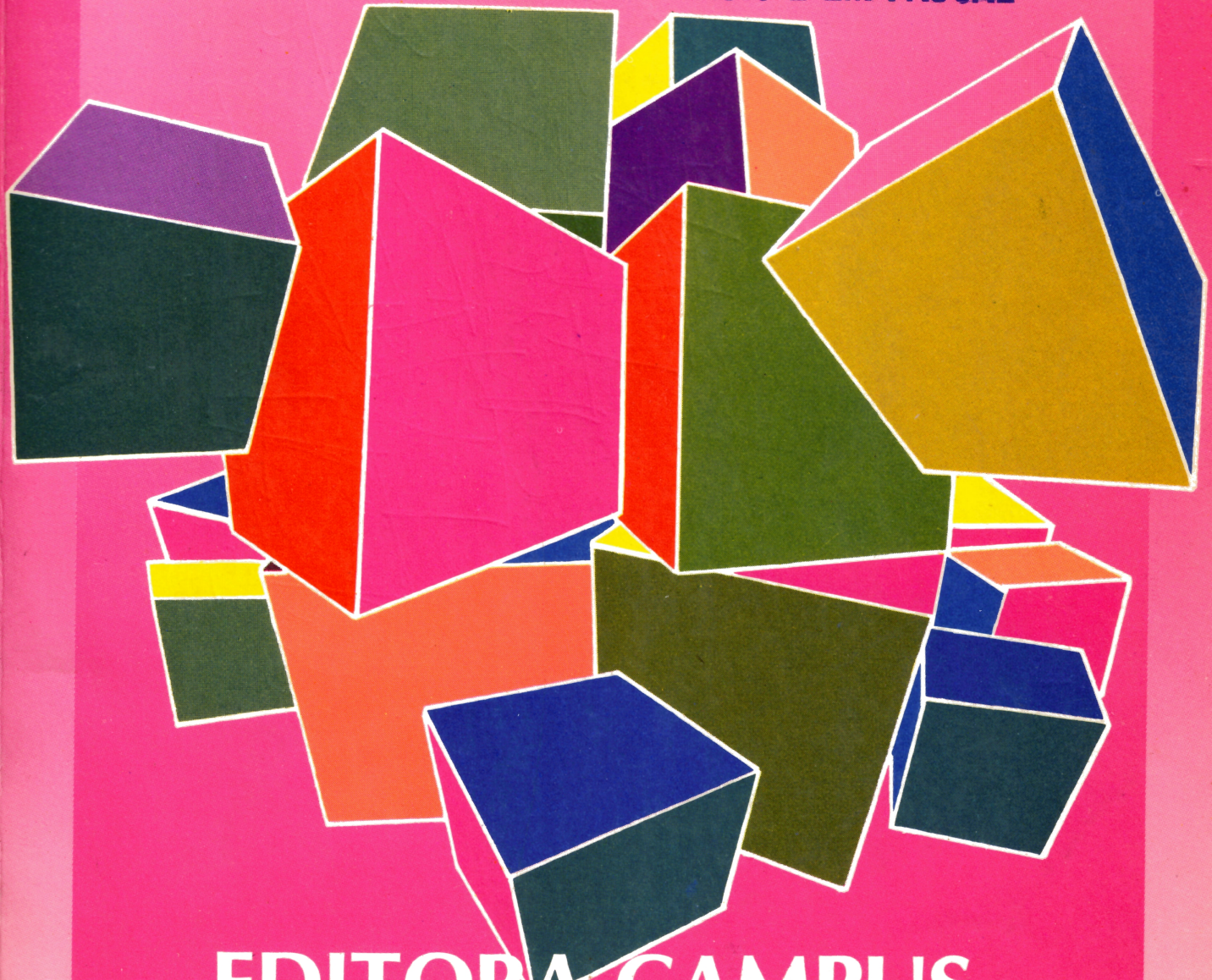
# COMO CONSTRUIR UM PROGRAMA

Jack Emmerichs

PROGRAMAS-EXEMPLO EM BASIC E EM PASCAL



EDITORA CAMPUS

# COMO CONSTRUIR UM PROGRAMA

*Jack Emmerichs*

**PROGRAMAS-EXEMPLO EM BASIC E EM PASCAL**

Tradução
Luiz Orlando Coutinho Lemos

Editora Campus Ltda.
*Rio de Janeiro*

Do original:
How to Build a Program.



Capa
Otavio Studart

Diagramação e revisão
Editora Campus Ltda.
Qualidade internacional a serviço do autor e do leitor nacional.
Rua Barão de Itapagipe 55 Rio Comprido
Tel.: (021) 284 8443 Telex (00038) 021-32606
20261 Rio de Janeiro RJ Brasil
Endereço Telegráfico: CAMPUSRIO

Composição e paginação
Linolivro Composições Gráficas Ltda.
Tel.: (021) 284 6017

ISBN 85-7001-222-5
(Edição original: ISBN 0-88056-068-1 dilithium Press, Oregon, USA.)

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

Emmerichs, Jack.
E46c Como construir um programa / Jack Emmerichs ; tradução de Luiz Orlando Coutinho Lemos. – Rio de Janeiro : Campus, 1987.

Tradução de: How to build a program
Apêndices
Glossário
Bibliografia
ISBN 85-7001-222-5

1. Computadores eletrônicos digitais – Programação I. Lemos, Luiz Orlando Coutinho II. Título

85-0009 CDD – 001.642

# Sobre o Autor...

Formado em Arquitetura pela Universidade Politécnica do Estado da Califórnia, Jack Emmerichs fez a seguir o mestrado em Administração de Tecnologia na Universidade Americana, em Washington, D.C. Escreveu muitos artigos para a revista *Byte* e para *Interface Age,* sendo autor também de *The Tiny Assembler 68000.*

Jack Emmerichs trabalhou em projetos de computadores para o governo americano, e em projetos e desenvolvimento de sistemas financeiros para A.O. Smith Corporation. Atualmente, é o vice-presidente da Software Services for Advanced Technology Associates, em Milwaukee, Wisconsin.

# Sumário

# Introdução

*"Antes de prosseguirmos, ouça-me."*
Shakespeare

Este é um livro sobre como se desenvolver um programa de computador, e destina-se a pessoas que não possuam nenhuma experiência anterior com computadores. Seu objetivo não é ensinar a escrever programas em BASIC, Pascal ou qualquer outra linguagem de computador, mas mostrar como desenvolver a estrutura e a lógica detalhada de um programa. Uma vez entendida esta parte, a criação do programa em si será muito mais fácil, e levará a menos erros do que se começássemos logo a escrever as instruções do programa.

Para quem já começou a criar seus próprios programas mas tem encontrado dificuldade em compreender os termos técnicos ou como é o funcionamento de um programa, este livro terá grande utilidade. Para quem já entende esse funcionamento, mas não sabe como desenvolver os programas a partir de uma idéia inicial, este livro também será muito útil. E, ainda, para os que estão planejando comprar um computador e desejam saber exatamente como escrever programas, ele será de grande ajuda.

Desde o aparecimento do computador pessoal, um número crescente de pessoas tem se interessado em escrever seus próprios programas. Mesmo havendo muitos programas à venda, há pelo menos quatro razões pelas quais, mais cedo ou mais tarde, uma pessoa desejará ser auto-suficiente:

1. Haverá sempre tarefas de interesse pessoal, para as quais ninguém escreveu um programa;

2. Mesmo que alguém tenha escrito um programa de que uma outra pessoa necessite, pode não ser exatamente o que essa pessoa imaginara;
3. As pessoas entenderão e desfrutarão mais o computador se souberem como controlá-lo com seus próprios programas;
4. Criar um programa bem escrito pode ser um projeto agradável e gratificante.

O texto mostrará como desenvolver um programa de computador, desde a idéia original, projeto, a implementação e o uso real.

Ao contrário do que popularmente se crê, não é preciso usar palavras mágicas ou fórmulas secretas para se criar um programa de computador bem projetado e que funcione. Existem, contudo, algumas técnicas que podem tornar a programação mais proveitosa. O processo de criação de um programa é apresentado passo a passo. Cada etapa é apresentada num capítulo distinto, que se baseia em idéias e técnicas desenvolvidas em capítulos anteriores.

O processo de desenvolvimento de programa usado neste livro chama-se *programação estruturada "top-down"*. É usado em projetos que visem desde a simples verificação de saldo bancário, em computadores pessoais, até em complexas aplicações comerciais feitas por programadores profissionais. O método é chamado *"top-down"* porque começa com os requisitos gerais do projeto, e vai descendo a níveis mais apurados de detalhes até que todos os detalhes tenham sido analisados. Chama-se *estruturado* porque o processo de desenvolvimento de cada nível de detalhe é um processo bem definido.

O Apêndice A contém um plano de projeto típico para o desenvolvimento de um projeto de programação completo. À proporção que cada procedimento para projeto de programa for desenvolvido no livro, será relacionado com este plano de projeto. Mostrará como todos os passos se relacionam entre si e ao projeto como um todo. Ao final do livro, um jogo de VINTE-E-UM terá sido desenvolvido como um exemplo de projeto. Ele pode vir a ser um dos primeiros programas de seu arquivo pessoal de programas.

Foi minha intenção fazer um livro o mais genérico possível. Por esta razão não há análise sobre nenhum computador ou equipamento em particular. De um modo geral, o tipo de computador utilizado tem alguma influência sobre a maneira de escrever o programa final, mas não sobre o modo como este será desenvolvido.

É difícil fazer um livro de programação independente de linguagens de programação específicas. Embora a técnica da programação estruturada seja independente de qualquer linguagem em particular, os exemplos devem ser mostrados em alguma linguagem ou conjunto de linguagens para que tenham utilidade. Por isso, os exemplos mais detalhados (inclusive o programa de VINTE-E-UM) são mostrados em BASIC e PASCAL.

O BASIC é usado por duas razões. Em primeiro lugar, é a linguagem mais largamente adotada no campo dos computadores pessoais e, por isso mesmo, a mais provável de ser encontrada em pequenos sistemas. Em segundo lugar, é muito simples. Se um programa puder ser escrito em BASIC, provavelmente

poderá ser escrito em qualquer linguagem de computador. Infelizmente, versões específicas de uma linguagem de computador diferem muitas vezes de um computador para outro. Por isto as explicações neste livro usarão as versões mais simples da BASIC.

A linguagem Pascal é usada porque foi projetada especificamente para utilizar o tipo de projeto estruturado que empregaremos. Muitos exemplos ficarão mais claros em Pascal do que em BASIC. Pascal está se tornando uma linguagem cada vez mais popular, e já pode ser encontrada em muitos computadores pequenos.

Quem tiver um interesse especial por outra linguagem, como APL ou LISP, pode mesmo assim utilizar este livro para aprender a desenvolver e escrever um programa. De qualquer modo, nesses casos, é bom usar o guia de instruções da linguagem específica do computador que se tiver, juntamente com este livro. Aqui são ensinados os princípios do bom projeto e desenvolvimento de programas. Esses princípios serão uma grande ajuda para uma programação mais fácil e mais segura em qualquer linguagem.

# 1

# Que é Programa?

*"Um pouco de instinto e algumas regras claras."*
William Wordsworth

O primeiro passo para se aprender a desenvolver um programa de computador é saber o que é um programa. Neste primeiro capítulo, definimos o que é um programa e damos alguns exemplos. (Alguns deles estarão em lugares inesperados.) Desenvolveremos também os diversos componentes ou módulos de um programa, examinaremos sua estrutura e veremos como podem ser combinadas num só programa. Finalmente, veremos alguma imitação natural do que pode ser programado com êxito.

## I. DEFININDO PROGRAMA

O computador em si não possui nenhuma inteligência embutida ou capacidade para raciocinar. É, contudo, extremamente bom para seguir ordens, e fará exatamente o que se lhe ordene. Sempre! (Ou até que falte energia ou que a máquina enguice.)

Desde que as instruções possam ser realizadas, o computador jamais as questiona, por mais extravagantes que sejam.

A lista de instruções que o computador segue para realizar uma tarefa específica chama-se *programa.* O programa é armazenado na memória do computador para que a máquina possa lê-lo e realizá-lo, ou *executá-lo,* uma instrução de cada vez. Alguns computadores podem executar comandos simples ou uma versão simples do BASIC assim que ligados. Essas máquinas contêm um programa para esses comandos, ou para BASIC, armazenado permanentemente na

memória. Este programa passa a controlar o computador logo que a máquina é ligada. Um computador sem programa é como uma semente sem água — tem grande potencial, mas não sai disso.

A verdadeira força do computador advém de duas características que ele possui ao seguir uma seqüência de instruções. Em primeiro lugar, por mais enfadonha ou repetitiva que as instruções possam parecer, o computador as executa a grande velocidade e de um modo perfeitamente controlado pelo tempo que for necessário. As instruções são seguidas na ordem estrita; mudar a ordem sem que seja dada instrução para tal está além da capacidade do computador.

Em segundo lugar, um computador de uso geral usa um tipo de memória que permite a mudança de instruções quando se quiser; o que é bem diferente das calculadoras manuais e dos jogos eletrônicos, cujas instruções não podem ser alteradas. Estes aparelhos restringem-se às tarefas para as quais foram originalmente concebidos.

O objetivo deste livro é ajudar no desenvolvimento de programas de computador. As coisas se tornarão mais fáceis, contudo, se começarmos com conjuntos de informações, ou programas, mais familiares. A partir deste ponto, empregaremos o termo *programa* para designar um conjunto de instruções ordenadas. O termo *instrução* irá designar uma instrução específica do programa e o termo *cláusula* designará uma parte de uma instrução.

## II. PROCURANDO ALGUNS PROGRAMAS COMO EXEMPLO

Há muitos exemplos de programas encontrados na vida real, mesmo que não tenham nada a ver com computadores. Um dos exemplos mais comuns é a receita do bolinho de aveia mostrada na Figura 1.1.

1 3/4 de uma xícara de aveia
2 3/4 de uma xícara mais duas colheres de farinha de trigo
3 2 colheres de sopa de açúcar mascavo
4 1 1/2 colher de chá de fermento
5 1/2 colher de chá de bicarbonato de sódio
6 1/2 colher de chá de sal
7 1 colher de chá de canela
8 1/2 xícara de margarina
9 1 ovo
10 3/4 de uma xícara de leitelho

Misture os sete primeiros ingredientes e bata bem. Ponha a margarina.

Bata o ovo e o leitelho e despeje sobre a primeira mistura.

Mexa até ficar cremoso.

Use 12 forminhas untadas, enchendo-as em 2/3 de sua capacidade. Polvilhe açúcar e canela por cima e leve-as ao forno aquecido por 15 minutos.

**Figura 1.1** — Uma receita de bolinho de aveia

Esta receita contém uma lista de ingredientes que serão usados e diz como usá-los. Seguindo-se cada passo corretamente, o resultado será o bolinho de aveia. O bolinho sairá do jeito como a pessoa que criou a receita imaginou. (A propósito, esses bolinhos são muito bons.)

A receita usa uma *linguagem* que precisa ser conhecida. Reparem que esta linguagem é um pouco diferente do linguajar normal. Expressões como "½ colher de chá de bicabornato de sódio" e "ponha a margarina" só têm sentido se a pessoa souber que tipo de bicarbonato usar e o que significa "ponha a margarina". Quando se começa a trabalhar com programas de computador, precisa-se aprender as linguagens especiais nas quais as instruções serão escritas.

Vejamos agora um programa mais complexo: as instruções para montagem de um brinquedo articulado que vem numa caixa onde está escrito "É Preciso Fazer Algumas Montagens". As instruções para montagem estão reproduzidas na Figura 1.2. Leiam-nas para ter uma noção de como elas norteiam a montagem do brinquedo. Não é importante entender cada instrução da primeira vez, já que depois cada uma delas será lida com mais vagar.

As ilustrações deste conjunto de instruções não foram incluídas aqui porque não estamos interessados em criar instruções gráficas, mas sim em criar a narrativa passo a passo.

Conquanto este conjunto de instruções seja mais complexo do que a receita do bolinho de aveia, não é, provavelmente, tão complexo quanto as instruções descobertas pelos pais que tentaram montar uma bicicleta na véspera de uma festa de aniversário. Estas instruções contém, contudo, exemplos de todos os tipos de instrução de que se precisa para criarem-se programas de computador.

## III. JUNTANDO DIFERENTES TIPOS DE INSTRUÇÃO

Vamos ver mais de perto cada tipo de instrução — ou comando — usada na montagem do brinquedo articulado. Essas instruções são módulos básicos que mais tarde serão usados para se criar um programa de computador.

### A. Instruções Diretivas

A mais simples forma de instrução em nosso exemplo é a instrução *diretiva* individual. Por exemplo:

**ABRA A EMBALAGEM DE PAPELÃO E REMOVA TODAS AS PEÇAS**

Uma instrução diretiva simplesmente diz o que deve ser feito, e são executadas em seqüência. Na Figura 1.2, instrução 15, como colocar as rodas no eixo forma um bloco de instruções diretivas. Muitos programas (a receita da Figura 1.1, por exemplo) podem ser escritos usando-se apenas instruções diretivas.

## B. Instruções Condicionais — IF-THEN

Um programa torna-se mais eficaz se puder avaliar o que ocorre num determinado momento e *desviar,* ou seja, fazer coisas diferentes sob condições diferentes. Num programa, uma condição é a pergunta que pode ter como resposta "verdadeiro" ou "falso". Por exemplo, na Figura 1.2, instrução dois, "Estamos montando um modelo W?", poderá ter como resposta "verdadeiro" ou "falso", dependendo do que tiver sido colocado na embalagem. As mesmas instruções são colocadas em todos os modelos, de forma que a pessoa deve determinar que modelo está montando. Outras condições são testadas nas instruções 2, 4, 6, 11 e 12.

FRAMMIS GENERAL CORPORATION
1 INDUSTRIAL PARK, EMERALD CITY, OZ 00100

MODELOS ARTICULADOS *A, B* E *W*

"INSTRUÇÕES DE MONTAGEM"

1. Abra a caixa de papelão e remova todas as peças.
   (NOTA: COMECE PELA INSTRUÇÃO N.º 1)
2. Se você tiver um modelo *W*, certifique-se então de que retirou todas as peças que estão por baixo dos suportes de espuma, no fundo da caixa.
3. Se estiver faltando alguma das peças relacionadas na lista de peças, então:
   - Recoloque todas as peças na caixa.
   - Feche a caixa e a leve ao revendedor para buscar as peças que estão faltando ou trocar por outra caixa.
4. Se você tiver um modelo *B* e ele tiver o suporte do arco à esquerda, então aparafuse a braçadeira do lado esquerdo no suporte. Caso contrário, aparafuse a braçadeira do lado direito no suporte.
5. Com as cinco rodas, faça o seguinte:
   - Prenda o pneu de borracha no aro da roda.
   - Prenda a roda num dos cinco eixos laterais como descrito na Instrução 15 (COLOCAÇÃO DAS RODAS NOS EIXOS).
6. Se você tiver um modelo *A* ou um modelo *B*, desvie para a Instrução 9 (QUER DIZER, PULE AS INSTRUÇÕES 7 E 8).
7. Prenda uma roldana no eixo superior da polia da bomba e outra no eixo inferior, como descrito na Instrução 15 (COLOCAÇÃO DAS RODAS NOS EIXOS).
8. Coloque a correia V em volta das duas roldanas colocadas.
9. Solte o parafuso do suporte da barra traseira esquerda, até que o braço de suspensão superior esteja livre para mover-se.

10. Gire o ajuste de nível até os indicadores de nível dos suportes laterais, como aparece na Figura VII-I-A. Aperte, então, o parafuso do suporte da barra traseira esquerda.
11. Se o braço de suspensão não se mover livremente entre as guias do conjunto do braço de suspensão e o suporte lateral, volte à Instrução 9 e tente o processo de ajuste novamente.
12. O braço de suspensão deve mover-se livremente também entre as guias do conjunto da barra traseira localizados no final do braço. Se não se mover, aperte o botão vermelho de auto-alinhamento no lado direito do conjunto da cabeça enquanto o braço não se mover livremente nos dois conjuntos de guias.
13. Instale o fusível no suporte seguindo a tabela abaixo:
    MODELO *A*: FUSÍVEL DE 10A
    MODELO *B*: FUSÍVEL DE 15A
    MODELO *W*: FUSÍVEL DE 13,4A
14. A montagem terminou. Veja o manual do operador para conhecer as instruções sobre o uso adequado do produto. (NOTA: O PROCEDIMENTO A SEGUIR DEVE SER SEGUIDO NAS CINCO RODAS.)
15. Colocação das rodas nos eixos:
    - Introduza uma arruela de aço número 1277996-ABH/47 na ponta do eixo.
    - Gire a roda na ponta do eixo.
    - Coloque uma segunda arruela de aço número 1277996-ABH/47 e uma porca castelo número 2 na ponta do eixo.
    - Aparafuse a porca e depois dê meia-volta atrás.
    - Coloque um contrapino número CP-1 no meio da porca castelo, como mostrado na ilustração VII-I-A.
    - Continue com a montagem no ponto onde você parou.

**Figura 1.2** — Instruções de Montagem de um Brinquedo Articulado

Instruções deste tipo são chamadas IF-THEN ou *condicionais*. A mais simples instrução IF-THEN tem duas partes: a condição de ser testada (IF) e a ação a ser tomada quando a condição for verdadeira (THEN). Se a condição testada for verdadeira, a cláusula *THEN* é executada; caso contrário, saltamos a cláusula THEN e partimos para a próxima instrução. Na instrução dois da Figura 1.2, a condição testada é se esse modelo é o W. Se for verdadeira a resposta, então deve-se procurar as peças adicionais no fundo da caixa. Notem que num idioma informal como o Inglês, a palavra THEN (então) pode ser deduzida ao lermos as instruções 6, 11 e 12. Numa linguagem formal de computador, contudo, o THEN será normalmente incluído de alguma forma.

Instruções IF-THEN podem se tornar mais úteis se checarem diversas condições numa só instrução. Chama-se a isso de *expressões condicionais*. O item 4 da Figura 1.2 contém uma instrução condicional que combina dois testes simples: se você tem o modelo B *e* tem a opção para canhoto. A palavra E (AND) unindo as duas condições significa que ambas devem ser verdadeiras para que a ação do THEN seja tomada. Se as duas condições fossem unidas pela palavra OU (OR), significaria que "quando uma das condições for verdadeira, realize a ação do THEN". Se uma das condições fosse precedida da palavra NÃO (NOT) significaria que "a condição não (NOT) deve ser satisfeita para que se tome a ação do THEN". Assim sendo, a "condição não" é verdadeira apenas onde a "condição" é falsa. Aqui estão algumas instruções IF-THEN complexas usando AND, OR e NOT.

Se (IF) você tiver um modelo B, E (AND) ele tiver o suporte do arco à esquerda, aparafuse então (THEN) aparafuse a braçadeira do lado direito no suporte.

Se (IF) tiver um modelo A, OU (OR) um modelo B, pule então (THEN) à instrução 9.

Se (IF) você estiver com algumas peças sobrando E (AND) ainda NÃO (NOT) desistiu deste projeto, ENTÃO (THEN) leia novamente as instruções para ver onde essas peças devem ser colocadas.

Como criar expressões condicionais de forma que possam significar exatamente o que se espera delas será mostrado mais adiante.

É importante perceber que as expressões condicionais são apenas uma forma conveniente de se criarem instruções IF-THEN mais eficazes. Por exemplo, as simples instruções condicionais a seguir têm o mesmo significado dos três exemplos anteriores:

Se (IF) você tiver um modelo B ENTÃO (THEN) SE (IF) ele tiver o suporte do arco à esquerda, aparafuse então (THEN) a braçadeira do lado direito no suporte.

Se (IF) você tiver um modelo A, ENTÃO (THEN) pule para a instrução 9.

Se (IF) você tiver um modelo B, ENTÃO (THEN) pule para a instrução 9.

Se (IF) algumas peças sobrarem ENTÃO (THEN) SE (IF) você NÃO (NOT) estiver cansado deste projeto ainda, releia então (THEN) as instruções para ver onde essas peças devem ser colocadas.

Como se vê, AND pode ser substituído pelo uso de uma instrução IF-THEN como a ação do THEN de outra instrução IF-THEN. Um grupo de diversas instruções IF-THEN pode ser usado para substituir OR.

Um tipo mais forte de instrução IF-THEN é chamado IF-THEN-ELSE (SE-ENTÃO-CASO CONTRÁRIO). Um IF-THEN-ELSE diz que ação tomar quando a condição testada for falsa, bem como quando for verdadeira. Na Figura 1.2, por exemplo, a instrução 4 diz que o cabo do suporte deve ser usado se a expressão condicional for falsa. Esta simples adição de uma cláusula muda a instrução IF-THEN para uma instrução IF-THEN-ELSE. Aqui temos mais alguns exemplos de IF-THEN-ELSE:

SE (IF) você tiver o modelo B, ENTÃO (THEN) aparafuse a braçadeira esquerda no suporte, CASO CONTRÁRIO (ELSE) aparafuse a braçadeira direita no suporte.

SE (IF) você tiver um modelo A ou um modelo B, ENTÃO (THEN) pule para a instrução 9, CASO CONTRÁRIO (ELSE) coloque as roldanas e a correia V.

SE (IF) você estiver com algumas peças sobrando, ENTÃO (THEN) releia as instruções para ver onde essas peças devem ser colocadas, CASO CONTRÁRIO (ELSE) dê-se por satisfeito e descanse um pouco.

Novamente é importante perceber que a cláusula ELSE apenas ajuda a fazer uma instrução IF-THEN mais fácil de ser usada. Os pares de instruções a seguir produzem exatamente os mesmos resultados dos três exemplos anteriores:

SE (IF) você tiver o modelo B à esquerda, ENTÃO (THEN) aparafuse a braçadeira esquerda ao suporte.

SE (IF) NÃO (NOT) tiver o modelo B à esquerda, ENTÃO (THEN) aparafuse a braçadeira direita ao suporte.

SE (IF) tiver um modelo A ou um modelo B, ENTÃO (THEN) pule para a instrução 9.

SE (IF) NÃO (NOT) tiver um modelo A nem um modelo B, ENTÃO (THEN) coloque as roldanas e a correia V.

SE (IF) estiver com algumas peças sobrando, ENTÃO (THEN) releia as instruções para ver onde essas peças devem ser usadas.

SE (IF) NÃO (NOT) estiver com peças sobrando, ENTÃO (THEN) dê-se por satisfeito e descanse um pouco.

Existe, contudo, uma outra maneira de aumentar a força de uma instrução IF-THEN: incluir diversas instruções na cláusula THEN (ou ELSE). Mas como isso requer o conhecimento de alguns tipos de instrução que ainda não vimos, voltaremos ao assunto mais à frente. Por ora, reparem que uma instrução IF-THEN simples é forte o bastante para criar todos os exemplos imaginários de IF-THEN-ELSE apresentados até aqui.

### C. Instruções GOTO

Todas as instruções examinadas até aqui foram colocadas de modo a que uma seguisse a outra numa ordem seqüencial normal. Às vezes, contudo, é necessário abandonar esta ordem do programa e usar instruções colocadas em outra parte. Um exemplo disso é mostrado na Figura 1.2, instrução 6, na qual diversas instruções podem ser puladas, dependendo do modelo que se estiver montando. Este é um exemplo de instrução "vá para" (GO-TO) aparecendo na cláusula THEN de uma instrução IF-THEN. A forma GO-TO é geralmente abreviada para GOTO. Uma instrução GOTO é a maneira mais simples de se mudar de uma posição para outra num programa. Ela diz simplesmente que instrução executar a seguir.

A instrução GOTO é exatamente o que precisamos ter numa instrução IF-THEN para controlar diversas instruções da cláusula THEN. Como vimos antes,

a instrução 3 para montagem do brinquedo contém uma lista de ações do THEN. Para realizar a mesma coisa, usa-se o IF-THEN-GOTO para se pular um grupo de instruções quando estas não devem ser executadas. O teste deve ser em condições que são opostas (NOT) às desejadas, se o grupo de instruções tivesse que ser executado. Isto porque estamos agora testando quando pular as instruções, e não quando executá-las. Os dois grupos de instruções a seguir produzem os mesmos resultados:

**IF-THEN Complexo**

1. Se (IF) você tiver um modelo W então (THEN)
   - Prenda as roldanas nos eixos.
   - Coloque a correia V nas roldanas.
2. Próxima instrução.

**IF-THEN Simples com GOTO**

1. Se (IF) você NÃO (NOT) tiver um modelo W ENTÃO (THEN) vá para (GOTO) a instrução 4.
2. Prenda as roldanas nos eixos.
3. Coloque a correia V nas roldanas.
4. Próxima instrução.

Outra instrução GOTO é encontrada na instrução 6 da Figura 1.2. Lá, é dito para se deixar de lado os itens 7 e 8 se tivermos os modelos A ou B. A instrução GOTO pode levar a outra instrução IF-THEN, para criar expressões condicionais AND. Um IF-THEN-GOTO é tudo de que se necessita para criar qualquer tipo de instrução IF-THEN-ELSE complexa.

As instruções GOTO podem, contudo, ser perigosas se forem usadas com muita freqüência num programa. É fácil um programador se perder num labirinto de instruções GOTO e não ter condição de dizer quando determinadas partes do programa serão usadas. O problema não é para onde ir; acontece que se perde a noção de onde se está vindo. Por isso, neste livro as instruções GOTO são usadas apenas para construir outros tipos de instruções estruturadas ou para lidar com condições de erro que requeiram uma ação especial. Numa linguagem como Pascal, que foi desenvolvida tendo por base técnicas de programação estruturada, programas inteiros podem ser escritos sem o uso de qualquer instrução GOTO. Na verdade, em algumas linguagens o GOTO não é aceito de forma alguma! No BASIC, contudo, o GOTO é necessário para se criarem alguns tipos de instrução estruturada.

## D. Instruções de Loop Simples

Mencionamos anteriormente que uma característica do computador é que ele não questiona tarefas enfadonhas ou repetitivas. A maneira mais fácil de se pro-

gramar uma tarefa repetitiva é criar um *laço* (loop) no qual as mesmas instruções sejam executadas repetidas vezes. Nas instruções de montagem (Figura 1.2), a instrução 5 cria um loop. Os itens do loop serão executados cinco vezes, uma vez para cada roda.

Um loop simples pode ser criado por uma instrução GOTO que se refira a uma instrução anterior. Por exemplo:

1. Coloque um pneu de borracha numa das rodas de aço.
2. Prenda a roda no eixo como descrito na instrução 15.
3. GOTO à instrução 1.

Este loop tem um problema maior: — não se tem como sair dele! Claro, uma pessoa poderia parar quando terminasse pneus ou eixos. Mas um computador não é tão esperto assim! Geralmente a única forma de parar a máquina é interromper o programa ou desligá-la e começar tudo de novo.

Por isso, todo loop deve ter algum tipo de controle que diga quando parar de executá-lo. O tipo mais simples de controle de loop usa um contador. O contador é um mecanismo que conta o número de vezes que o loop foi executado. Uma instrução IF-THEN faz sair do loop quando se atinge o limite estabelecido. Isso pode ser conseguido no exemplo anterior alterando-se a instrução 3:

1. Coloque o pneu de borracha numa das rodas de aço.
2. Prenda a roda no eixo como descrito na instrução 15.
3. SE (IF) cinco rodas já estiverem colocadas, vá para (GOTO) a instrução seguinte, caso contrário (ELSE) vá para (GOTO) a instrução 1.

Pode-se criar um loop com contador usando-se um simples IF-THEN combinado com GOTO. O contador deve ser acionado por cada passagem do loop e pode ser usado a partir de instruções contidas no loop. Um conjunto completo de instruções para a montagem das cinco rodas no exemplo anterior, portanto, ficará assim:

1. O primeiro eixo a se trabalhar é o número 1.
2. SE (IF) o número de eixos for maior que 5, então (THEN) vá para (GOTO) a instrução 7.
3. Coloque um pneu de borracha numa roda de aço.
4. Prenda a roda no eixo como descrito na instrução 15.
5. Acrescente um ao número de eixos contados.
6. Vá para (GOTO) a instrução 2.
7. A instrução seguinte.

Um loop com contador é tão útil na programação de computador que a maioria das linguagens de computação possui um conjunto especial de instruções para ele. BASIC e Pascal usam a instrução FOR/NEXT. PL/1 e FORTRAN

usam o loop DO. As estruturas de loop variam muito de uma linguagem para outra, mas todas apresentam as seguintes características de controle:

- A primeira instrução e a última instrução a serem controladas pelo loop devem ser identificadas.
- O contador a ser usado deve ser definido.
- Os valores inicial e final do contador devem ser definidos.
- O incremento — isto é, o valor acrescido a cada passagem pelo loop — deve ser definido.

O mais comum incremento num loop é 1, e este valor é geralmente assumido se não for indicado nenhum outro. É útil, contudo, que o contador possa ter incremento de outros valores, tais como 5 ou 10. Particularmente útil é a capacidade de contar para trás através de um incremento negativo. Por exemplo, o loop pode contar de dez a zero através de um incremento —1.

A forma geral de um loop simples, apresentando todas as características de controle, é:

Com o contador do loop mudando de um número inicial até um número final com um incremento qualquer, fazer o seguinte:
As instruções a serem executadas no loop.
Incremente o contador do loop e faça a próxima repetição.

## E. Instruções DO-WHILE e DO-UNTIL

Há dois tipos de loop simples que não são controlados por contadores. São as instruções DO-WHILE (faça enquanto) e DO-UNTIL (faça até). Em ambos os casos, uma condição é testada a cada passagem pelo loop. Esta condição determina se o loop será executado novamente ou não. É importante perceber que nem sempre se poderá saber com antecedência quantas passagens um loop terá. Os procedimentos de ajuste nas instruções de montagem são loops deste tipo. Algumas linguagens permitem que se saia dos loops simples com contador através de condições não relacionadas com o valor corrente do contador usando uma instrução especial, EXIT (saída). O processo é mais fácil de se entender, contudo, se usarmos um loop controlado por uma condição em primeiro lugar.

A instrução DO-WHILE faz com que o loop se repita enquanto a condição expressa for verdadeira. A instrução 12 da montagem é nitidamente um loop DO-WHILE:

ENQUANTO (WHILE) o braço estiver fora de ajuste, FAÇA (DO) o ajuste.

O teste da condição, que é similar à cláusula IF da instrução IF-THEN, está no início do laço. Se o braço já estiver livre para mover-se, o processo de ajuste não será executado. Enquanto o braço não estiver movendo-se livremente nos dois conjuntos de guias, o processo de ajuste continua.

A instrução DO-UNTIL faz com que o loop seja executado até que a condição dada seja falsa. As instruções de montagem de 9 a 11 são uma forma de loop DO-UNTIL:

FAÇA (DO) o ajuste ATÉ (UNTIL) o braço estar ajustado.

O teste da condição está no *final* do loop. Por isso, se a condição não for nunca falsa, o loop é executado uma vez (antes que a condição seja checada pela primeira vez). Na Figura 1.2, por exemplo, o ajuste do braço, controlado pelas instruções de 9 a 11, é feito pelo menos uma vez. Dependendo do resultado do ajuste, a instrução 11 determina se outra execução do loop é necessária.

O problema com os loops DO-WHILE e DO-UNTIL é que não há garantias de que as condições testadas sejam jamais satisfeitas. Nos exemplos acima, o que acontece se o braço não puder ser ajustado? Não há dúvida de que depois de muitas tentativas a pessoa voltará ao revendedor e tentará ajustar todo o brinquedo. O processo de ajuste foi finalmente interrompido porque a pessoa que estava seguindo as instruções possui um contador natural: o nível de frustração que interrompe a ação do loop. Já que o computador não tem nível de frustração, ficaria repetindo o loop sem saída.

Quando se tiver dúvida sobre o final adequado do loop, deve-se colocar um contador à expressão de condição para se determinar um número máximo de passagens a serem feitas. Se o limite for ultrapassado, o programa deve sair do loop com algum tipo de indicação de erro.

### F. Chamadas de Sub-Rotina

Às vezes um bloco de instruções precisa ser repetido em diversos lugares diferentes do programa. Neste caso, um único loop simples não pode controlar todas as execuções das instruções repetidas. E não é viável repetir este bloco de instruções toda vez que for necessário. Precisamos de uma localização única para este bloco, algum modo de ter acesso a ele sempre que for preciso usá-lo, e uma forma de voltar ao programa principal depois que as instruções forem executadas.

Na montagem do brinquedo, por exemplo, há dois lugares em que se deve colocar as rodas nos eixos. As instruções detalhadas para a colocação das rodas estão na instrução 15. Na verdade, essa instrução é um bloco de seis instruções separadas às quais nos referimos pela única instrução 15. Quer dizer, ela é, na verdade, um pequeno programa separado. Este bloco é *chamado* nas instruções 5 e 7. Para voltar ao programa principal, a última instrução da instrução 15 diz para se voltar ao lugar do programa onde a chamada foi feita. Por exemplo, quando a chamada é feita na instrução 7, realiza-se o grupo de instruções da instrução 15, depois volta-se ao final da instrução 7 e passa-se à instrução 8.

Pequenos segmentos do programa como este, que podem ser chamados de qualquer parte do programa, são chamados sub-rotinas. Neste livro, a instrução que transfere o controle à sub-rotina é a instrução CALL, e a instrução usada para se voltar à localização da chamada é o "RETURN".* Como o GOTO, a

---

* Os termos sub-rotina, CALL (e GOSUB) e RETURN são usados em BASIC. A linguagem Pascal usa termos diferentes para os mesmos conceitos. Consulte seu manual de linguagem de programação para maiores detalhes.

instrução CALL é um modo de mover-se de uma parte para outra do programa. Ao contrário de GOTO, contudo, CALL sempre mantém o local de onde veio e retorna para lá depois que a sub-rotina for executada.

É possível uma sub-rotina chamar outra sub-rotina. Cada sub-rotina "sabe" para onde voltar depois de terminada. Quando cada sub-rotina é chamada, o controle do programa desce à cadeia de rotinas conectadas. E quando cada rotina termina, o controle volta cadeia acima até chegar, finalmente, à chamada da primeira sub-rotina. Isso permite que se escrevam blocos de instruções em diversos níveis de detalhe, sendo que cada nível chama um nível inferior quando for necessário um detalhe maior. Como veremos mais adiante, este é o fundamento da estrutura top-down que os *programas estruturados top-down* usam.

### G. Instrução SELECT-A-CASE

Em algumas circunstâncias é preciso selecionar uma ação específica dentre uma lista de diversas ações possíveis. Na Figura 1.2, por exemplo, a instrução 13 determina (seleciona) a instalação do fusível adequado para cada um dos modelos. Este é um tipo mais amplo de estrutura IF-THEN na qual há diversos valores possíveis para a condição testada, em lugar de apenas "verdadeiro" ou "falso". Neste caso presente, é indicada uma ação para cada valor da condição.

Esta estrutura geral é chamada SELECT-A-CASE, ou simplesmente CASE. A instrução de CASE requer um item específico que possa ter uma variedade de valores, e uma tabela que relacione a ação a ser tomada para cada valor do item testado. Deve haver uma lista de apoio para o caso de o item testado não ter nenhum dos valores apresentados. Aqui está outra instrução CASE num formato mais detalhado:

```
No CASO (INCASE) de o modelo ser
    A: chamar (CALL) a rotina-A
    B: chamar (CALL) a rotina-B
    C: chamar (CALL) a rotina-C
    W: chamar (CALL) a rotina-W
    FIM (END) da instrução.
```

Notem que as instruções reais são chamadas de sub-rotina. Cada sub-rotina pode conter tantas instruções quantas forem necessárias para processar o caso devidamente. O item "modelo" pode supor o valor de qualquer modelo do fabricante. Este valor é comparado com a lista de valores da tabela e a instrução de chamada adequada é executada. O programa continua então com a instrução seguinte depois do final da instrução CASE.

Cada opção da tabela pode ser substituída por uma instrução IF-THEN separada. Por este motivo, muitas linguagens não têm instruções CASE específicas. Quando a instrução CASE é possível, como o é em Pascal, é muito mais eficiente do que múltiplas instruções IF-THEN. Mas quando não for possível,

como em BASIC, podem ser usadas múltiplas instruções IF-THEN para se atingir o mesmo resultado.

### H. Instrução STOP ou END

A última coisa de que um programa necessita é de uma indicação lógica de que o trabalho terminou. Isso requer uma instrução STOP ou END. O nome varia de uma linguagem para outra e algumas linguagens utilizam as duas. Em BASIC, por exemplo, STOP significa parar de processar mas estar pronto para prosseguir, enquanto END significa que o programa terminou. Em outras linguagens, END pode simplesmente indicar o fim de algum bloco de código e não o fim do programa. E em outras linguagens, como a Pascal, o final é indicado pela estrutura do programa, em vez de o ser por uma instrução específica. Neste livro usaremos o termo STOP porque é empregado em muitas linguagens.

Nas instruções da Figura 1.2, a instrução 14 diz que as instruções terminaram. Observem que *não* é a última instrução do programa, já que a sub-rotina para prender as rodas aparece depois da última instrução regular. Se não houvesse a instrução 14, a sub-rotina da instrução 15 seria indevidamente executada depois da instrução 13. Além disso, o RETURN no final da instrução 15 não saberia a que instrução CALL voltar.

Além de encerrar este programa, a instrução 14 diz ao leitor que há outro conjunto de instruções a ser seguido, agora que este está pronto. O final de um programa dando início à execução de outro programa é chamado encadeamento. Nem todas as linguagens admitem o encadeamento, mas ele pode ser útil se admitido. Deve-se prestar atenção em como a linguagem que usamos indica que se atingiu o fim de um programa e que o processamento deve parar (STOP).

### I. Comentários e Definições de Dados

Há dois tipos adicionais de informação que aparecem na receita do bolinho de aveia e nas instruções de montagem do brinquedo. Instruções de *Definição de Dados,* como os ingredientes da receita, definem os dados usados no programa. *Comentários,* tais como notas entre parênteses na Figura 1.2, não são executados pelo programa. Eles contêm informações para ajudar a explicar o programa a quem o estiver lendo.

A forma das definições de dados varia de uma linguagem para outra. Por exemplo, o BASIC poderá entender alguns tipos de dados de um programa sem a necessidade de definições em separado. (Tipos especiais de informação, tratados mais adiante, requerem definições especiais.) A linguagem Pascal, por outro lado, requer que *todos* os dados do programa sejam definidos antes de serem usados. A melhor maneira de explicar as instruções de definição de dados é através de exemplos. Por isso, vamos apresentar essas instruções à medida que forem necessárias, ao longo do livro.

Os comentários existem em quase todas as linguagens. Ele é ignorado pelo computador e não tem nenhum efeito na execução do programa. É usado simplesmente para tornar o programa mais significativo para quem o estiver lendo. Como veremos, os comentários são muito importantes para ajudar a pessoa a saber o que cada parte do programa está fazendo e como ela se relaciona com as demais partes. Um criterioso e largo uso de comentários é um bom hábito.

## IV. A ESTRUTURA DAS INSTRUÇÕES

Os nove tipos de instruções que vimos são tudo de que se precisa para se escrever um programa bem estruturado. Cada instrução pode ser considerada como um único bloco de instruções a ser combinado com outros blocos na elaboração do programa. Cada instrução tem um só ponto de entrada no início e um só ponto de saída no fim. Isso permite que tipos diferentes de instruções sejam combinados em seqüência com o final de uma instrução conectado ao início da seguinte. Esta estrutura é fácil de entender se a pessoa souber sempre para onde vai e de onde está vindo.

Só há uma exceção: a instrução GOTO. Uma instrução GOTO tem um único ponto de entrada no início, mas o ponto de saída pode ser em qualquer lugar do programa.

Ao contrário de CALL, GOTO não deixa indício de onde veio. Há dois usos para GOTO em programação estruturada: lidar com condições de erro e criar outras instruções estruturadas não existentes numa determinada linguagem de programação. Por isso, no restante deste livro GOTO será considerada apenas como parte de outras instruções, e não uma instrução em si.

Isso nos deixa com oito estruturas de instruções para usar na elaboração de programas. A Figura 1.3, partes A-H, mostra a estrutura de cada instrução que analisamos.

Observem que em cada diagrama a parte mostrada com tracejado forte tem um único ponto de ENTRADA e um único ponto de SAÍDA. Esses pontos de entrada e saída são os únicos lugares por onde as instruções estruturadas interagem entre si. Conectando-se blocos de instruções somente nesses pontos de ENTRADA e SAÍDA, não se interfere com os controles internos das instruções mais complexas. Muitas regras cheias de detalhes e difíceis de serem lembradas que já foram ensinadas como guias de programação são observadas simplesmente ao se unirem as instruções adequadas como as peças de um brinquedo de montar.

Cada bloco em tracejamento forte pode ser tratado como uma *caixa preta*; ou seja, um processo que tem uma única entrada, realiza algumas funções (sem termos que saber como) e tem uma única saída. Essa caixa preta pode conter em si qualquer instrução ou grupo de instruções. Isso significa que as ações do THEN de uma instrução IF-THEN-ELSE, por exemplo, podem ser uma instrução DO-WHILE, uma chamada de sub-rotina ou qualquer outro tipo de instrução.

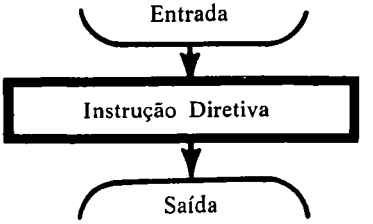



**Figura 1.3-A** — Instrução Diretiva

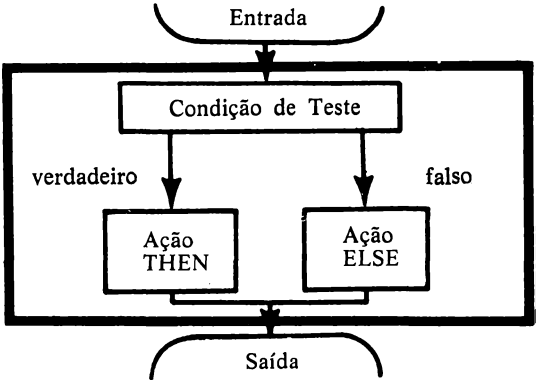



**Figura 1.3-B** — IF-THEN-ELSE

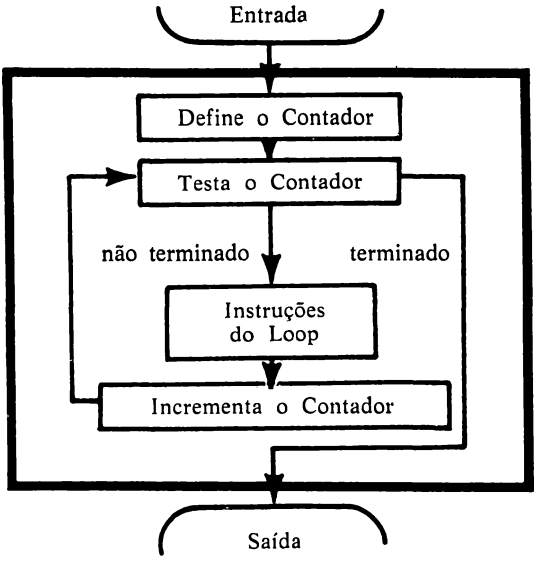



**Figura 1.3-C** — Loop com Contador

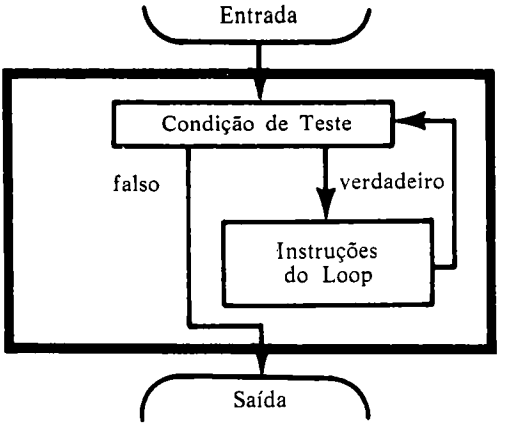



**Figura 1.3-D** — DO-WHILE

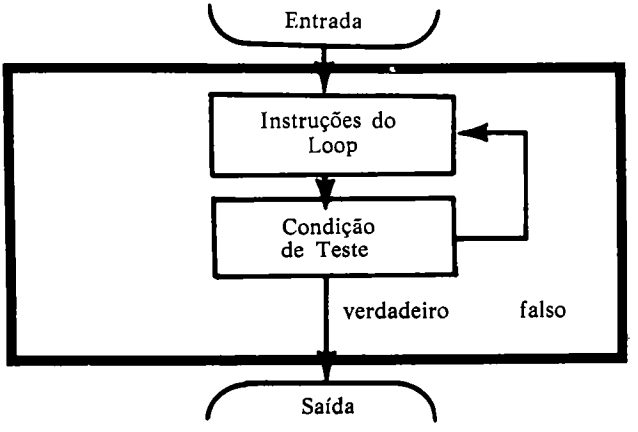



**Figura 1.3-E** — DO-UNTIL

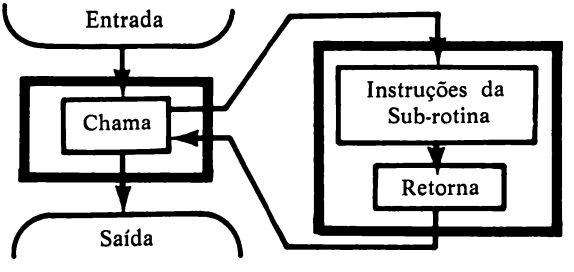



**Figura 1.3-F** — Chamada de Sub-rotina

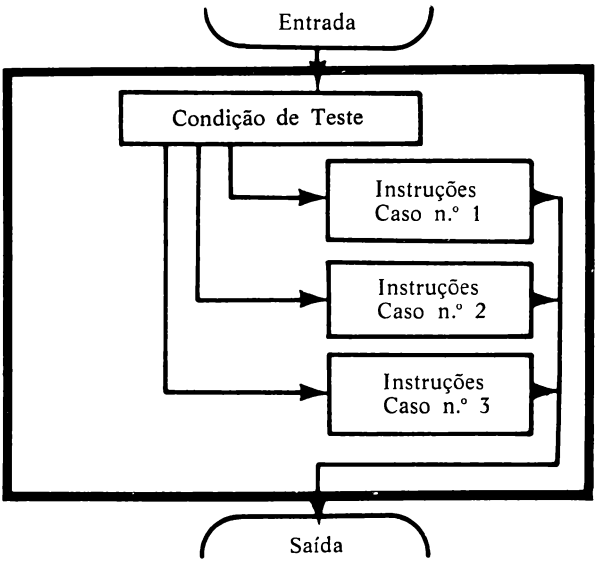



**Figura 1.3-G** — SELECT-A-CASE

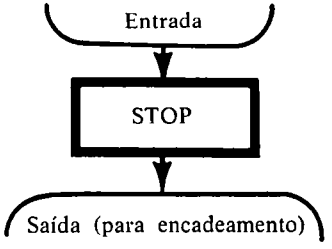



**Figura 1.3-H** — STOP

Um exemplo de como diversas instruções podem ser combinadas numa só estrutura é mostrado na Figura 1.4. Este diagrama contém indicadores de INÍCIO e de FIM, mostrando como entrar e sair do programa. O programa em si consiste num bloco de instruções diretivas, seguido de uma instrução IF-THEN-ELSE, e encerrado com uma instrução STOP. No IF-THEN-ELSE, a cláusula THEN é um bloco de instruções diretivas; a cláusula ELSE é uma chamada de sub-rotina. A sub-rotina, por sua vez, é uma instrução DO-WHILE.

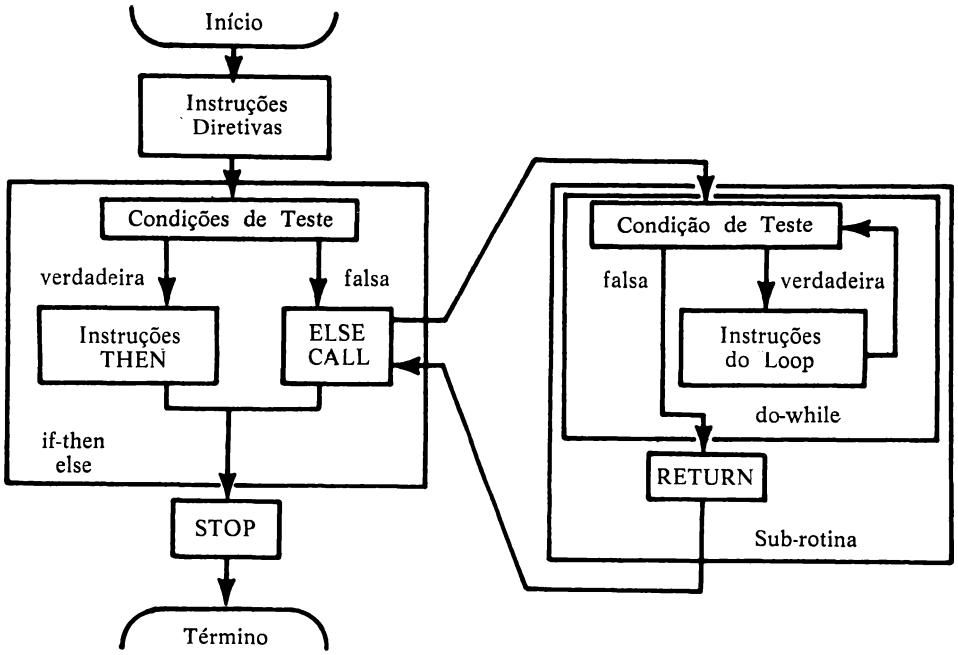



**Figura 1.4** — Instruções Complexas

Blocos de instruções podem ser construídos deste modo até o nível de detalhe desejado. Um último exemplo de instruções combinadas é mostrado na Figura 1.5. Este é o diagrama das instruções de montagem da Figura 1.2, mostrando a estrutura de cada uma das instruções.

Este processo de partir blocos grandes em outros menores, todos com a forma de instrução estruturada, deu nome à programação estruturada. Do mesmo modo, a criação de um programa a partir de blocos estruturais separados serviu como inspiração para o nome deste livro.

## V. ALGUMAS LIMITAÇÕES NATURAIS

De um modo geral, qualquer conjunto de instruções pode ser escrito utilizando-se instruções estruturadas. Há, contudo, limitações do que se pode executar com um programa adequadamente desenvolvido. Por exemplo, alguns conjuntos de instruções são tão grandes que não vale a pena desenvolvê-los. As instruções do imposto de renda podem encher livros inteiros mas são escritas por uma grande equipe.*

* Não há dúvida de que as instruções seriam mais fáceis de ser entendidas se fossem escritas como instruções estruturadas.

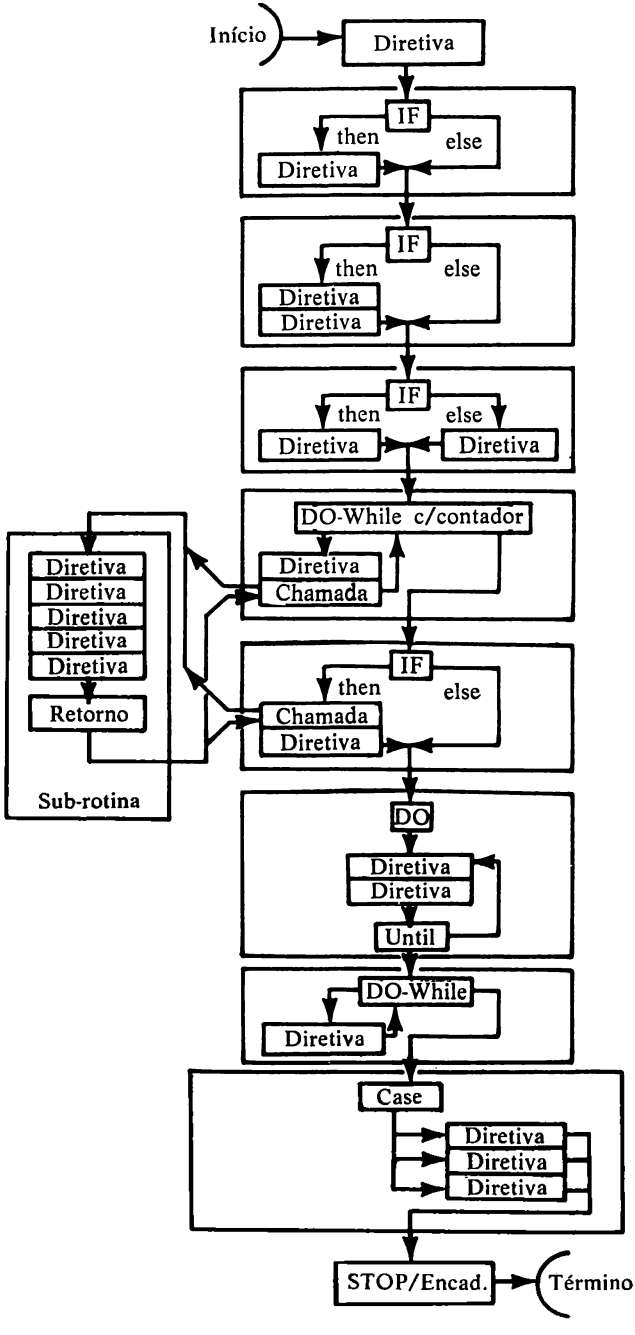



**Figura 1.5** — A Estrutura das Instruções de Montagem

É mais fácil preencher um formulário de imposto do que escrever um programa de computador que contenha todas as instruções sobre imposto. É mais provável, contudo, que limitações de tamanho tornem-se problema quando se tentar introduzir um programa muito grande na memória disponível de um pequeno computador. Haverá sempre complexos e fascinantes projetos que serão muito grandes para o computador que se esteja utilizando.

Um problema maior é quando um processo é tão difícil de ser definido que não se pode escrever, de modo algum, uma série de instruções para ele. Por exemplo, como se julgar a qualidade de um quadro, uma sinfonia ou uma peça. Mais difícil ainda é compreender o processo criativo que produz essas obras de arte. É verdade que os computadores já geraram obras de arte, música e poesia, mas essas tentativas jamais demonstraram o gênio criador de um verdadeiro artista. Há também problemas em que uma informação está sempre incompleta, tal como estratégia do mercado de ações. Nesses casos, a pessoa deve usar a intuição ou fazer suposições. Uma lista completa de instruções é simplesmente impossível. Até agora, os problemas mal definidos e os que apresentam informação insuficiente têm sido muito difíceis de ser solucionados por qualquer programa.

Essas dificuldades, contudo, são limitações da programação em geral, e não apenas da programação estruturada. Se um processo puder ser definido usando-se um grupo de instruções específicas, será programado de maneira mais fácil usando-se instruções estruturadas.

## VI. REVISÃO DAS PRINCIPAIS IDÉIAS

Este capítulo tratou da definição do que é programa e como ele é usado. Os pontos importantes a lembrar são:

- Um programa é um conjunto de instruções em ordem seqüencial.
- Todos os programas podem ser feitos com os seguintes tipos de instruções:

DIRETIVAS
IF-THEN (ELSE)
LOOPS SIMPLES
LOOPS DO-WHILE
LOOPS DO-UNTIL
CHAMADAS de SUB-ROTINA
SELECT A CASE
STOP

- A instrução GOTO não deve ser usada, exceto para construir uma das instruções acima, ou para se sair do programa em caso de condições de erro muito graves.
- Além dessas instruções pode haver instruções de definição de dados e comentários dentro de um programa.

- Cada tipo de instrução estruturada tem um ponto de entrada e um ponto de saída.
- Cada bloco de instrução composto de instruções estruturadas também tem apenas um ponto de entrada e apenas um ponto de saída.
- Qualquer instrução ou bloco de instruções pode ser ligado num bloco de instruções controladas. Os tipos disponíveis de instruções podem ser ligados uns aos outros ou reunidos seqüencialmente para formar um programa bem complexo. O fluxo de instruções de um programa deste tipo ainda assim será fácil de ser entendido.
- Esta técnica, chamada *programação estruturada,* tem sido usada para construir grandes sistemas comerciais e mostrou-se útil na programação de projetos de todos os tamanhos. Realmente funciona!

# 2

# Examinando os Programas para Computador

*"E agora vejo com olhos serenos*
*o verdadeiro pulsar da máquina."*
William Wordsworth

Agora que já vimos como escrever programas para as pessoas, vamos aprender a escrever programas para os computadores. Antes de passarmos a isso, contudo, precisamos ver como um computador funciona e o tipo de linguagem necessária para que se escrevam instruções para ele. No Capítulo Um dissemos que qualquer conjunto de instruções ordenadas é chamado programa. No final do Capítulo Dois ficará bem claro por que os programas de computador são escritos de uma determinada maneira e como funcionam. Para quem já está familiarizado com tipos de dados, constantes, variáveis, matrizes, montadores, compiladores, interpretadores e como essas coisas se inter-relacionam, este capítulo servirá como revisão. Para essas pessoas basta fazer uma rápida leitura para se certificarem de que entenderam todas as idéias aqui apresentadas.

Neste capítulo examinaremos os diferentes tipos de informação usados por um computador e como ele os usa. Definiremos o que se entende por *linguagem de computador*, como tal linguagem difere da linguagem falada normal e como se pode utilizar uma linguagem de computador para escrever programas. Finalmente, veremos um programa simples escrito em duas linguagens de computador comuns para se ter uma idéia de como tudo fica no conjunto.

# I. TIPOS DE FUNÇÕES EXECUTADAS POR COMPUTADORES

As instruções que vimos até aqui, como a receita do bolinho de aveia, com a mistura de seus ingredientes e as forminhas untadas, servem para uma pessoa com duas mãos que esteja trabalhando na cozinha ou numa oficina. Infelizmente, não têm utilidade nenhuma para um computador. Precisamos entrar no mecanismo interno da máquina para que se entenda por quê. É mais fácil entendermos o programa que se escreve se sabemos o que ele realmente faz. Não se apavorem, contudo, esta será apenas uma pequena viagem de apresentação pelo interior da máquina e ao final dela não haverá nenhuma prova.* Em vez de tomarmos um computador em particular para examinar, veremos as funções encontradas na maioria dos pequenos computadores atualmente à venda.

O computador é feito de dois componentes básicos, a *memória* e o *processador*, juntamente com outras partes menores que ajudam a memória e o processador a trabalhar juntos. A memória é uma grande coleção de comutadores eletrônicos na forma de posições LIGADO e DESLIGADO. O processador contém circuitos muito semelhantes aos de uma calculadora mais sofisticada. Ele usa as informações armazenadas na memória para resolver os problemas, depois comunica os resultados ao mundo exterior — quer dizer, a você! Esta organização é mostrada na Figura 2.1.

Embora um computador digital de uso geral possa ser um equipamento espantoso, suas funções são realmente limitadas. Na verdade, a maior parte do tempo ele está simplesmente deslocando as coisas de sua memória conforme indicação do programa. Para se entender como a informação é usada pelo processador, precisamos ver como ela é armazenada na memória.

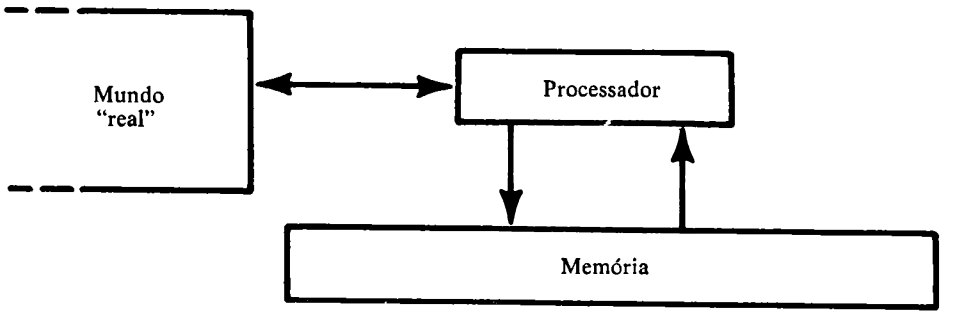
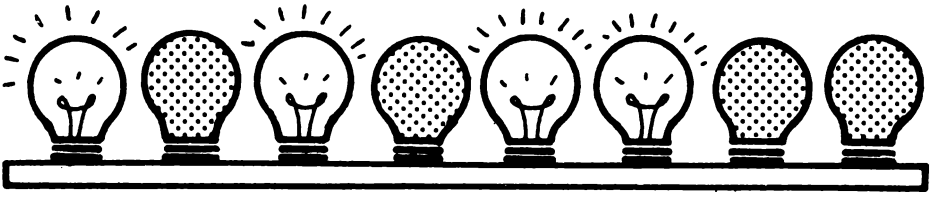



**Figura 2.1** — A Organização do Computador

* Para quem desejar aprender mais sobre computadores em geral, as referências no final do livro oferecem informações mais detalhadas.

## A. Com Que o Computador Trabalha

A informação é guardada na memória em partes individuais chamadas *bytes*. Cada *byte* contém oito pequenos comutadores eletrônicos chamados *bits*. Cada *bit* está ligado (ON) ou desligado (OFF) como uma lâmpada. Assim sendo, cada *byte* pode ser representado desta forma:



O uso de lâmpadas nos primeiros computadores para visualizar a informação levou à imagem popular de lâmpadas piscando num console de computador. Nas máquinas de hoje, a maioria das lâmpadas foi substituída por meios de comunicação mais eficientes, mas muitas pessoas sentem-se decepcionadas com os computadores que não apresentam lâmpadas piscando.

Por convenção, os *bits* que estão "ligados" são representados pelo dígito um (1) e os que estão "desligados" são representados pelo dígito zero (0). Por exemplo, os cinco *bytes* seguintes podem ser usados para representar os números de zero a 4:

| | | | | |
|---|---|---|---|---|
| 00000000 | 00000001 | 00000010 | 00000011 | 00000100 |
| 0 | 1 | 2 | 3 | 4 |

Há 256 combinações possíveis dos oito dígitos 1 e 0 de um *byte*. Cada *byte* pode ser imaginado como um único meio de informação disponível para o computador. Os *bytes* podem ser usados numa variedade de maneiras para registrar a informação que o computador usará. Alguns usos comuns de um só *byte* incluem:

- Uma única instrução para o processador. Muitos computadores são, portanto, limitados a um total de 256 instruções.*
- Um número inteiro de 0 a +255. Reparem que o zero é o 256.º símbolo.
- Um só caracter impresso, incluindo letras maiúsculas e minúsculas do alfabeto, dígitos de "0" a "9" e vários sinais de pontuação.
- Uma série de *bytes* formando uma unidade maior. Por exemplo, dois *bytes* de 8 *bits* juntos contêm 16 *bits*. Isso dá um total de 65536 combi-

* Algumas das máquinas pequenas mais novas usam mais de um símbolo para cada instrução.

nações de 1 e 0. Números de 16 *bits* são geralmente usados em BASIC e Pascal para números inteiros de —32768 a +32767. Os números 256, 32768 e 65536 são vistos com freqüência quando se lida com computadores.

- A localização de outros símbolos na memória. Geralmente essas localizações são feitas por dois *bytes* juntos.

Qualquer localização na memória pode ser usada para armazenar qualquer *byte*. Cada localização da memória tem um *endereço* que vai de zero até a última posição na memória (talvez 65535). Cada *byte* de memória é identificado por seu endereço, assim como as casas de uma rua são identificadas por seus números.

Lembrem-se de que um *byte* pode ser usado de mais de uma maneira. Por exemplo, a maioria dos pequenos computadores usa o mesmo *byte* para representar o número 33, o ponto de exclamação (!) e uma instrução de máquina. O computador deve saber como interpretar cada *byte*. Por exemplo, deve manter os números e caracteres distintamente, sem tentar somar um símbolo que represente uma letra do alfabeto a um que represente um número. Felizmente, a linguagem de programação cuidará de quase tudo isso.

Quando os *bytes* são usados como informação por um programa, são chamados de *dados*. A principal função do computador é, portanto, o *processamento de dados*.

**B. Como os Computadores Processam a Informação**

Da mesma forma que uma calculadora só pode executar as funções para as quais existe uma tecla, o computador só pode usar a informação da memória de acordo com as possibilidades dos circuitos do processador. Cada operação que o computador pode executar é chamada *instrução de máquina*. Todas essas instruções juntas formam o *conjunto de instruções* do computador. O conjunto de instruções inclui geralmente os seguintes tipos de operação:

- Move um *byte* de uma posição da memória para outra.
- Soma ou subtrai dois *bytes*.
- Compara dois *bytes* para decidir se o primeiro é menor, igual ou maior que o segundo.
- Começa a executar as instruções em determinada posição, possivelmente com base nos resultados de uma comparação anterior.
- Começa a execução de uma sub-rotina de instruções em determinada posição ou retorna de uma sub-rotina.
- Dá entrada de *byte* num dispositivo, como um teclado, e saída de *byte* para equipamentos.

Instruções adicionais são geralmente obtidas para operações especiais que estão além do objetivo de nossa análise. Quanto mais instruções tiver o compu-

tador, mais potente ele é. E mesmo o limitado conjunto de instruções de um pequeno computador é suficiente para grandes tarefas programáveis.

A verdadeira força de um computador advém da possibilidade de que suas instruções sejam modificadas à medida que o programa avança. Ele é muito mais flexível do que outros tipos de equipamentos de cálculo, como jogos eletrônicos, em que os programas estão permanentemente fixos nos circuitos das máquinas. Do mesmo modo como as pessoas que montaram o brinquedo articulado do capítulo anterior leram e seguiram as instruções, assim também o processador lê uma instrução após outra e desvia, faz loops ou chama sub-rotinas, dependendo das condições do momento. Cada instrução de máquina inclui os endereços de todos os dados com os quais ela vai lidar, assim como as instruções do brinquedo apresentavam os nomes das peças.

## II. JUNTANDO DADOS E INSTRUÇÕES

O trabalho de um programa é manipular os dados armazenados na memória através da utilização das instruções disponíveis. Contudo, normalmente não se especificam dados, numa linguagem de programação, nas formas de 0 e 1. Por isso, nossa próxima tarefa é examinar como os dados são incluídos ou identificados numa linguagem de programação.

A fim de ilustrar exemplos de tipos de dados que podem ser manipulados por um programa, voltaremos mais uma vez ao brinquedo articulado montado no primeiro capítulo. Uma das suas mais surpreendentes funções é embaralhar e distribuir cartas. Na verdade, junto com o brinquedo vem um baralho especial usado no jogo de pescaria.

Como num baralho comum, o canto superior esquerdo da carta tem o seu valor, começando pelo Ás e terminando por Valete, Dama e Rei. Abaixo do valor vem o naipe da carta: Espadas, Copas, Ouros e Paus. Essas duas indicações aparecem nos dois extremos da carta, ou seja, ela pode ser identificada mesmo que esteja de cabeça para baixo. A cor das cartas é a mesma, como num baralho convencional. Contudo, no centro de cada carta há o desenho de tipos diferentes de peixes em lugar dos naipes ou desenhos. No verso de cada carta há o desenho de um monstro marinho de um antigo mapa de navegação e o nome do fabricante. Os dados do fabricante e a frente e o verso de um Sete de Espadas aparecem na Figura 2.2.

O baralho tem as 52 cartas normalmente usadas. Não há curingas, mas uma "carta" extra que traz as regras do jogo de pescaria.

Na explicação a seguir, usaremos a informação dessas cartas para ilustrar os tipos de dados que podem ser usados num programa de computador e como esses dados podem ser organizados.

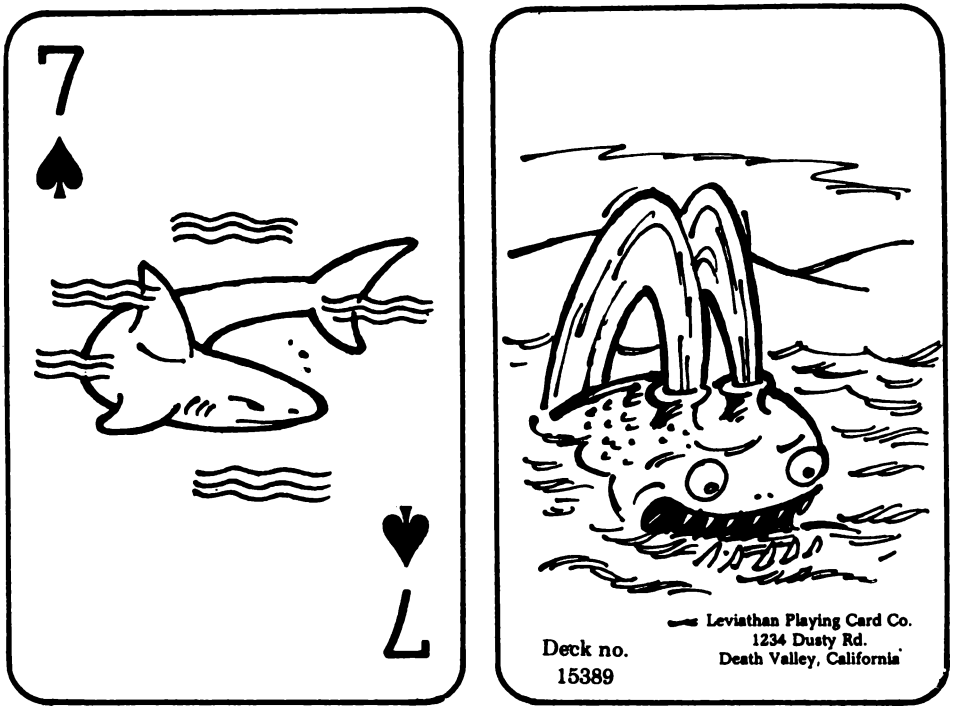



**Figura 2.2** — Uma Carta Usada no Jogo Pescaria

## A. Números

O tipo mais comum de dados é simplesmente um número. Números, ou *dados numéricos*, são largamente usados em programas sobre coisas que podem ser contadas, como cruzados e centavos. E quase todos os programas utilizam números para contar os loops e controlar as informações estruturadas que desenvolvemos no primeiro capítulo. Os valores numéricos podem ser manipulados pelas regras da aritmética como adição e divisão. Os números também podem ser comparados para se ver qual deles, entre dois, é o maior.

Os números podem ser tanto valores exatos, como o número de cartas de um baralho, quanto valores aproximados, como 1/3 (aproximadamente 0,33333). Usar a aritmética em números aproximados pode levar a respostas peculiares. Por exemplo: como 0,33333 é um pouco menor que 1/3, pode-se descobrir que ao se dividir 1 por 3 e depois multiplicar o resultado por 3 — o que daria 1 (1,00000) — o resultado será um inesperado 0,99999.

O baralho usado em nosso jogo de pescaria é identificado nas costas com o número 15389. O valor de cada carta pode ser representado pelos números

de um a treze, sendo que o Valete (J), a Dama (Q) e o Rei (K) são, respectivamente, onze, doze e treze. Da mesma forma, os naipes podem ser representados por números de um a quatro. O número de cartas pode ser então calculado multiplicando-se o número de cartas de um naipe pelo número de naipes:

BARALHO = N.º DE CARTAS DE UM NAIPE × NÚMERO DE NAIPES

ou

$$52 = 13 \times 4$$

## B. Caracteres e Strings

O segundo tipo mais comum de dados são os *dados em forma de caracteres*, que são as letras do alfabeto maiúsculas e minúsculas, os algarismos de "0" a "9" e diversos sinais de pontuação. Como este tipo de dados inclui tanto letras quanto números, ele é chamado geralmente de *alfanumérico*. Um elemento de dados formado por diversos caracteres agrupados chama-se *string de caracteres* ou simplesmente *string*. Os strings são usados para representar os nomes das coisas, para visualizar informações em tela de terminal, ou para imprimir cabeçalhos e títulos em relatórios. Todo o texto deste livro é uma série de dados em forma de caracteres. Os espaços entre as palavras não são simplesmente lugares vazios mas caracteres de *espaço em branco*.

Observem que o "1" neste caso não é o valor do número um, mas sim o caracter "1" que pode ser impresso numa série de textos como "1234 Dusty Road". Para o computador, a diferença entre o valor um e o caracter "1" é uma distinção importante e criará problemas se não for bem compreendida. Por isso este livro irá sempre se referir ao *número* um quando se tratar do valor um; e o *caracter* "1", entre aspas, quando se tratar de *caracter* a ser impresso.

Ao contrário dos dados numéricos, os strings de caracteres não são nunca valores aproximados e não são processados aritmeticamente. Os strings podem ser movidos de um lugar para outro, agrupados em strings maiores, divididos em strings menores, ou impressos como saída. As séries também podem ser comparadas entre si e classificadas em alguma ordem específica (por exemplo, ordem alfabética).

O nome do fabricante e seu endereço no verso das nossas cartas são representados como três strings de caracteres impressos. Os nomes dos naipes podem ser representados pelos strings: "ESPADAS", "COPAS", "OUROS" e "PAUS".

Há muitos outros tipos de dados que um programa pode usar. Por exemplo, a cor de cada carta da pescaria é um item especial que tem um valor entre dois: vermelho ou preto. O desenho do peixe em cada carta é um item de *informação gráfica* e a representação de uma ilustração no computador chama-se *gráficos em computador*. O objeto da aplicação gráfica do computador é uma área complexa e de rápido desenvolvimento das aplicações de programação. Quem

tiver um computador com caracteres gráficos poderá ver os símbolos dos quatro naipes na tela do terminal.

Na maioria dos casos, contudo, os dados serão representados por números ou strings de caracteres. Por exemplo, as cores vermelho e preto poderiam ser representadas pelos números um e dois ou pelos caracteres "V" e "P". Os nomes dos peixes dos desenhos poderiam ser representados por 52 strings de texto descritivo.

Um tipo especial de dados que usaremos de vez em quando chama-se *flag* ou *sinalizador*. Um sinalizador tem apenas dois valores possíveis: *verdadeiro* e *falso*. Pode ser representado pelos números zero e um (como um *bit*), ou por valores de caracteres "V" e "F". Na linguagem BASIC é geralmente representado como uma variável numérica usando o zero para falso e "menos um" para verdadeiro. Em Pascal, o sinalizador é um tipo especial de dado chamado variável BOOLEANA.*

No restante deste livro vamos lidar quase que totalmente com dados numéricos e strings de caracteres, com o "flag" usado de vez em quando para controlar o processamento.

## III. COMO OS TIPOS DE DADOS SÃO USADOS

Os dados de um programa também podem ser classificados como *constantes* ou *variáveis*. O valor de uma constante nunca muda, como o nome e endereço do fabricante no verso das cartas que mostramos. Em contraste, uma variável pode assumir diversos, até centenas de valores, durante a execução de um mesmo programa.

### A. Constantes

As constantes podem ser numéricas ou strings. As constantes numéricas são normalmente escritas com numerais:

```
5
—1,50
45,20
```

Isso porque 5 é sempre 5, e nunca —1,5 ou qualquer outro valor. Nas instruções de montagem do brinquedo do Capítulo Um, há um loop que controla a colocação das rodas nos eixos e que deve ser executado *exatamente* cinco vezes. Reparem que 5 é considerado o *número* 5, não um caracter.

Constantes de strings de caracteres são geralmente escritas entre aspas ou apóstrofo. Por exemplo:

---

* Seu nome vem da Álgebra de Boole, o ramo da matemática que trata de valores verdadeiros e falsos.

"Leviathan Playing Card Company"
"1234 Dusty Road"
"Deseja outra carta?"

Observem que os caracteres "1234" entre aspas são considerados como uma cadeia de caracteres, assim como 'Road", e não como dados numéricos.

**B. Variáveis**

As variáveis são elementos de dados cujos valores devem poder mudar enquanto o programa se processa. O modo como as variáveis são escritas varia de uma linguagem para outra. Neste livro, os nomes variáveis são geralmente escritos numa só palavra. Lembrem-se de que quando se escreve em código de computador para um programa pode-se ter que usar nomes diferentes (especialmente em BASIC). Cada variável do programa deve ter um único nome para evitar confusão. Ajuda muito se o nome indicar como a variável será usada. Por exemplo, o valor de uma carta pode ter o nome de variável VALOR. Reparem que o nome não indica que valor a carta realmente terá. Neste caso, poderia ser qualquer valor de um a treze. Na verdade, pode ter todos esses valores antes que o programa chegue ao fim.

Tanto as variáveis numéricas como strings recebem nomes de variáveis. Em algumas linguagens o nome indica o tipo de variável. Na maioria das versões de BASIC, por exemplo, os nomes de variáveis strings terminam com o sinal de cifrão. A1 é uma variável numérica, A1$ é uma variável string. (Reparem que elas seriam diferentes das constantes de string "A1" e "A1$".) Os nomes variáveis geralmente têm que começar por uma letra para distingui-los das constantes numéricas.

Para nosso baralho de pescaria, os seguintes nomes variáveis podem ser usados:

| | |
|---|---|
| VALOR | O valor no alto da carta (numérico) |
| NAIPE | O naipe da carta (string) |
| COR | A cor da carta (string) |
| PEIXE | O peixe desenhado na carta (string) |

Há limites estritos dos valores que podem ser atribuídos a cada variável dada. Não se pode atribuir valor numérico a variáveis string e vice-versa. Variáveis numéricas geralmente têm limites inferior e superior para os valores que lhes podem ser atribuídos. Por exemplo, Pascal e muitas versões de BASIC contêm *números inteiros* que só podem ser um número de 32768 a +32767 sem casa decimal. Números como 0,33333 são geralmente chamados de números *reais*. Podem tanto ser números de precisão simples, com sete ou oito dígitos significativos, ou números de dupla precisão, que têm até 17 dígitos significativos. As variáveis de string geralmente têm um limite sobre o número de caracteres que podem ser

atribuídos a elas. Dentro desses limites, contudo, as variáveis podem ter qualquer valor legal que se deseje. Uma variável pode ter um valor constante ou o valor corrente de qualquer outra variável do mesmo tipo. Por exemplo, para se determinar a COR adequada de uma carta uma vez que o NAIPE já seja conhecido, a instrução IF-THEN-ELSE a seguir poderia ser usada:

Se (IF) O NAIPE FOR IGUAL A "ESPADAS" OU (OR) SE (IF) O NAIPE
FOR IGUAL A "PAUS", ENTÃO (THEN) A COR É IGUAL A "PRETO"
CASO CONTRÁRIO (ELSE)
A COR É IGUAL A "VERMELHO"

### C. Tipos Primários de Dados

Até aqui vimos os tipos primários de informações disponíveis, números e strings de caracteres, e seu uso como constantes e variáveis. Tudo isso pode ser resumido na Figura 2.3.

É importante compreender a Figura 2.3 antes de começar a escrever programas de computador. Quem tiver alguma dúvida sobre os tipos de dados, constantes e variáveis, deve rever esta parte antes de prosseguir.

## IV. ORGANIZANDO OS DADOS

### A. Estrutura dos Dados

Existe uma técnica final para organização dos dados que precisamos ver. Tem o objetivo de reunir diversos itens de informação que tenham relação e tratar o grupo como um novo tipo de dados.

Este novo tipo de dados chama-se *estrututura de dados.* Observem que uma estrutura de dados é feita sempre com um grupo de tipos de dados já existentes. A estrutura de dados terá um nome próprio que lhe será atribuído do mesmo modo como outros tipos de dados. Como já devem ter imaginado, nosso baralho de cartas é feito de estruturas de dados.

| USOS DOS DADOS / TIPOS DE DADOS | CONSTANTES | VARIÁVEIS |
|---|---|---|
| NUMÉRICO | 5<br>−1,50<br>0,33333 | VALOR<br>A1 |
| CARACTERE | "VALOR"<br>"−1,50"<br>"CASADO" | NOME<br>A1$ |

**Figura 2.3** — Tipos e Usos Primários dos Dados

Cada elemento numa estrutura de dados é chamado *registro*. De um modo geral, um registro pode conter qualquer número de itens de dados. Uma vez estabelecido o formato do registro, todos os demais registros do mesmo tipo terão o mesmo formato. Os itens necessários para um registro de nossas cartas são VALOR, NAIPE e PEIXE. (A COR poderá ser deduzida do NAIPE, por isso não é preciso indicação dela para cada carta.) Portanto, as 52 CARTAS terão o seguinte formato:

```
VALOR  (numérico    —  de um a treze)
NAIPE  (caractere   —  "ESPADAS" será preto
                       "COPAS" será vermelho
                       "OUROS" será vermelho
                       "PAUS" será preto)
PEIXE  (numérico    —  de um a cinqüenta e dois)
```

Quando todos os registros são agrupados, formam uma estrutura de dados mais complexa que se chama *arquivo*. No nosso caso vamos chamar o arquivo de *baralho*. Geralmente todos os registros de um arquivo são do mesmo tipo, mas isso não é obrigatório. Pode haver registros especiais que dêem informação sobre o próprio arquivo, como a "carta" que contém as instruções para se jogar Pescaria.

**B. Matrizes**

Uma estrutura de dados que é feita de repetição de um único tipo de dados chama-se *matriz*. Os itens de uma matriz são geralmente indicados pelo nome da matriz, seguidos pela posição do item na matriz escrito entre parênteses. O número entre parênteses chama-se *índice*. Esta é uma estrutura de dados muito conveniente. Por exemplo, podemos juntar nossos 52 peixes numa matriz. Então, em lugar de precisar de 52 nomes variáveis, de PEIXE1 a PEIXE52, temos uma única matriz PEIXE que contém de PEIXE(1) a PEIXE(52). O sétimo peixe seria PEIXE(7). Chama-se a isso de *matriz unidimensional* porque apenas um índice é necessário para identificar cada item da matriz:

```
PEIXE (índice)
        ↑_____  O índice indica qual peixe se está usando
                e terá um valor de 0 a 52.
```

Algumas linguagens de computador (BASIC, por exemplo) não são favoráveis para trabalhar com registros. Se este for o caso, podemos querer representar nosso baralho de cartas como três matrizes: uma para os valores, outra para os naipes e uma terceira para os peixes. Cada matriz conteria 52 entradas, uma para cada carta. A sétima carta do baralho seria composta do VALOR(7), NAIPE(7) e

PEIXE(7). Todas essas matrizes são matrizes unidimensionais porque só requerem um índice.

VALOR(índice)
NAIPE (índice)
PEIXE (índice)
↑
└— O mesmo índice indica que carta se está usando.

Toda essa informação pode ser combinada numa estrutura de dados usando-se uma matriz com dois índices: um índice para o número da carta e outro índice para indicar qual desses três itens de dados está sendo usado para cada carta. Chama-se a isso de *matriz bidimensional.* Já que uma matriz não pode conter informação numérica e de string, teremos que usar o mesmo tipo de dados para todas as informações. O NAIPE deverá, portanto, tornar-se uma variável numérica, com valores de um a quatro. Nesta matriz bidimensional, o primeiro índice indicará a carta do baralho, como antes; terá um valor de um a cinqüenta e dois. O segundo índice indicará que item sobre a carta está sendo usado, como a seguir: um para VALOR, dois para NAIPE e três para o PEIXE.

Portanto, as três matrizes VALOR, NAIPE e PEIXE serão substituídas pela nova matriz bidimensional BARALHO. O valor da primeira carta do baralho será BARALHO(1,1). O naipe desta carta será BARALHO(1,2). O peixe será BARALHO(1,3). O valor, naipe e peixe da sétima carta do baralho, será BARALHO(7,1), BARALHO(7,2) e BARALHO(7,3). O formato da matriz BARALHO é o seguinte:

BARALHO (índice-da-carta, índice-do-item)

1 = valor da carta
2 = naipe da carta
3 = peixe da carta
1-52 = carta sendo usada

O desenvolvimento de estruturas de dados mais complexas está fora dos objetivos deste livro. São possíveis matrizes com muitas dimensões, registros que contenham matrizes, matrizes de registros, e assim por diante. De um modo geral, o único limite à organização de dados é a imaginação de quem os estiver organizando.

Um último uso de matriz será mencionado aqui porque é geralmente usado para economizar espaço e simplificar grandes programas. Deu para perceber que na matriz bidimensional desenvolvida acima perdemos a habilidade de imprimir o naipe da carta como um string de caracteres. O naipe passou a ser agora um número de um a quatro. Se ainda quisermos os nomes desses naipes acessíveis podemos colocá-los numa pequena matriz de strings chamada NAIPE. NAIPE será uma matriz de strings unidimensional com quatro entradas: "ESPA-

DAS", "COPAS", "OUROS" e "PAUS". Uma matriz com valores comuns como esses é geralmente chamada de *tabela.* O nome do naipe pode ser consultado na tabela da mesma forma como fazemos para ver o horário de saída de um trem em uma tabela ou para encontrarmos um valor matemático numa tabela de números.

O NAIPE pode ser usado com o BARALHO para determinar o naipe de uma carta, como a seguir: procure primeiro o naipe de uma carta no BARALHO, e depois procure o nome do naipe em NAIPE. Por exemplo, vamos chamar o nome do naipe de NOME-DO-NAIPE. Para imprimir o nome do naipe da sétima carta, encontre o número do naipe assim:

```
NÚMERO = BARALHO(7,2)
```

Será um número de um a quatro. O nome desse naipe será:

```
NOME-DO-NAIPE = NAIPE(NÚMERO)
```

Este será a entrada adequada na tabela e poderá ser impresso no vídeo. Este processo de duas operações pode ser abreviado usando-se a seguinte expressão:

```
NOME-DO-NAIPE = NAIPE(BARALHO(7,2))
```

Isso pode ser ampliado para uma forma mais geral substituindo-se a constante 7 pela variável CARTA. Agora o nome do naipe da carta representada pela variável CARTA será:

```
NOME-DO-NAIPE = NAIPE(BARALHO(CARTA,2))
```

Uma vantagem de tudo isso é que em vez de repetir as quatro séries de caracteres dos naipes treze vezes, como fazemos quando tínhamos três matrizes com cinqüenta e dois valores em cada uma, precisamos apenas defini-las uma vez na pequena tabela. Quando se lida com séries de caracteres muito longas isso economiza muito espaço. A segunda vantagem desta organização é que em lugar de ter muitos nomes variáveis para as diferentes cartas e naipes do baralho, há apenas duas: BARALHO e NAIPE. Qualquer carta pode ser identificada, atribuindo-se os valores adequados aos índices dessas matrizes.

Outra tabela poderia ser criada para os nomes dos valores das cartas, incluindo "AS", "VALETE", "DAMA" e "REI". Na prática, os programas geralmente têm muitas tabelas contendo valores especiais que podem ser consultados quando necessário.

Devemos agora mudar o enfoque de como os dados são organizados para quando serão usados no programa. Aquelas pessoas que ainda não entenderam completamente todas as idéias sobre estruturas de dados já apresentadas não

precisam se preocupar. Voltaremos a esse assunto durante os exemplos desenvolvidos no livro. Por enquanto é suficiente entender a importância de organizar os dados de um programa de uma maneira adequada.

## V. COMO OS DADOS SÃO USADOS POR UM PROGRAMA DE COMPUTADOR

Há quatro maneiras de um programa de computador usar os dados à sua disposição: (1) alocar espaço para os dados na memória; (2) ter qualquer entrada necessária do mundo externo (o usuário); (3) realizar os cálculos necessários sobre os dados e (4) tornar os dados acessíveis ao mundo externo (saída). A alocação de espaço para os dados pode ser feita automaticamente ou não pelo computador, dependendo do sistema utilizado. As outras três operações, contudo, são as funções básicas da maioria dos programas de computador. O fluxo de dados através de um programa pode ser mostrado no *diagrama de fluxo de dados*. Aqui temos um diagrama simples mostrando o fluxo primário de dados pelo programa:


### A. Alocação de Dados

A primeira coisa que um programa tem que fazer é certificar-se de que sabe que dados usará e onde os encontrará. A primeira parte da receita do bolinho de aveia é uma lista de todos os ingredientes que serão necessários. Isso permite que a cozinheira providencie tudo antes de começar. Da mesma maneira, um programa de computador deve definir cada item da informação e onde ele se encontra na memória. A porção de memória necessária dependerá do tipo de dados usado. Por exemplo, ocupará mais espaço armazenar um string de 25 caracteres do que armazenar um único número. O programa deve, por isso mesmo, ter noção dos dados, de sua localização na memória, e do seu comprimento.

As constantes são definidas como parte do programa pelo usuário e armazenadas na memória quando o programa for colocado na memória. Contudo, os conteúdos das variáveis não se tornarão significativos até que o programa comece a ser processado e elas recebam algum valor específico.

### B. A Entrada de Dados

É possível criar programas que não requeiram qualquer informação do mundo externo. A receita do bolinho de aveia é um programa assim. Toda vez que é executado, o resultado é o mesmo. A única forma de alterar a saída é alterar o programa. Muitos programas, contudo, são escritos para produzir resul-

tados apropriados baseados na mudança de informação, de forma a que não seja preciso reescrever o programa toda vez que o problema mudar. Isso significa que alguma informação precisa ser introduzida ao programa, pelo usuário, quando ele for executado. É claro que o programa tem que ser escrito para solicitar esta entrada.

Muitos programas recebem informação de um terminal de computador, unidade de fita magnética, ou qualquer outro equipamento. A forma mais comum de dispositivo de entrada dos pequenos computadores de hoje é um teclado, gravador cassete, ou drive de disco flexível. Cada parte dos dados deve ser devidamente identificada pelo programa e armazenada em seu lugar adequado na memória.

Alguns programas, como cálculos de pagamento de empréstimos, podem necessitar de apenas uma pequena quantidade de dados no início. Outros programas, como jogos de computador, podem necessitar de informação constante sobre o que o jogador deseja fazer a seguir. É geralmente muito importante que o programa verifique todas as entradas para impedir erros. Quaisquer condições em que um valor não válido possa entrar devem ser checadas e transmitidas por mensagens indicadoras de erro e o usuário solicitado a fazer entrar o valor correto. Já que o computador faz exatamente o que o programa lhe diz para fazer com os dados, uma expressão comum entre programadores experientes é "todo lixo que entra, tem que sair", por isso é muito importante verificar a entrada.

### C. Processando os Dados

Geralmente, o programa de um computador está centrado na realização das operações necessárias com os dados que lhe foram fornecidos. Os números são calculados, as variáveis mudadas, os valores comparados e um resultado final é obtido. Pode ser uma operação única, como no caso da receita do bolinho de aveia, ou pode ser uma operação que necessite ser repetida várias vezes pela mudança de condições, como num jogo de computador.

### D. Emitindo os Resultados

Um programa não teria sentido algum se guardasse para si os resultados de todo este trabalho. É necessário, portanto, gravar ou mostrar os resultados do programa de alguma forma. Isso é feito geralmente num terminal de vídeo, impressora ou em qualquer outro dispositivo de visualização. Em alguns casos, os resultados de uma operação do computador controlarão diretamente outras máquinas, como robôs das linhas de montagem. A forma mais comum de equipamento de saída nos pequenos computadores da atualidade é uma tela de vídeo ou TV. Muitos sistemas também fornecem cópias permanentes numa impressora, gravador cassete ou disco flexível. Da mesma forma que na entrada, a operação de saída pode acontecer apenas uma vez ou ser contínua.

Agora que já se sabe o que um computador pode fazer, que tipos de dados pode usar e como os dados são usados, é hora de se ver como conversar com o computador.

## VI. COMO AS INSTRUÇÕES DEVEM SER ESCRITAS PARA QUE O HOMEM E A MÁQUINA POSSAM ENTENDÊ-LAS

Até aqui os programas apresentados foram escritos no idioma em que estamos habituados a falar, o que parece correto, porque as pessoas que lerem o livro poderão entendê-lo. Como já vimos, contudo, os computadores não são controlados por nossa linguagem, mas por um conjunto limitado e especial de instruções. Se concordamos que o computador só pode "entender" essas poucas instruções, então podemos apresentar um significativo problema de programação: o programador e o computador não entendem a mesma linguagem e portanto não entendem um ao outro.

Na verdade, o problema é similar quando se encontram duas pessoas que não falam o mesmo idioma. As alternativas são: uma pessoa aprende a linguagem da outra, as duas aprendem uma terceira linguagem ou é feita a tradução das duas linguagens através de um intérprete. Como veremos, muitas linguagens de computador foram desenvolvidas para ajudar a melhorar essas soluções.

### A. O que se Entende por "Linguagem de Computador"

As línguas faladas, ou linguagem "natural", desenvolveram-se em função da necessidade de as pessoas transmitirem sentimentos, idéias, medos, humor e uma infinidade de coisas maravilhosas e sutis umas às outras. Essas linguagens são, contudo, ricas em descrições, cheias de contradições e inacreditavelmente imensas e difíceis de serem aprendidas. (Felizmente aprendemos nosso idioma quando ainda somos muito pequenos, para vermos como isso é difícil!) Em contraste, o único objetivo de uma linguagem de computador é expressar um conjunto de instruções o mais precisamente possível. Tal linguagem requer clareza e estrutura formal, mas não necessita de atributos interessantes como descrição, emoção ou humor. Nesta linguagem não é cabível uma instrução correta, mas contraditória. Uma linguagem de programação de computador, portanto, troca as capacidades de uma linguagem natural pela clareza e precisão.

Embora haja centenas de linguagens de computador em uso atualmente, elas podem ser agrupadas nas três categorias seguintes: linguagem de máquina, linguagens montadoras (assemblers) e linguagens de alto nível. Uma linguagem de máquina é usada em cada novo tipo de computador construído. Consiste de todas as instruções diretamente compreendidas pelo computador. Linguagens montadoras estão intimamente relacionadas com as linguagens de máquina, mas traduzem a maioria da representação numérica da linguagem de máquina para um código alfanumérico que o programador pode entender com mais facilidade.

As linguagens de alto nível não estão intimamente ligadas a nenhum computador. Elas traduzem as idéias do programador para qualquer programa de linguagem de máquina que se queira.

**B. Linguagem de Máquina**

As instruções numéricas que compõem o conjunto de instrução de um microprocessador* formam também a linguagem de máquina. São essas as únicas instruções que o computador pode realmente usar, de forma que todos os programas devem ser, mais cedo ou mais tarde, traduzidos para uma série de instruções em linguagem de máquina.

Os exemplos seguintes mostram programas em linguagem de máquina verdadeira que simplesmente adicionam os números um e dois e armazenam o resultado na primeira posição da memória (posição zero). O processo pode ser expresso como "calcular um mais dois". Esses programas só calculam a adição; para imprimirem a resposta seriam necessárias outras providências. O primeiro programa é para o microprocessador Intel 8080 e o segundo para o microprocessador Motorola 6800.

```
8080:   62   1   198   2   50   0   0
6800:   131  1   139   2   151  0
```

**Listagem 2.1** — Dois programas em Linguagem de Máquina para Somar Um mais Dois

O primeiro programa seria armazenado na memória como o seguinte conjunto de *bits* 1 e 0 em 7 *bytes* da memória do computador:

```
00111110
00000001
11000110
00000010
00110010
00000000
00000000
```

Isso é muito bom para as chaves eletrônicas do computador, mas é quase impossível de ser usado por uma pessoa. Outra desvantagem do programa em linguagem de máquina é que as instruções para computadores diferentes serão também diferentes. Como este exemplo mostra, mesmo um programa simples pode variar muito em conteúdo e em tamanho. Programas escritos para um computador não podem ser facilmente alterados para outro, se estão em linguagem de máquina.

---

* O microprocessador é o "cérebro" de um microcomputador, e é responsável por controlar todas as instruções, executando-as com o auxílio de outros dispositivos especiais de entrada e saída. (N. da R.T.)

## C. Linguagem Assembly

É difícil aprender a linguagem do computador. Mas também é impossível para o computador aprender uma linguagem complexa como a nossa, ou o computador e nós aprendermos uma terceira linguagem. A melhor coisa a fazer é traduzir de uma linguagem para outra.

Esta tradução pode ser feita pelo próprio computador, usando um programa especial que lê as instruções na forma que seja mais conveniente para o programador e depois as converte em programas em linguagem de máquina. Os primeiros programas de tradução permitiam o uso de palavras e expressões em código, em vez de números para escrever as instruções. As instruções eram, na verdade, as mesmas, mas numa forma em que as pessoas pudessem entendê-las. Esses programas eram chamados de *assemblers* (montadores) porque montavam instruções mais próximas da nossa linguagem nos programas em linguagem de máquina. Tais programas ainda são utilizados atualmente.

O assembler preocupa-se muito com os detalhes necessários para escrever um programa. Cada instrução de máquina é representada por uma pequena abreviação que é mais fácil de se lembrar do que seu valor numérico. As instruções são escritas uma em cada linha, verticalmente, com comentários (que são desprezados durante a tradução) para tornar o programa mais compreensível. Qualquer posição da memória, em que os dados ou parte de um programa ficarão, deverá receber um nome que o programador possa entender. O assembler traduz então o programa escrito nessa "linguagem" para a verdadeira linguagem de máquina, com suas instruções e endereços.

Para escrever nosso exemplo de programa de adição em linguagem assembler, teremos que acrescentar uma fase para definir a variável SOMA. O programa da Listagem 2.2 foi escrito para o computador Intel 8080. Ele criará o programa de linguagem de máquina mostrado na Listagem 2.1. Um valor de um será primeiro deslocado para uma parte do processador chamado A, que executa a aritmética. Um valor de dois será adicionado a ele. O resultado ainda em A, será então armazenado em SOMA. Embora isso ainda possa parecer estranho, há diversos avanços aqui, no tocante ao programa em linguagem de máquina. Cada instrução tem sua própria linha. Os comentários, que começam com um ponto-e-vírgula, fazem uma descrição do que está acontecendo. Uma instrução como STA SOMA é mais fácil de se entender do que uma instrução de máquina 50 0 0. E os nomes dos dados e endereços no programa são mais fáceis de serem identificados e usados.

Mesmo assim, ainda estamos muito longe do "some um mais dois". Cada linha de instrução do programa em linguagem assembly cria uma instrução de máquina completa, de modo que ainda é preciso saber como o processador trabalha. Os programas ainda não poderão ser transferidos com facilidade de um computador para outro. É claro que precisaremos de um programa de tradução melhor para tornar o computador mais fácil de se usar.

```
; Este e o programa em Linguagem Assembler 8080 para somar 1 + 2
; Autor: Jack Emmerichs                          Ultima alteraçao: 01/01/82
        ORG  0          ; Defina em zero a origem das variaveis
Soma:   DS   1          ; Soma, defina a armazenagem de um BYTE
        ORG  1          ; Defina em um a origem do programa
Inicio:                 ; Inicio do programa
        MVI  A,1        ; Coloque 1 no local A
        ADI  2          ; Adicione, a isso, dois
        STA  SOMA       ; Armazene a resposta em soma
        END             ; Fim do programa
```

**Listagem 2.2** — Programa em Linguagem Assembler para Adicionar Um mais Dois

## D. Linguagens de Alto Nível

As linguagens de alto nível são programas de tradução que permitem que se escrevam instruções de forma mais próxima da linguagem natural. Como se pode esperar, esses programas são muito mais complexos que os assemblers. Geralmente, muitas instruções de máquina são necessárias para se realizar uma só função. O programador não precisa mais conhecer o funcionamento interno da máquina, já que o programa de tradução aloca espaço para os dados e cria as instruções de máquina adequadas.

É importante entender que não há nenhum tipo de mágica na forma da linguagem em si. A sintaxe usada é meramente um modo formal de descrever entradas significativas para o programa de tradução. É o tradutor que transforma as instruções do programador no programa em linguagem de máquina operativo.

A Listagem 2.3 mostra nosso simples programa de adição de um mais dois escrito em quatro linguagens de alto nível. Reparem como são mais fáceis de se entender do que os exemplos anteriores. Aqui não precisamos saber onde os números estão armazenados na memória, embora ainda tenhamos que definir as variáveis que vão contê-los. Nas linguagens que não requerem definição prévia de variáveis, o tradutor alocará espaço para elas à medida que forem encontradas. O programador tem pouco — ou nenhum — controle sobre como isso é feito.

BASIC e Pascal foram escolhidas como exemplos porque são as linguagens mais populares usadas nos pequenos computadores de hoje. Em computadores maiores, o FORTRAN é a linguagem científica mais comum, enquanto o COBOL é a mais comum no mundo dos negócios. Esses exemplos, a propósito, não são programas completos, mas apenas o suficiente para mostrar como é cada linguagem.

Isso é na verdade um avanço. Essas instruções são muito mais fáceis de serem entendidas. Além disso, programas escritos nessas linguagens são geralmente fáceis de serem transportados de uma máquina para outra. As instruções

originais do programador, com algumas pequenas mudanças, talvez, são simplesmente traduzidas novamente para a outra máquina.

```
BASIC:
        REMARK      BASIC usa pequenos nomes variaveis
        REMARK      e nao requer que eles
        REMARK      sejam definidos antes do tempo

        REMARK      Faz os calculos
        F = 1 + 2

PASCAL:
                    (* Pascal requer definicao dos dados
                       e permite nomes variaveis grandes      *)

        VAR         SOMA: INTEGER;

                    (* Faz os calculos *)

        SOMA: = 1 + 2;

FORTRAN:
        C       FORTRAN parece com BASIC mas permite
        C           maiores nomes de variaveis.

        C           Faz os calculos

        SOMA = 1 + 2

COBOL:
        *           COBOL usa definicoes de dados maiores e
        *               frases em ingles.

                    01      SOMA        PICTURE 999.
        *               Faz os calculos
                    ADD 1 TO 2 GIVING SOMA.
```

**Listagem 2.3** — Programas de Alto-Nível para Adicionar Um mais Dois

As vantagens das linguagens de alto nível são tão grandes que virtualmente todas as aplicações são agora escritas usando-se uma delas. O restante deste livro empregará exclusivamente linguagens de alto nível.

Como acontece com a maioria das coisas boas, há problemas com essas linguagens. O maior deles é que, uma vez que as instruções não são limitadas a

---

* Daqui por diante, nas listagens em BASIC usaremos a palavra comentário abreviada em inglês (REM). (N. do T.)

um determinado equipamento, a linguagem pode mudar de vez em quando. Isso permite aumento da linguagem, mas permite também que haja versões conflitantes dela, já que nem todos os fabricantes produzirão essas melhorias do mesmo modo. Ao longo dos anos, muitas linguagens de alto nível viram nascer *dialetos.* É por isso que programas que são publicados para uso geral, como os jogos em BASIC e os exemplos deste livro, podem não funcionar no seu computador.

Outro problema é que o programa de tradução não é tão eficiente quanto um programa escrito em linguagem de máquina. Por isso, programas ou rotinas que devem ser processados muito depressa ou usar pouca memória são até agora geralmente escritos usando linguagem de máquina. Isso é particularmente importante em pequenos computadores em que a velocidade e o tamanho da memória são limitados. Os programas de tradução são, eles próprios, por exemplo, escritos em assembly.

### E. Compiladores e Interpretadores

Há muitas maneiras em que a linguagem de alto nível pode operar. A maioria dos grandes computadores usa programas de tradução que lêem as instruções do programador uma vez e as convertem em um novo programa, a nível de máquina. Este tipo de tradutor chama-se *compilador.* A tradução, que é complexa, é feita apenas uma vez. A maioria dos sistemas Pascal usa compilador.

Outra maneira, geralmente usada em pequenos computadores, é traduzir cada linha do programa de alto nível à medida que ele vai sendo processado. O processo assemelha-se a um turista que usa um guia de viagem para falar uma língua estrangeira. Cada vez que se precisa dizer alguma coisa é necessário olhar no livro de novo. Esses programas são chamados interpretadores. Já que a tradução é feita de forma contínua, os programas que usam interpretadores são processados um pouco mais lentamente. Os interpretadores são mais fáceis de ser escritos, contudo, e podem necessitar de menos memória do que um programa com compilador. A maior parte dos pequenos sistemas BASIC usa interpretadores.

Um meio termo entre essas duas formas é o *semicompilador.* Este é um programa que compila a maior parte do programa de entrada em código de máquina (ou qualquer coisa muito similar). Outro programa menor, chamado *pacote em tempo-de-execução,* é armazenado na memória juntamente com o programa semicompilado. A maioria das partes mais complexas do programa funciona como chamadas de sub-rotina das rotinas-padrão do pacote. Isso nos oferece a simplicidade de um intérprete e a velocidade de um compilador.

## VII. APLICAÇÃO REAL DE PROGRAMAS DE COMPUTADOR

Agora que vimos o que é uma linguagem de computador de alto nível, teremos alguns exemplos reais.

Um exemplo simples que empregará muito do que vimos até aqui é o programa de consulta do valor e naipe de uma carta do baralho de pescaria e a

impressão da informação na tela. A entrada do teclado será um número de um a cinqüenta e dois indicando a posição da carta num novo baralho que não esteja misturado. O número um representa o ÁS de Espadas e o cinqüenta e dois o Rei de Paus, na outra extremidade. A saída será o valor e o nome do naipe da carta cujo número serviu de entrada. O programa necessitará da tabela de nomes de naipes desenvolvida anteriormente e outra tabela com os valores. As instruções aparecem na Figura 2.4.

1. Escolha o número de uma carta e certifique-se de que é um número válido compreendido entre zero e 52.
2. Se o número de entrada for zero, encerre o programa.
3. Determine o naipe da carta sabendo que elas são dispostas em ordem seqüencial e que as 13 primeiras são de espadas, as 13 seguintes de copas, depois ouros e finalmente paus.
4. Determine o valor da carta sabendo que, depois de cada grupo de 13 cartas, a seqüência de valores recomeça. Lembre-se que o um é o ÁS e que onze, doze e treze são VALETE, DAMA e REI.
5. Apresente o naipe e valor da carta.
6. Continue consultando os valores das cartas até uma carta com valor zero entrar.

**Figura 2.4** — Etapas de um Problema "Real"

Muito embora tudo pareça muito específico, ficaram algumas coisas sem definição. Como deve ser solicitada a informação de entrada? Como deve ser manipulada uma carta com número inválido? Como usar o que se sabe sobre o baralho para se encontrar o valor e o naipe? Esse grau de liberdade num conjunto de instruções provavelmente funciona bem para uma pessoa, mas não funcionará com o computador. Ninguém desenvolveu ainda a instrução de programa: "Você sabe o que eu quero dizer com isso."

As instruções da Listagem 2.4 são muito mais específicas. Cada instrução foi escrita segundo as instruções desenvolvidas no primeiro capítulo. Reparem que elas parecem aquele tipo de instrução que "qualquer bobo" pode entender. É justamente o que queremos, já que o computador é um tipo de tolo. A Figura 2.5 mostra um diagrama dessas instruções.

A Listagem 2.4 não é um programa verdadeiro de computador. Um programa real numa linguagem específica de computação chama-se *listagem de programa* ou *código de programa*. O exemplo, no entanto, está tão próximo de um programa que se chama *pseudocódigo* ou *projeto de código*. Os programas estruturados são desenvolvidos neste formato porque é um estilo livre, fácil de se trabalhar e sem ser ligado a nenhuma linguagem específica.

```
PROGRAMA PRINCIPAL
        DEFINE AS VARIAVEIS
        CARREGA A TABELA DE NAIPES
        CARREGA A TABELA DE VALORES

        DO LOOP
            DO LOOP
                IMPRIME O PEDIDO DE UM N.° DE CARTA
                OBTEM UM N.° DE CARTA
                IF NAO FOR VALIDO THEN
                    IMPRIME MENSAGEM DE ERRO
            UNTIL O NUMERO DA CARTA SEJA VALIDO

            IF O NUMERO DA CARTA NOT ZERO THEN
                CHAMA A ROTINA-DE-IMPRESSAO

        UNTIL A CARTA SEJA ZERO


FIM DO PROGRAMA PRINCIPAL
STOP PROGRAMA


ROTINA-DE-IMPRESSAO

        CASE O N.° DA CARTA SEJA DE

             1 A 13 : NAIPE = 1
            14 A 26 : NAIPE = 2
            27 A 39 : NAIPE = 3
            40 A 52 : NAIPE = 4

        END CASE
        VALOR = N.° CARTA
        DO WHILE VALOR MAIOR QUE 13
            SUBTRAI 13 DE VALOR


        END DO WHILE
        IMPRIME TABELA-VALOR (VALOR), TABELA-NAIPE (NAIPE)
RETURN
```

**Listagem 2.4** — Pseudocódigo de um Exemplo Real (Jogo de Vinte-e-Um)

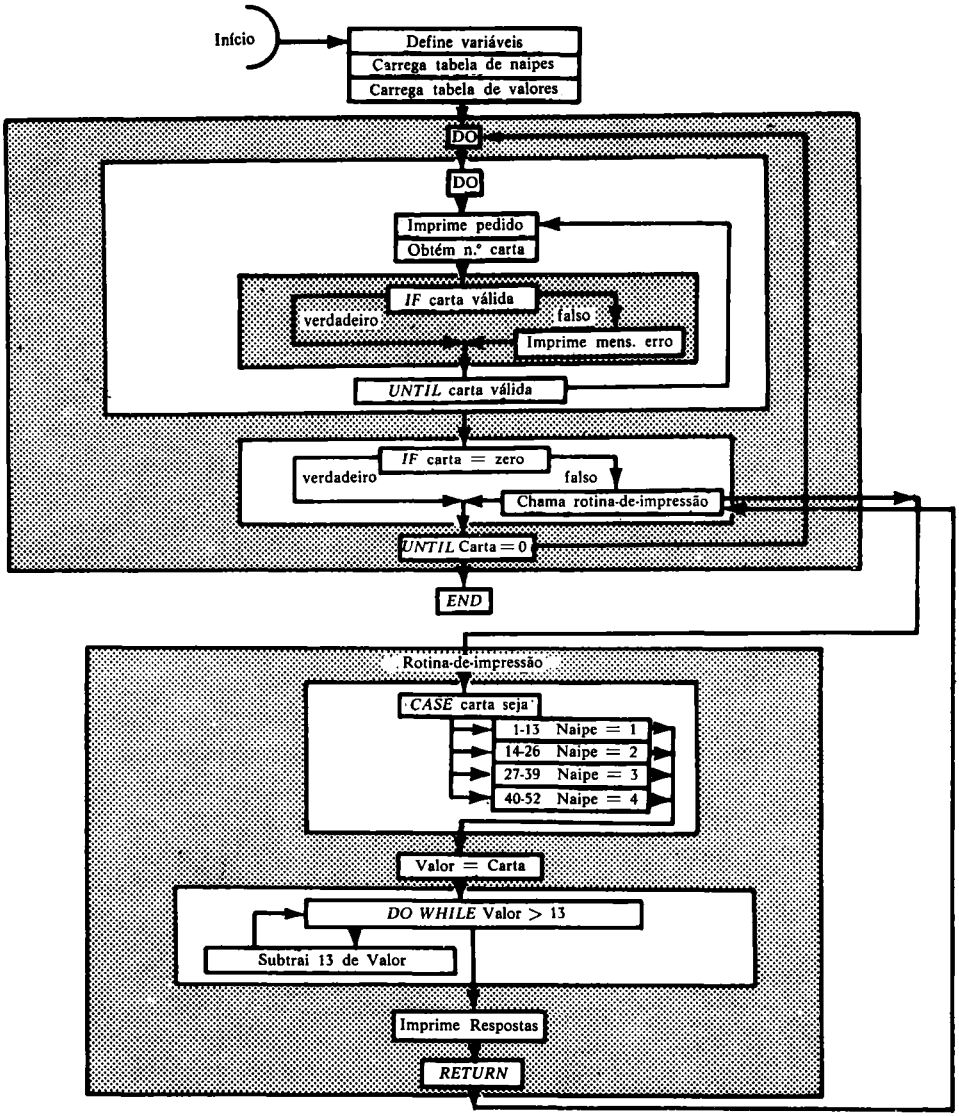



**Figura 2.5** — Estrutura do Problema-Exemplo

Uma vez que o programa seja desenvolvido em projeto de código, é fácil traduzi-lo para o programa final da linguagem que estiver sendo usada. A estrutura apresentada na Figura 2.5 mostra como essas instruções relacionam-se com as desenvolvidas no Capítulo 1. Daqui em diante, usaremos projetos de código em lugar de diagramas de estrutura porque os projetos de código são mais fáceis para se trabalhar e se parecem mais com as listagens do programa final.

Os programas mostrados nas Listagens 2.5 e 2.6 são o exemplo de consulta de cartas escrito finalmente nas linguagens de computador BASIC e Pascal.

Os comentários apresentados nas listagens referem-se a itens de interesse das linguagens específicas.

Os programas apresentados como exemplo neste livro necessitarão de pequenos comentários. Contudo, aqui estão algumas coisas que é preciso que se tenha em mente à medida que avançamos.

Os programas deste livro são escritos em suas partes mais importantes numa ordem específica. No começo há um comentário que identifica o programa, quem o escreveu e quando foi alterado pela última vez. Isso é seguido de definições dos nomes de todas as variáveis que não são autodefiníveis. Os programas em BASIC terão sua parte principal, chamada de *principal,* no início, seguida pelas sub-rotinas que utiliza. O Pascal requer que todas as rotinas sejam definidas antes de serem usadas, de forma que as sub-rotinas serão listadas em primeiro lugar e o programa principal no final.

Em todos os programas, as matrizes são definidas mostrando-se o tipo de dados usado na matriz, quantas dimensões há e de que tamanho podem ser os índices. Em Pascal, isto é feito na seção VAR, na qual todas as variáveis são definidas. Em BASIC, a instrução DIM é usada para definir as matrizes.

Todas as linhas do programa em BASIC são numeradas. Pode-se passar (GOTO) a qualquer instrução ou chamar (GOSUB) uma instrução usando-se o número de sua linha. Toda linha que começar com a palavra-chave REM é um comentário. As dimensões das duas matrizes são definidas na linha 140. São carregadas com valores usando as instruções READ nos loops da linha 180 até a linha 230, e a instrução DATA da linha 150 até a 170. Cada execução de uma instrução READ usa a constante seguinte nas instruções DATA. As instruções estruturadas DO-WHILE, DO-UNTIL e CASE foram construídas a partir de instruções IF-THEN-GOTO simples. A instrução PRINT mostra a saída no vídeo. A instrução INPUT "lê" as variáveis do teclado. Esta é uma versão razoavelmente padrão do BASIC e deve funcionar em qualquer sistema que permita matrizes de strings de caracteres — como o Radio Shack TRS-80.* Os programas em BASIC deste livro são escritos nesta versão.

```
10    REM  Este e o exemplo da impressao da carta escrito em BASIC
20    REM       Autor: Jack Emmerichs
30    REM       Ultima alteracao: 01/01/82
40    REM
50    REM       As variaveis usadas serao:
60    REM         N$      TABELA DE NAIPES     V$    TABELA DE VALORES
70    REM         N       NUMERO DO NAIPE      V     NUMERO DO VALOR
80    REM         C       NUMERO DA CARTA
90    REM         X       CONTADOR GERAL
100   REM
110   REM    NOTA: Alguns BASIC necessitam de uma instrucao CLEAR neste ponto
```

* TRS-80 é marca registrada da divisão Radio Shack da Tandy Corporation.

```
120     REM  DEFINE AS DUAS MATRIZES E AS CARREGA COM OS VALORES
130     REM
140 DIM  N$  (4),  V$  (13)
150 DATA  "ESPADAS", "COPAS", "OUROS", "PAUS"
160 DATA  "AS", "2", "3", "4", "5", "6", "7", "8", "9", "10"
170 DATA  "VALETE", "DAMA", "REI"
180 FOR    X  =  1  TO  4
190     READ  N$  (X)
200 NEXT  X
210 FOR    X  =  1  TO  13
220     READ  V$  (X)
230 NEXT  X
240     REM
250     REM          ENTRA UM N.° DE CARTA E TESTA VALIDADE
260     REM
270 PRINT  "Entre um numero de carta ou zero (fim):";
280 IMPUT     C
290 IF  C  >  = 0 AND C< = 52 AND C = INT(C) THEN GOTO  350
300 PRINT "***** Por favor, entre um numero de carta adequado."
310 GOTO  270
320     REM
330     REM          AGORA PROCESSA A CARTA
340     REM
350 IF  C  <>  0 THEN GOSUB  4
360 IF  C  <>  0 THEN GOTO  270
370 END
380     REM
390     REM          SUB-ROTINA PARA LISTAR OS VALORES DA CARTA
400     REM
410 N  =  1
420 IF  C  >  13  THEN  N=2
430 IF  C  >  26  THEN  N=3
440 IF  C  >  39  THEN  N=4
450 V=C
460 IF  V  <=13  THEN GOTO  490
470 V=V-13
480 GOTO  460
490 PRINT  " A carta e ";V$(V); "de"; N$(N);"."
500 RETURN
```

**Listagem 2.5** — Exemplo do Programa em BASIC

```
(*  Este e o exemplo de impressao de cartas escrito em Pascal
                    Autor: Jack Emmerichs, Ultima alteracao: 01/01/82    *)

PROGRAM  CARTAS (INPUT, OUTPUT);
      VAR
         NAIPE:          ARRAY  [1.. 4]  OF  SPRING  [7];
         VALOR:          ARRAY  [1..13]  OF  SPRING  [6];
         CARTA:          INTEGER;
         NUM-NAIPE:      INTEGER;
         NUM-VALOR:      INTEGER;

   PROCEDURE LISTAGEM;                  (*  Sub-rotina de listagem          *)
      BEGIN
         CASE (CARTA — 1) DIV  13  OF
            0:    NUM — NAIPE  : =  1;
            1:    NUM — NAIPE  : =  2;
            2:    NUM — NAIPE  : =  3;
            3:    NUM — NAIPE  : =  4;
         END;                              (* O  END  refere-se a instrucao CASE *)

         NUM — VALOR  : =  CARTA;
         WHILE NUM — VALOR > 13  DO
            NUM — VALOR  : =  NUM — VALOR — 13;

         WRITELN (OUTPUT, 'A carta e o',
            VALOR[NUM — VALOR], 'de', NAIPE[NUM — NAIPE], '.');
      END;                                 (* Fim da rotina LISTAGEM  *)


BEGIN                                     (* Inicio: carrega valores das tabelas *)
   NAIPE [1] : = 'ESPADAS';               NAIPE [2] : = 'COPAS';
   NAIPE [3] : = 'OUROS';                 NAIPE [4] : = 'PAUS';
   VALOR  [1] : = 'AS';     VALOR  [2] : = '2';      VALOR [3] : = '3';
   VALOR  [4] : = '4';      VALOR  [5] : = '5';      VALOR [6] : = '6';
   VALOR  [7] : = '7';      VALOR  [8] : = '8';      VALOR [9] : = '9';
   VALOR [10] : = '10';     VALOR [11] : = 'VALETE';
   VALOR [12] : = 'DAMA';   VALOR [13] : = 'REI';


   REPEAT                                 (*  Parte principal do programa    *)
      REPEAT
         WRITE (OUTPUT, 'Entre um n.º de carta ou zero para terminar: ');
         READLN (INPUT, CARTA);
         IF (CARTA < 0) OR (CARTA > 52) THEN
            WRITELN (OUTPUT,
                  '***** Favor entrar um numero de 0 a 52');
      UNTIL (CARTA > = 0) AND (CARTA < = 52);
```

```
        IF CARTA < > 0 THEN          (* Diferente de 0, processa a carta  *)
            LISTAGEM;
    UNTIL CARTA = 0;
END.                                 (* Fim do programa                   *)
```

**Listagem 2.6** — Exemplo do Programa em Pascal

Pascal é uma linguagem com forma livre que não requer um formato determinado para cada linha do programa. As instruções são separadas por um ponto-e-vírgula (;). Diversas instruções podem ficar numa mesma linha, ou uma só instrução pode ocupar diversas linhas. Todas as variáveis e sub-rotinas devem ser definidas antes de serem usadas. A instrução PROCEDURE LISTAGEM define a sub-rotina do relatório. Reparem que a instrução CASE foi mudada para se calcular o naipe exato da carta em lugar de se usar a relação de valores mostrada no projeto de código. Claro que esta não é a única forma de se escrever o programa. Por exemplo, pode-se substituir a instrução CASE pela linha:

```
NUM-NAIPE:=((CARTA-1) DIV 13)+1;
```

A rotina LISTAGEM é chamada na terceira linha de baixo para cima, usando-se simplesmente a palavra LISTAGEM. Qualquer coisa entre pares de caracteres "(*" e "*)" é um comentário. Pascal não tem formas como as instruções READ e DATA do BASIC para carregar as matrizes, de forma que o programa começa (no comando BEGIN) carregando as matrizes com os valores adequados. As instruções WRITE e WRITELN indicam saída no vídeo e a instrução READLN "lê" as variáveis pelo teclado. O LN no final das instruções READ ou WRITE indica que se está lendo ou escrevendo uma linha de informação completa. Os arquivos padrão INPUT e OUTPUT definidos na instrução PROGRAM são usados para essas operações. Este exemplo é escrito em Pascal UCSD* que permite um string de caracteres como um tipo de variável. Todos os programas em Pascal, neste livro, serão escritos em Pascal UCSD.

Neste ponto, todos devem estar aptos a entender o que cada instrução de cada exemplo apresentado faz. Cada pessoa deve usar o manual de programação do sistema que possuir para obter as explicações específicas das diversas instruções.

## VIII. REVISÃO DAS PRINCIPAIS IDÉIAS

Este capítulo tratou das considerações sobre como escrever instruções para os computadores. As idéias principais, a seguir, devem ficar bem entendidas antes que se continue a ler o restante do livro:

---

* Pascal UCSD é marca registrada dos Membros do Conselho da Universidade da Califórnia.

- A única tarefa do computador é manipular a informação que está em sua memória e comunicar os resultados ao mundo exterior. Um número relativamente pequeno de instruções controla sua operação.
- Os tipos primários de dados com os quais um programador lida são números e séries de caracteres. Em algumas linguagens, outros tipos de dados podem ser possíveis.
- Os dados de um programa consistem em constantes, que nunca mudam, e variáveis, que são identificadas por nomes e podem ter um valor que seja adequado ao tipo de dados da variável.
- As estruturas de dados podem ser desenvolvidas para organizarem elementos de dados relacionados em matrizes, registros, ou outros tipos de dados.
- As máquinas e os programadores raramente usam a mesma linguagem. Existe geralmente um programa de tradução usado para converter as instruções da linguagem de programação para um programa em linguagem de máquina. Na maioria dos casos, inclusive nos exemplos deste livro, usam-se linguagens de alto nível.
- As linguagens de alto nível vêm em diversos dialetos, de modo que os programas usados num computador podem não servir para outro computador. Um programa pode necessitar de modificações para ser usado num computador que utilize uma versão diferente da mesma linguagem.
- As instruções para um programa devem ser bem específicas para que o computador, que só faz o que se lhe diz para fazer, possa executá-las.
- Uma vez que se saiba o que um programa deve fazer, as instruções podem ser desenvolvidas usando-se um projeto de código independente da linguagem a ser usada. O programa final será escrito então na verdadeira linguagem de programação a partir desta delineação inicial.

# 3

# Quando um Programa Deve Ser Escrito (Estudo de Viabilidade)

*"Comece pelo início."*
Lewis Carroll

## I. O INÍCIO DE UM PROJETO DE PROGRAMAÇÃO

Nos dois primeiros capítulos desenvolvemos muitas das ferramentas necessárias para se escrever um programa de computador. Agora que temos as ferramentas com que trabalhar, podemos começar a aprender como usá-las. Vamos fazer isso partindo logo para o projeto de desenvolvimento de um programa. É aqui que começa realmente toda a graça da programação.

O entusiasmo com o programa é geralmente muito grande neste ponto, e há uma inclinação natural para se sentar diante do computador e começar a programar. Infelizmente, é como se corrêssemos até a cozinha e misturássemos diversos ingredientes para ver se conseguiríamos fazer um bolinho de aveia. Embora algumas cozinheiras possam fazer isso às vezes, certamente não é recomendável agir-se desta maneira, se o que se deseja é um produto de alta qualidade sem frustrações nem pseudocomeços.

Neste projeto, vamos usar o método top-down descrito no primeiro capítulo. Este método envolve muito mais planejamento e projeto antes de se começar a escrever um programa, mas depois que se iniciar a programação ela será feita de forma muito mais rápida e com menos dificuldades.

Grandes projetos comerciais que são desenvolvidos ao longo de muitos meses ou anos são geralmente divididos em partes menores. Embora não haja neces-

sidade de se ser tão formal quando se trabalha com um programa menor ou com um projeto de interesse pessoal, não é má idéia dividir esses projetos em partes menores da mesma forma. Seja como for, esta técnica propiciará maior êxito numa programação pessoal, permitindo, assim, que se tire maior proveito dela.

Toda a vida de um programa ou sistema de programas relacionados, desde sua concepção até deixar de ser usado, chama-se *ciclo de vida do projeto.* O ciclo de vida de um projeto de tamanho relativo aparece na figura 3.1.

O primeiro passo do projeto, chamado *estudo de viabilidade,* define o que deve ser feito e determina se isso é possível. Este planejamento é geralmente desconsiderado em projetos pequenos, mas os resultados sempre sofrem com isso. O *projeto geral* é onde o programa começa realmente a ganhar forma. Os principais itens de dados e as funções básicas do programa são definidos e relacionados entre si. No *projeto detalhado,* o projeto geral é subdividido e os detalhes específicos incluídos. Desenvolve-se um dicionário com todos os itens de dados, e o pseudocódigo do programa é escrito. Só na fase da *implementação* é que o programa é realmente escrito numa linguagem de programação, testado e processado no computador. O total de tempo e esforço despendidos geralmente em cada passo do projeto está representado no gráfico pelas áreas correspondentes.

O limite entre um passo e outro é mostrado como uma linha inclinada, porque não há pontos definidos em que se pare de executar uma fase para se dar início a outra. O primeiro passo define de maneira geral as funções a serem executadas. O projeto deve ser, então, uma suave progressão no sentido de definições mais detalhadas até que o programa inteiro esteja terminado. Os detalhes envolvidos num nível de desenvolvimento podem forçar a mudanças na concepção anterior do projeto, de forma que existe uma movimentação para trás e para frente ao longo do curso de desenvolvimento do projeto. O certo, contudo, é

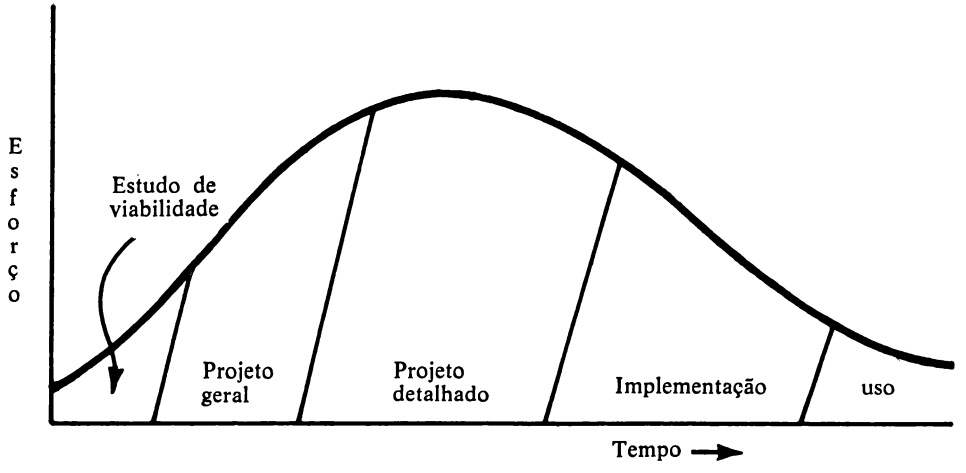



**Figura 3.1** — Ciclo de Vida de um Projeto Simples

realizar-se cada passo da melhor maneira possível para que esses retornos sejam mantidos num mínimo indispensável. Quando todos os passos do projeto estiverem finalmente completados, o programa é escrito, na fase de implementação. Os limites entre as fases são usados principalmente como pontos de checagem para se realizar uma revisão de todo o projeto.

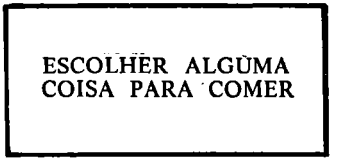
Uma vez que o programa esteja em uso, pensa-se sempre em acréscimos ou correções que poderiam tornar o programa melhor. Isso leva geralmente a mudanças no projeto original e pode levar mesmo a um projeto completamente novo. Quando se trabalha com um programa durante muito tempo, é comum surpreendermo-nos no início de um novo ciclo de vida do projeto.

Para mostrar como essas fases funcionam num desenvolvimento de projeto, cada um dos capítulos subseqüentes do livro se ocupará de uma das fases do projeto. Este capítulo começa com a etapa inicial, determinando o que desejamos fazer e se é possível fazê-lo.

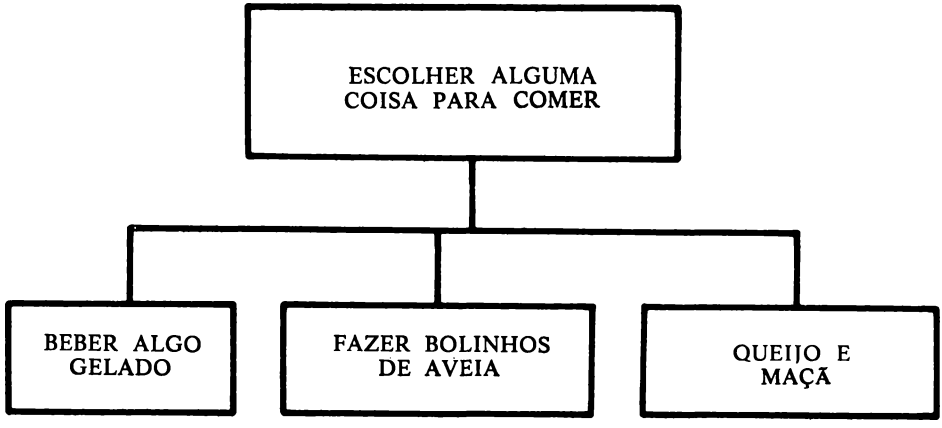
## II. IDENTIFICANDO O PROBLEMA

A coisa mais importante para se ter em mente no início de um projeto é que o enunciado do problema deve ser específico. "Vamos escolher alguma coisa para comer" pode fazer uma pessoa levantar-se e ir até a cozinha, mas é muito vago para fazer alguma coisa além disso. Uma vez na cozinha, defronta-se com a verdadeira decisão: o que escolher.

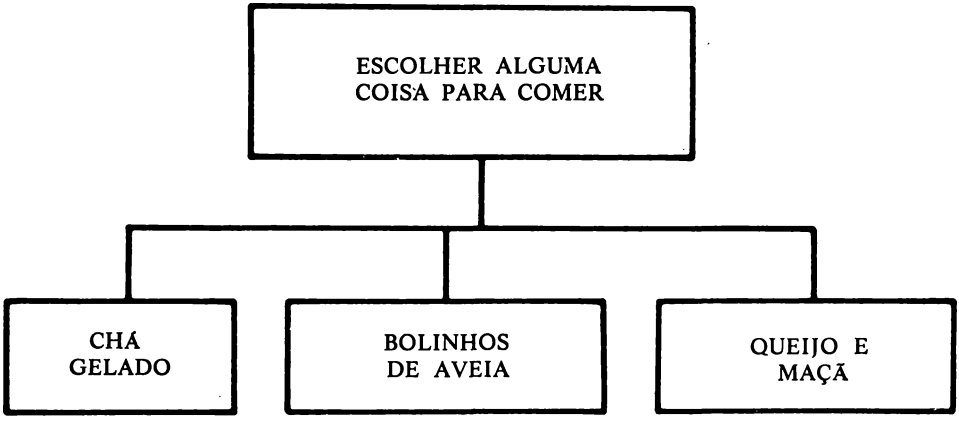
A definição de um projeto deve ser apurada até que se torne bem precisa. Se a idéia que nos ocorrer em primeiro lugar for muito ampla, deve-se estreitá-la até um item específico ou subdividi-la em diversos itens. Este processo de subdivisão de funções amplas em diversas funções menores é o princípio básico do desenvolvimento top-down, que será exposto mais detalhadamente no capítulo seguinte. Por enquanto, vamos fazer uma abordagem um tanto informal em nossa análise. Todo o projeto pode ser representado inicialmente como uma simples função:




Olhando a despensa, descobrimos que ainda não sabemos o que fazer. Assim, aperfeiçoamos a definição do problema para incluir três ações específicas: escolher alguma coisa gelada para beber, fazer bolinhos de aveia ou arrumar um prato com queijo e maçãs cortadas. A função de ESCOLHER ALGUMA COISA PARA COMER não foi alterada; simplesmente foi apurada.




Como se pode ver, não defini nenhuma função nova, mas simplesmente aperfeiçoei a função original. Todas as funções da parte inferior desta estrutura estão ligadas à função da parte superior. Quase todas parecem boas, mas BEBER ALGUMA COISA GELADA ainda é muito vago. Precisamos saber que tipo de bebida escolher. Um aperfeiçoamento mais preciso é o seguinte:




Desta vez melhoramos a definição do problema ao sermos mais específicos. Podemos agora refinar a definição do problema através da subdivisão de cada função em diversas subfunções. Este processo top-down deve ser executado até que todo o projeto esteja enunciado em termos de funções específicas que sejam claramente entendidas.

Em muitos casos, o projeto como originariamente enunciado acaba sendo mais amplo do que realmente se desejou. A função geral pode então ser redefinida para refletir esta mudança. No caso de escolher alguma coisa para comer, podemos decidir que os bolinhos de aveia seriam o suficiente, por enquanto. O projeto será então redefinido, como a seguir:


Num grande projeto de programação comercial, este processo de definição do problema pode levar meses para se completar. Geralmente o futuro usuário do programa não sabe qual o problema ou o que é preciso para que a tarefa seja feita. Num pequeno projeto, este processo pode ser feito de maneira informal. Entrar na cozinha, por exemplo, a fim de fazer um lanche, e decidir-se pelos bolinhos-de-aveia pode levar apenas alguns segundos. Definir um projeto de programação pode levar alguns dias, enquanto se conversa com outras pessoas interessadas no que se está fazendo, faz-se um esboço das idéias e tenta-se alternar os enunciados do problema. Independente de quanto tempo se gastar, o mais importante é definir o problema como uma função específica a ser executada. Se o projeto cobre um assunto muito vasto, desenvolver subfunções é uma solução. Deve-se saber, antes de se começar a programar, o que precisamente se vai fazer.

Mas não é preciso ter uma perfeita definição do problema no início do projeto. E não se deve pensar que é impossível mudar depois de iniciado o projeto. Uma das razões de se deixar a fase de escrever o programa para o final do processo é justamente a possibilidade de se mudar de idéia antes de se começar a programar. Quase todo mundo desenvolve uma idéia mais específica do que deve ser feito à medida que o projeto avança. O que se deseja no início é uma idéia específica sobre onde queremos chegar. Se este objetivo for modificado no meio do caminho, ótimo.

### A. O Que Vamos Fazer?

Grande parte dos projetos de programação que obtém êxito começa com um problema. O interesse da pessoa no problema leva-a a investigar o que é preciso para resolvê-lo e, finalmente, desenvolver uma solução. É mais provável obter-se êxito com os projetos pelos quais se tem interesse do que com aqueles que se começa apenas pela vontade de escrever um programa. Por isso, o restante deste livro tentará definir um objetivo interessante, desenvolver um programa para este objetivo e usar o exemplo como um benefício extra à medida que avançamos.

Nosso projeto deve ser amplo o bastante para servir como um exemplo, mas não muito complexo para uma introdução ao projeto de programa. Deve tratar de um assunto que a maioria dos leitores considere familiar e fácil de entender. Finalmente, deve produzir um programa que seja divertido e fácil de usar.

*Hei! Tive uma idéia brilhante — vamos desenvolver um programa que possa jogar cartas conosco!*

A maioria dos projetos começa nesse nível de detalhes. Agora, vamos aperfeiçoar isso em algo que possa ser utilizado.

**B. O Enunciado do Problema para Nosso Exemplo**

Até aqui nossa "idéia brilhante" é apenas uma vaga noção: fazer o computador jogar cartas conosco. Esta idéia deve ser desenvolvida para incluir funções específicas necessárias para se jogar qualquer tipo de jogo de cartas, ou seja, deve ser delimitada a uma subfunção mais específica de jogo de cartas. A Figura 3.2 mostra algumas das subfunções que podem ser desenvolvidas.

Há muitas formas para se subdividir a função "jogar cartas". Alguns jogos podem se enquadrar sob mais de uma categoria "objetivo do jogo" e sob mais de um grupo de "número de jogadores". Se levássemos em consideração todos os exemplos de todos os tipos de jogos de cartas para qualquer número de jogadores, terminaríamos com centenas de funções para definir. Talvez fosse mais fácil escolher um jogo de cartas específico e concentrarmo-nos nisso.

Devemos evitar os jogos muito complexos como o *Panguingue*, que pode não ser muito conhecido. Devemos evitar também os jogos que dependam muito de se lembrar das cartas jogadas. Os computadores são ótimos para lembrar o que fizeram, e não tem graça nenhuma jogar com um adversário que nunca se engana nem esquece.

Depois de considerar vários jogos de cartas, achamos que a versão do pôquer chamada Vinte-e-Um tem todas as características de que necessitamos. O jogo é conhecido da maioria das pessoas e tem regras bem simples. A parte de quem dá as cartas é bem definida. E o jogo requer apenas o banqueiro e um outro jogador. Por isso, a primeira instrução para nosso projeto será esta função:

| FAZER O COMPUTADOR JOGAR VINTE-E-UM |
|---|

Isso poderá ser futuramente aperfeiçoado pela definição de que papel o computador desempenhará e que regras serão usadas no jogo. A maior parte

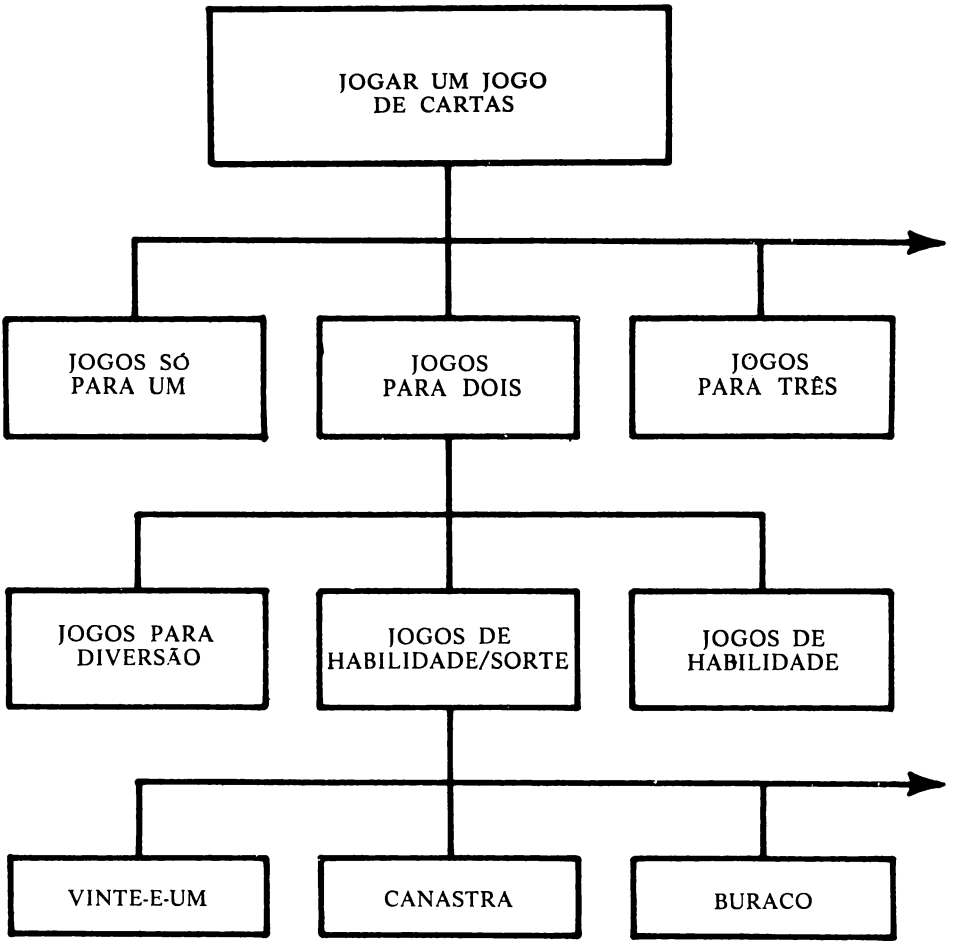



**Figura 3.2** — Principais Funções de Jogar Cartas

das pessoas que joga vinte-e-um no cassino joga contra uma banca permanente. Pode ser mais natural, por isso, que o computador seja o banqueiro e que joguemos contra ele. O jogo será desenvolvido de acordo com as regras do jogo de vinte-e-um, com uma banca permanente, publicadas no *Hoyle's Official Rules of Card Games* (Fawcett Crest Books, New York, 1968).

Agora, restringimos nossa idéia original a uma função específica em lugar de refiná-la em várias subfunções detalhadas. Esta função única pode ser enunciada assim:

FAZER O COMPUTADOR
SER O BANQUEIRO NUM
JOGO DE VINTE-E-UM
DE ACORDO COM HOYLE

O objetivo do nosso projeto será desenvolver um ou mais programas, juntamente com seus dados associados, que satisfaçam este enunciado do problema. Pode-se chamar de sistema a um grupo de programas e dados assim.

O termo *sistema de computador* é usado de formas diferentes pelas pessoas. Em seu sentido mais amplo, abrange equipamento, programas, dados e procedimentos necessários à execução de uma função determinada. Neste livro não trataremos do equipamento (chamado *hardware*) nem de como ele é usado. Em vez disso, vamos enfocar os programas (chamados *software*) que trabalham no *hardware*. Assim sendo, usaremos o termo sistema de *software*, ou simplesmente sistema, quando nos referirmos aos programas e dados desenvolvidos para serem processados como parte de um sistema de computador completo. Nosso objetivo, então, é desenvolver um sistema de *software* para desempenhar o papel de banqueiro num jogo de vinte-e-um, segundo as regras de Hoyle. Vamos resumir esta descrição para "Sistema Vinte-e-Um".

## III. ISSO PODE SER FEITO?

Decidir o que vamos fazer faz parte do primeiro passo do estudo de viabilidade. O próximo passo é ver se o que desejamos fazer é possível de ser feito. Há duas áreas gerais a considerar, quando se estiver checando a possibilidade de execução de um projeto. A primeira, determina se o equipamento disponível pode realizar as funções requeridas. A segunda, se a pessoa é capaz de escrever o programa ou programas para a tarefa. Assim como o problema original de definição, este processo pode levar um tempo que varie entre alguns minutos e o tempo necessário a uma investigação minuciosa.

### A. É Mais do que Meu Equipamento pode Executar?

Antes que tenhamos escrito alguns programas que atinjam os limites de nosso computador, será muito difícil responder a esta pergunta com certeza. Há algumas coisas a considerar, ao se iniciar um novo programa: (1) Haverá memória suficiente para armazenar o programa e os dados (e, com BASIC, talvez um intérprete separado ou um pacote em tempo de execução)?; (2) O computador poderá operar rápido o suficiente para tornar o programa útil?

A maioria dos pequenos computadores é vendida com uma quantidade relativamente pequena de memória, a fim de reduzir o custo inicial da máquina. Mas

pode-se, geralmente, aumentar a capacidade da memória a uma quantidade máxima. Com um pouco de prática, logo se tem uma boa idéia do tamanho de programa que a memória terá capacidade de armazenar. Uma lei de programação bem conhecida diz que, independente da capacidade de memória de que se disponha, sempre haverá um projeto de nosso interesse que será um pouquinho amplo para a capacidade da máquina que se possui.

A velocidade de execução apresenta um tipo diferente de problema. Não há, de um modo geral, um meio de se aumentar a velocidade de um computador. A principal dificuldade surge quando o computador precisa checar um determinado volume de informação num curto espaço de tempo, como num jogo interativo. Isto é mais provável de acontecer nos sistemas que utilizam interpretador. A velocidade de execução e o tamanho do programa são os motivos pelos quais os programas desenvolvidos pelo fabricante como parte do sistema do computador são escritos a nível de linguagem de máquina.

As considerações sobre espaço de armazenamento e velocidade de execução também se aplicam à informação armazenada fora do computador, em fitas magnéticas ou unidades de disco. Essas unidades chamam-se dispositivos externos de armazenamento, e podem conter grandes quantidades de dados. O tamanho dos arquivos armazenados em tais dispositivos pode ser estimado calculando-se quanto espaço será necessário para cada item do arquivo e multiplicando-se o resultado pelo número de itens a serem armazenados. Por exemplo, se uma receita média requer 500 caracteres e desejamos colocar 500 receitas no arquivo, ele terá que ter espaço para 250 mil caracteres ($500 \times 500 = 250.000$). A velocidade dos dispositivos externos é muito menor do que a do computador, e o armazenamento em fita é muito mais lento do que o armazenamento em disco.

Em certas ocasiões, haverá outros dispositivos a considerar. Se desejamos relatórios, devemos ter uma impressora capaz de imprimir os caracteres apropriados num papel largo o suficiente para conter os relatórios. Uma impressora sem letras minúsculas, por exemplo, não seria uma boa escolha para a produção de cartas comerciais. Outros projetos podem envolver equipamentos de fabricação caseira como robôs simples, controladores de trenzinhos elétricos ou qualquer outra coisa. Quem planeja usar equipamentos assim deve se certificar de suas limitações antes de iniciar o projeto.

### B. É Mais do que Desejo Fazer Agora?

A segunda área a considerar quando se avalia a possibilidade de execução de um projeto é determinar se somos capazes de escrever programas. Esta avaliação é muito subjetiva. O objetivo desta avaliação é evitar projetos longos e que requeiram muito esforço, podendo se converter em insucessos frustrantes por se tratar de projetos que nos desafiam com a necessidade de mais conhecimento e maior capacidade.

A regra mais importante é entender o enunciado do problema. Se a própria pessoa desenvolve a definição do problema, como explicado acima, isso não

deve criar dificuldades. Se o problema é tirado de um artigo de revista ou a idéia é de outra pessoa, é bom certificar-se de que realmente se entendeu o que deve ser feito, antes de começar. Poucas coisas são tão frustrantes quanto se trabalhar num projeto que não se entende.

A próxima coisa a decidir é se existe tempo suficiente para se trabalhar no projeto. Começar um projeto que necessite de muito planejamento, programação e avaliação quando não se tem tempo nem para realizar o que se está fazendo no momento, significa que nunca se terá tempo de concluí-lo. Somos nossos melhores juízes quanto ao tempo de que dispomos, mas devemos nos lembrar de que programação requer longos períodos de reflexão e trabalho ininterruptos.

Um item final a considerar é o que será preciso aprender para que se compreenda todo o projeto. Um projeto que faça um robô andar por uma sala pode requerer conhecimentos de eletrônica, projeto de máquinas, geometria e diversas outras disciplinas. Claro que não é preciso que se conheça tudo sobre todos esses itens no início, mas certamente será preciso conhecer essas matérias à medida que se avança com o projeto. É bom tentar alguns projetos que envolvam áreas de que já se tenha algum conhecimento ou pelas quais se tenha um certo interesse em saber algo mais. Quando se tem que aprender muita coisa sobre um assunto que não nos interessa, acabamos perdendo o estímulo e não terminamos o projeto. Isso tira rapidamente o prazer da programação.

O momento de considerar tudo isso é agora — antes que o problema ocorra. Claro que não se pode antever tudo. O melhor momento para se pensar em parar qualquer projeto em favor de outro com maiores possibilidades de obter êxito é quando se estiver consciente do problema. O objetivo da abordagem top-down é identificar problemas insuperáveis logo que possível, a fim de se evitar perda de tempo num projeto que não poderá ser concluído.

### C. O Programa do Vinte-e-Um Pode Ser Feito?

Os testes para a viabilidade de execução de um projeto podem ser aplicados agora a nosso Projeto Vinte-e-Um. Este jogo já foi programado por muitas pessoas; deve funcionar bem na maioria dos pequenos computadores da atualidade. A principal dificuldade está nos gráficos. Se um jogo completo com boa visualização de tela for desenvolvido, pode ser muito extenso para alguns computadores com pequena capacidade de memória. Podemos ter que desenvolver uma versão em escala menor do programa para as máquinas menores. Esta versão em escala reduzida diminuiria o uso de matrizes e séries de caracteres do programa, já que são geralmente limitadas nos pequenos computadores. Não há requisito de tempo no jogo, de forma que a velocidade não seria problema. E, finalmente, não haveria arquivos armazenados em dispositivos externos.

O desenvolvimento de programas de vinte-e-um não deve ser muito difícil para iniciantes. Todas as pessoas que não conhecerem as regras do jogo terão que estudar duas páginas das *Regras Oficiais de Jogos de Cartas* (Apêndice B deste

livro). O programa não exigirá muito tempo para ser desenvolvido, mas oferece oportunidade de um projeto engenhoso e programação elegante.

Não deve haver surpresa, pela inexistência de razões que obriguem a desistir do projeto.

## IV. ENCONTRANDO AJUDA EXTRA

Agora que decidimos o que vamos fazer, o próximo passo deve ser verificar se alguém já fez isso. Tradicionalmente, os programadores de computador gastam muito tempo reinventando programas que já tenham sido desenvolvidos e testados. Mesmo que ninguém tenha feito exatamente o que se deseja fazer, deve haver partes do projeto que já foram desenvolvidas.

Em muitos casos, há programas para computadores comerciais que servirão para as nossas necessidades. As lojas de computadores geralmente oferecem uma grande gama de produtos de *software* que podem ser usados imediatamente. Também uma vez obtida a listagem do programa, pode-se modificá-lo a fim de que sirva às necessidades que se tem em mente. Isso é geralmente possível nos sistemas pequenos que usam interpretadores BASIC, já que nesse caso os programas originais rodam diretamente no computador. Os programas desenvolvidos com um compilador ou semicompilador não são vendidos, de um modo geral, com a listagem do programa original e por isso são difíceis de ser modificados. É preciso, contudo, ter-se muito cuidado ao se modificar o programa de outra pessoa. Isso certamente tornará nula qualquer garantia que o programa possa ter. Além disso, quem vende ou distribui programas alterados está com toda certeza infringindo os direitos autorais do autor original. Quem desejar mudar os programas de outra pessoa, deve tirar uma cópia extra do programa, guardar o original e certificar-se de que ninguém mais usará o programa modificado.

Outro lugar onde procurar ajuda seriam os livros e os artigos sobre programação. Há muitas revistas que trazem artigos sobre a solução de problemas de programação. Esse tipo de revista também responde a perguntas comuns enviadas pelos leitores. Mesmo que não se encontre nada de utilidade para o momento, pode-se aprender alguma coisa que será útil no futuro. As revistas também são uma maneira excelente de se manter atualizado sobre os programas disponíveis diretamente nas lojas ou por remessa postal.

No caso do nosso programa de vinte-e-um, podemos comprar um dos diversos programas existentes e fazer com que funcionem. Porém, desenvolver o programa por nós mesmos será mais instrutivo. Além do que, há sempre a sensação de que "Fiz isso sozinho". Já que nesse caso está se programando por distração, esta atitude é aceitável. Mas quando se está interessado em ter um produto final de qualidade com o mínimo de perda de esforço, é bom lembrar que há vários lugares onde se pode ir em busca de ajuda — inclusive para comprar o programa inteiro logo de saída.

## V. REVISÃO DAS PRINCIPAIS IDÉIAS

Este capítulo serviu a dois propósitos. Primeiro, definiu um projeto de programação como um ciclo que pode ser dividido em várias etapas — a partir de um estudo de viabilidade, no início, até o uso do produto final. Isso nos levou então ao primeiro passo de nosso Projeto de Vinte-e-Um. Os pontos a seguir são as principais idéias apresentadas neste capítulo:

- Um projeto de programação é mais fácil de se lidar quando é subdividido em diversos passos, como mostrado no Ciclo de Vida do Projeto.
- O primeiro passo do ciclo é desenvolver um enunciado específico do problema como uma função a ser executada, e determinar se a função é possível.
- Um problema com enunciado vago pode ser refinado subdividindo-o em subfunções, reduzindo-se o objetivo da idéia original numa, ou pela combinação das duas técnicas.
- Uma vez que se tenha escolhido um projeto específico, o equipamento disponível deve ser testado para se ver se ele é capaz de suportar tudo que deve ser feito.
- O tempo e o interesse da pessoa como programador devem ser considerados.
- Como primeiro passo no sentido de desenvolver o projeto, deve-se procurar ajuda nos programas já existentes ou em artigos sobre programação que podem se aplicar ao caso que se tenha.
- As razões para essa preparação são (1) conhecer, antes de se iniciar, o que deve ser feito, e (2) identificar, logo que possível, os projetos que não são viáveis ou desejáveis.

# 4

# O Que o Programa Deve Fazer? (O Projeto Geral)

*"Construa de baixo para cima e não de cima para baixo."*
Franklin Delano Roosevelt
*"A heresia de uma geração torna-se a ortodoxia da seguinte."*
Helen Keller

Um dos principais objetivos do capítulo anterior foi identificar que tipo de sistema de *software* vamos desenvolver, e definir sua função como um todo. Neste capítulo, continuamos este processo subdividindo a função de alto nível em subfunções menores até que cada tarefa a ser executada tenha sido definida em termos de diversas funções simples. Definiremos também os tipos de dados necessários, identificaremos de onde eles virão e determinaremos como eles serão usados. Quando tivermos terminado a segunda etapa — o Projeto Geral — saberemos tudo o que o sistema deve fazer, quantos programas serão necessários para fazê-lo, de que informações precisaremos, de onde essas informações virão e como, finalmente, serão usadas.

Da mesma forma como existem várias maneiras de se classificar os jogos de cartas, há geralmente muitas maneiras de se organizar uma função de programação complexa. O projeto desenvolvido neste livro não é a única solução "correta". Depois que se termina de fazer alguns projetos de programação, desenvolve-se um estilo próprio de programação — ou seja, cada pessoa tem seu método particular de percorrer as etapas de um projeto de sistema. Os procedimentos usados no projeto estruturado de top-down seguem uma linha de desenvolvimento bem ordenada, mas independem do estilo do programador.

Há duas maneiras de se começar a desenvolver nosso Projeto Vinte-e-Um. Uma delas é começar definindo o que desejamos como saída final e depois desenvolver a entrada e as funções necessárias para se obter esta saída. A outra forma é começar com as funções que devem ser executadas e depois desenvolver os dados que essas funções usarão como entrada e saída. Antes de o Projeto Geral estar completo, teremos que trabalhar nas funções e dados, de forma que possamos começar pelo que for mais fácil. Isso geralmente depende do projeto: se ele for basicamente *processamento de dados*, em que os dados são mais importantes e o processamento é relativamente simples, ou se for *processamento de funções*, em que os procedimentos (funções) executados são mais importantes e os dados relativamente simples.

No caso do Projeto Vinte-e-Um começaremos pela análise das funções a serem executadas, porque estamos interessados primeiramente nos procedimentos do jogo. Além disso, temos uma boa lista deles nas *Regras Oficiais dos Jogos de Cartas*. Lembrem-se, contudo, de que poderíamos ter começado igualmente pelos dados.

## I. DIVIDINDO AS FUNÇÕES EM UNIDADES MENORES

O objetivo de analisarmos as funções em nosso sistema é podermos subdividir cada função completa em outras mais simples. A tarefa de cada função, nesse caso, passa a ser a de coordenar a operação de suas subfunções. Se esses subelementos ainda forem complexos, podem, por sua vez, ser subdivididos em unidades menores. Este processo continua até que tudo que deva ser feito no sistema esteja definido em termos de operações simples. Cada uma dessas operações, que se chama módulo do sistema, executa uma única e bem definida função. O trabalho do sistema como um todo será o de controlar as operações desses diversos módulos. Durante a implementação, cada módulo será traduzido para o verdadeiro código da linguagem de computador.

Há dois princípios gerais que devemos ter em mente ao decidirmos como dividir uma função complexa em subfunções menores:

1. Toda operação de uma função deve estar diretamente relacionada com todas as outras operações daquela função.
2. As operações de uma função não devem estar diretamente relacionadas às operações de nenhuma outra função.

As instruções da programação estruturada desenvolvidas no primeiro capítulo são excelentes exemplos desses princípios.

### A. Força e Acoplamento

À medida em que as operações de um único módulo relacionam-se entre si chamamos *força* do módulo. Cada módulo deve executar apenas uma função.

Uma instrução DO-WHILE, por exemplo, não contém nenhuma operação usada por uma instrução DO-UNTIL ou por qualquer outro tipo de instrução. Se alguma operação estranha for encontrada num módulo, deve ser colocada em outra função ou o módulo todo deve ser depois dividido em subelementos menores. Isso será muito importante mais adiante quando se começar a entender o que cada parte do sistema está fazendo.

À medida em que dois módulos têm processamento independente um do outro chamamos *acoplamento* entre eles. Como as instruções de programação, cada módulo do sistema deve ter um ponto de entrada e um ponto de saída. Isso tomará geralmente a forma de chamada e retorno de sub-rotinas. Quando chamado, cada módulo deve realizar uma única e bem definida função e retornar, independente da posição dos outros módulos do sistema. Isso também será muito importante mais adiante, quando for preciso fazer mudanças em alguma parte do sistema. Se uma função precisar ser mudada, não deve haver efeito colateral em nenhuma — nem de nenhuma — função que estiver acoplada a esta.

Num sistema bem projetado, todos os módulos devem ter força bastante e ser acoplados tão livremente aos demais como as próprias instruções de programação. Na vida real, não é sempre possível conseguir esses objetivos completamente, mas devemos tê-los em mente enquanto desenvolvemos nosso sistema vinte-e-um.

## II. DESENVOLVENDO A ESTRUTURA DO SISTEMA

Quando se define um sistema em termos de elementos e subelementos funcionais, que mostram o que controla o que, tem-se uma forma de hierarquia. Esta estrutura é geralmente desenhada na forma de uma árvore, similar à Figura 4.1.

Cada uma função do sistema está representada no quadro por um quadradinho com uma indicação da função. A função do sistema como um todo aparece num quadro do primeiro nível do desenho. Os subelementos de cada módulo formam o nível seguinte da ramificação desse módulo. Um módulo é composto por todos os seus subelementos e por todos os procedimentos necessários para o controle das operações desses subelementos. Quando o desenho se ramifica nos outros níveis inferiores, todas as funções complexas do sistema são definidas em termos de simples operações de "baixo-nível". Cada função complexa terá entre dois e oito subelementos, de um modo geral. A extensão de cada ramificação do desenho depende da complexidade das funções definidas nesta parte do sistema; não precisa haver relação entre o número de níveis de cada ramificação. O último módulo de cada ramificação deve ser uma única e simples função.

Ao discutirmos as relações entre os módulos numa hierarquia, os seguintes termos serão usados (referem-se aos nomes dos módulos da Figura 4.1):

A é pai de B e H.
B e H são irmãos.
B e H são descendentes de A.
D tem esses três tipos de parentesco com seus "vizinhos" (B, C, E, F, G).

Outros parentescos, como tias, tios e sobrinhos, são geralmente evitados porque não são precisos.

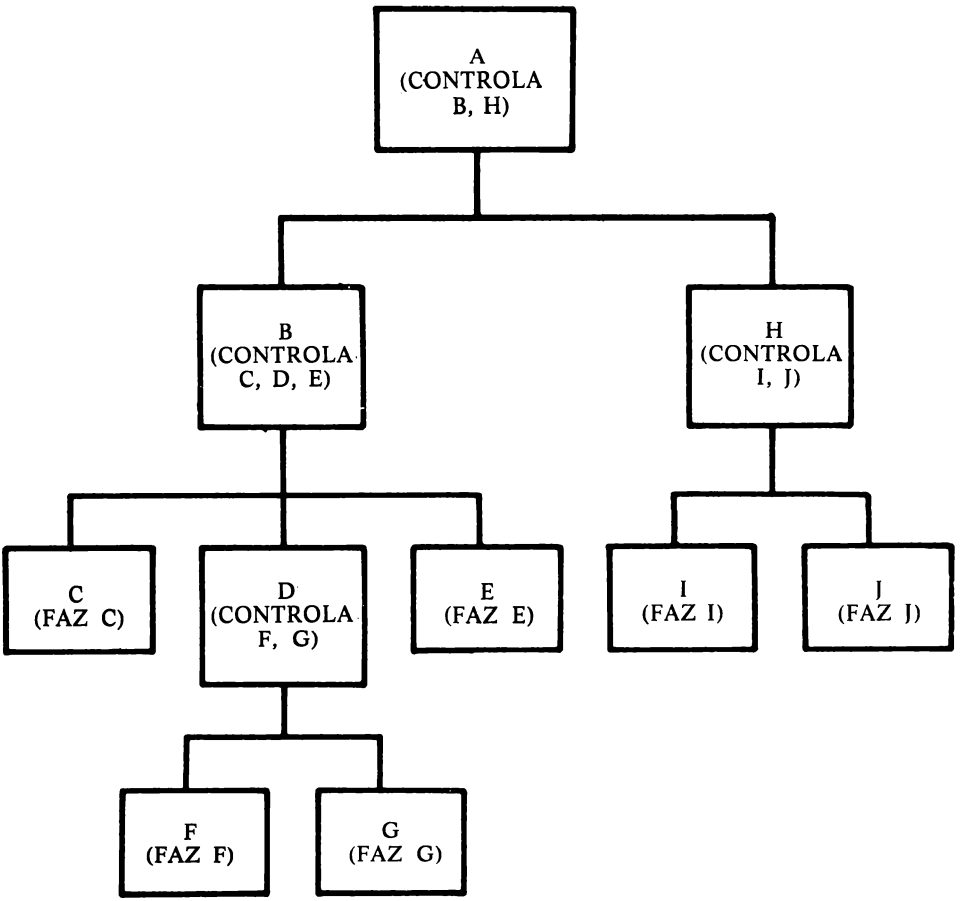



**Figura 4.1** — Uma Hierarquia Estrutural

Qualquer módulo de uma hierarquia pode ter diversos irmãos e descendentes, mas deve ter apenas um pai. É possível ter apenas um descendente se a execução de uma função é tão complexa que o controle daquele único módulo constitua uma função de alto nível.

Cada módulo do desenho pode usar seus descendentes chamando-os como sub-rotinas. Na hierarquia mostrada na Figura 4.1, a função A pode chamar a função

B e a função H para completar suas funções. Essas funções podem ser chamadas em qualquer ordem e quantas vezes for preciso para realizar a função A. Cada uma dessas funções pode, por sua vez, chamar seus subelementos, o que não é de responsabilidade da função A. Cada módulo é acoplado à sua descendência e a seu pai por chamadas e retornos, mas não tem conexão direta com nenhuma outra função do sistema.

Há casos em que uma função pode ser chamada por mais de um módulo de nível superior, ou "pai". Esta é uma estrutura mais complexa chamada de rede. Neste caso as ramificações da árvore podem convergir nos níveis mais baixos, como mostrado na Figura 4.2.

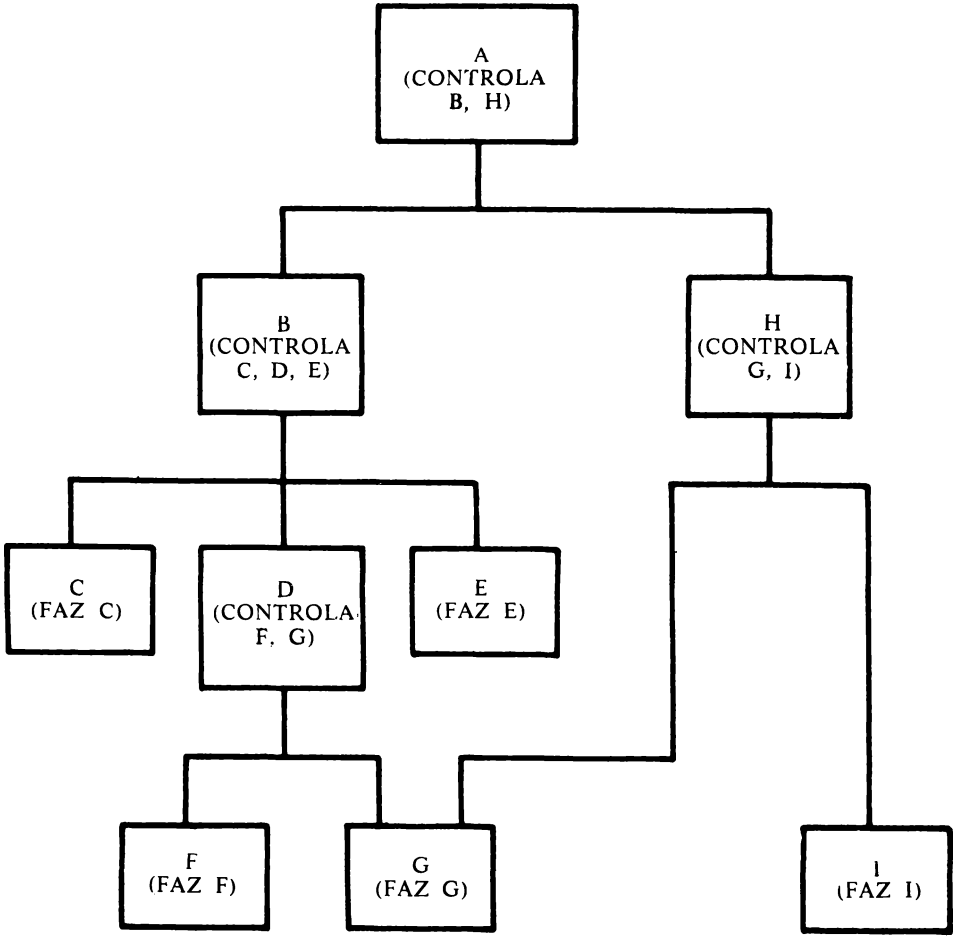



**Figura 4.2** — Uma Rede Estrutural

Nesta estrutura, G pode ser chamada tanto por D quanto por H. Contudo, ela sempre retorna ao módulo que a chamou. Quando um módulo é chamado por muitos lugares do sistema, é chamado de sub-rotina utilitária. Redes complexas são geralmente desenhadas com as sub-rotinas utilitárias agrupadas numa seção separada do desenho sem que apareçam todas as conexões delas. Isso evita que o desenho se torne confuso.

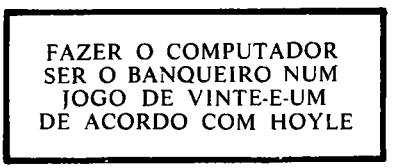
Desenvolver um projeto de sistema subdividindo as funções em rede oferece as seguintes vantagens:

- Tudo que o sistema deve fazer será definido durante a fase do projeto.
- Nos níveis mais elevados e complexos do sistema não é preciso se preocupar com os detalhes de como as coisas serão feitas. Cada subelemento é simplesmente chamado com uma chamada de sub-rotina.
- Quando se estiver trabalhando com os processos de baixo nível do sistema, deve-se lidar apenas com uma operação razoavelmente simples de cada vez.
- As funções que serão usadas em diversos lugares do sistema podem ser facilmente identificadas.
- Uma relação clara entre todas as operações do sistema é estabelecida.

## A. A Estrutura Inicial do Nosso Sistema Vinte-e-Um

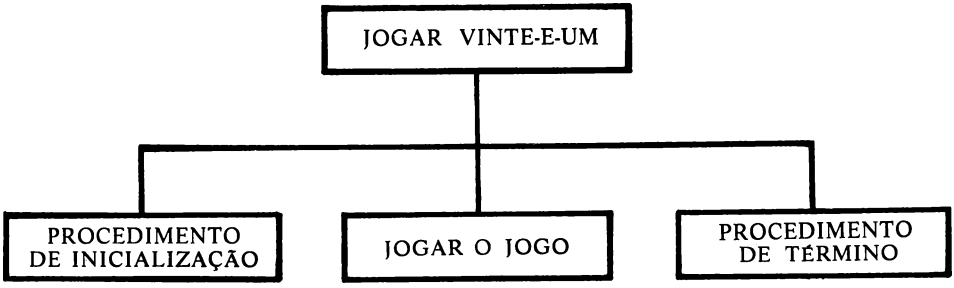
Está na hora de desenvolvermos os primeiros níveis do nosso Sistema Vinte-e-Um. Vamos começar dividindo as principais funções do sistema no primeiro nível de seus subelementos.

O início do Projeto Geral vai começar no ponto onde o capítulo anterior parou, usando o enunciado do problema do projeto como o nível mais alto da hierarquia do sistema. Assim sendo, a primeira representação de nosso projeto é:




Agora é hora de começarmos a dividir esta função em seus vários subelementos. Uma prática comum é começar pelos elementos chamados funções preparatórias. Essas funções incluem tanto as coisas que devem ser feitas antes de executar qualquer coisa, e que são chamadas de *procedimentos de inicialização,* como as que devem ser feitas no final de todo o processamento, chamadas *procedimentos de término.* Quase todos os sistemas terão algumas funções que se encaixarão nessas categorias.

No caso do jogo de vinte-e-um, por exemplo, haverá necessidade de carregar as matrizes com os nomes das cartas e de mostrar ao jogador uma tela com as instruções no início do jogo. Haverá, provavelmente, uma mensagem final e alguns totais para serem mostrados no final do jogo. Neste ponto não conhecemos os detalhes dessas operações, mas apenas que elas serão necessárias. Por isso, os dois primeiros níveis da estrutura de nosso sistema ficarão assim:




A função geral do sistema recebeu nova instrução: JOGAR VINTE-E-UM, para deixar as coisas mais fáceis. Os procedimentos de inicialização e de término, seja lá o que possam conter, foram criados a partir de como se joga o vinte-e-um. Agora podemos nos concentrar no desenvolvimento do jogo, sabendo-se que, quando qualquer das funções de inicialização ou de término aparecem, poderão ser colocadas em seu lugar apropriado, sob estas novas funções.

**B. Uma Estrutura mais Detalhada para Nosso Sistema Vinte-e-Um**

Agora vem a parte engraçada — deste ponto vamos para onde? Não há regras fixas ou procedimentos específicos para o desenvolvimento da estrutura de um sistema. Por isso cada programador desenvolverá os projetos de modo um pouco diferente. Como fazer então?

O processo de projeto torna-se mais fácil com a prática, de modo que não deve causar surpresa o fato de que algo que pareça fácil neste exemplo torne-se um pouco mais difícil quando se estiver trabalhando só. Veja o projeto geral como um quebra-cabeça no qual devemos encontrar todas as peças e juntá-las. Então é bom pegar um lápis e muito papel em branco e começar. O processo de projeto apresentado neste exemplo será o procedimento real que o autor usou para desenvolver o programa Vinte-e-Um.

A primeira coisa a fazer é identificar as funções de um modo ordenado. Isso pode ser conseguido começando-se a trabalhar da primeira função a ser executada e continuando-se até a última, ou usando-se qualquer outra ordem que ajude a organizar nossos pensamentos. Contudo, procurem sempre o mais alto nível de abstração possível. Depois comecem a trabalhar do alto do diagrama da estrutura até a parte de baixo, deixando os detalhes para mais tarde.

Já que nosso objetivo é jogar vinte-e-um segundo as regras de Hoyle, vamos começar lendo as regras do jogo nas Regras Oficiais de Jogos de Cartas. Essas

regras estão no APÊNDICE B, no final do livro. À medida que se encontrar uma nova função, devemos escrevê-la numa folha de papel sem nos preocuparmos (ainda) com a questão de como uma função se relaciona com outra. Para o jogo de vinte-e-um, a primeira lista de funções poderia ficar assim:

- APANHAR DOIS BARALHOS DE CARTAS
- EMBARALHAR AS CARTAS
- APOSTAR
- DISTRIBUIR AS CARTAS
- ATENÇÃO PARA OS NATURAIS
- NOVA CARTA PARA O JOGADOR
- NOVA CARTA PARA A BANCA
- PAGAR AS APOSTAS
- EMBARALHAR NOVAMENTE AS CARTAS
- SEPARAR OS PARES DE CARTAS

Isso cobre os pontos das regras e está quase que em ordem seqüencial de um jogo real.

Uma vez identificadas as funções iniciais, estas devem ser organizadas de alguma forma. A primeira coisa a fazer é ver se existem funções em duplicata. Neste caso, embaralhar as cartas e embaralhar novamente as cartas poderiam ser combinadas numa única função — embaralhar. Podemos indicar que às vezes é preciso durante a distribuição das cartas mudar a função DISTRIBUIR AS CARTAS para DISTRIBUIR E EMBARALHAR NOVAMENTE.

Quando as funções em duplicata tiverem sido eliminadas, a lista de funções deve ser organizada pelo agrupamento dos itens que tiverem alguma relação. Um que fica isolado dos outros é APANHAR DOIS BARALHOS DE CARTAS. Isso será feito apenas uma vez antes que a primeira mão seja jogada, e na verdade não faz parte do jogo. É uma função inicial e será listada dentro da primeira maior subfunção do nosso desenho. Outra função que fica de fora é embaralhar. Isso é feito no início do jogo e entre as mãos, mas não entre todas as mãos. Todas as demais funções são usadas durante a jogada de cada mão de vinte-e-um e podem ser agrupadas.

Tudo, com exceção de apanhar os baralhos, será listado em JOGAR O JOGO. A função SEPARAR OS PARES DE CARTAS é listada depois da mão dos naturais e antes que os jogadores comecem a pedir cartas, já que é aí que a separação de pares é normalmente feita. Ainda não há procedimentos de término.

Lembrem-se de que essas observações e as mudanças são um modo de se organizar a primeira lista de funções. Embora siga o padrão geral de eliminar as duplicatas, organizar por grupos e depois colocar os grupos em ordem, os detalhes de cada uma dessas decisões são pessoais. Esta reorganização resulta na seguinte lista modificada das principais funções:

PROCEDIMENTOS DE INICIALIZAÇÃO
- APANHAR DOIS BARALHOS DE CARTAS

JOGAR O JOGO
- EMBARALHAR AS CARTAS
- JOGAR UMA MÃO DE VINTE-E-UM
    - APOSTAR
    - DAR AS MÃOS E EMBARALHAR NOVAMENTE
    - ATENÇÃO PARA OS NATURAIS
    - SEPARAR OS PARES DE CARTAS
    - NOVA CARTA PARA O JOGADOR
    - NOVA CARTA PARA O BANQUEIRO
    - PAGAR AS APOSTAS

PROCEDIMENTOS DE TÉRMINO

Reparem que esta é uma lista tabulada que usa indentação para indicar relações, em lugar de uma hierarquia estrutural. O formato usado para mostrar informação estrutural não é tão importante quanto a informação em si. Contudo, essas funções também podem ser mostradas, usando-se descrições de módulo menores, como o diagrama estrutural da Figura 4.3.

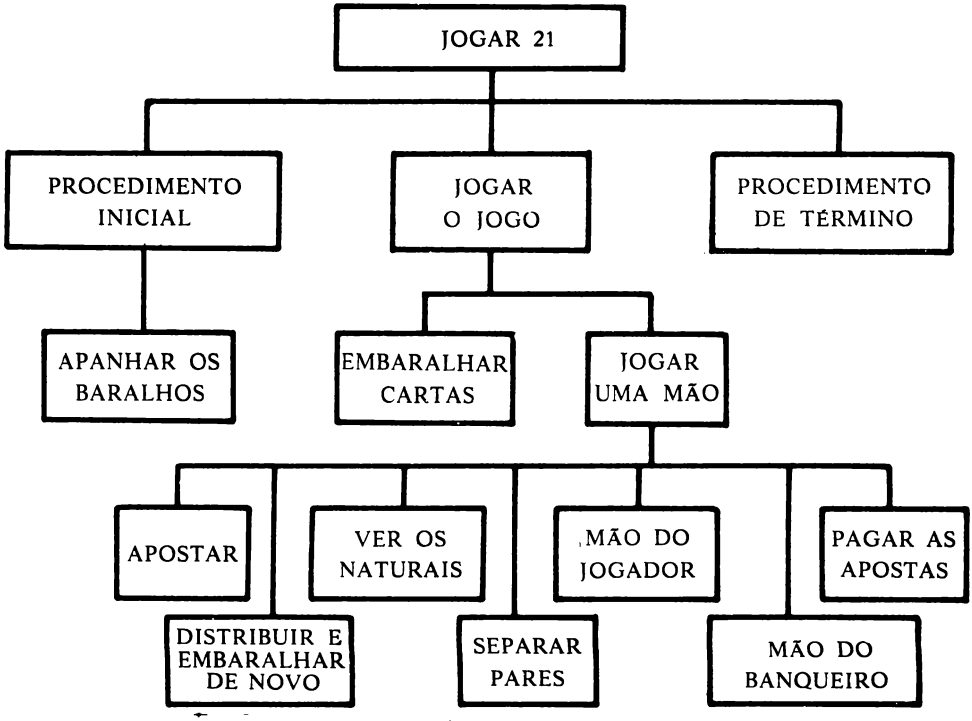



**Figura 4.3** — Um Formato Alternativo para Demonstrar as Funções

Há mais num jogo de cartas, porém, do que foi considerado até aqui. Isso porque as regras que usamos, como aparecem no Apêndice B, são para pessoas que têm algum conhecimento sobre jogos de cartas. Atividades comuns, como ter noção das fichas perdidas ou ganhas, ou o banqueiro e o jogador conversarem durante o jogo, são deixadas para o senso comum dos leitores. Num programa de computador, contudo, devemos definir *todos* os aspectos do jogo em detalhes.

Uma boa maneira para identificar qualquer parte do jogo que possa ter sido omitida é jogar um jogo imaginário. Enquanto se joga, pode-se perceber tudo que o computador precisa fazer para jogar o jogo. Na maioria dos projetos analisar um "exercício imaginário" ajuda a identificar todas as funções necessárias. A cena seguinte representa minha sessão imaginária na mesa de vinte-e-um, com o computador no papel de banqueiro:

> Ao me aproximar de uma mesa de jogo, num cassino, pergunto ao *crupiê* qual é o jogo e como se joga. Em resposta, o *crupiê* diz que é um jogo de vinte-e-um e me informa sobre as regras usadas. Parece interessante e então compro algumas fichas da banca e sento-me para jogar a primeira mão. O *crupiê* embaralha dois baralhos e manda-me cortar. A primeira mão termina e, logicamente, eu ganho.
>
> Diversas mãos são jogadas enquanto eu perco algumas e ganho outras. Finalmente, decido parar um pouco. Neste momento, gostaria de ver como tinha me saído, e então pergunto ao *crupiê* quanto ganhei, ou perdi e durante quanto tempo estive jogando. Se ganhei muito dinheiro ou se tenho poucas fichas posso trocá-las por dinheiro ou comprar outras na banca. Nesse momento decido que o baralho deve ser embaralhado novamente porque a pessoa que está atrás de mim assistindo ao jogo é boa observadora e isso me deixa nervoso. Uma vez sozinho na mesa, contudo, vou deixar o baralho sem embaralhar porque assim tento observar as cartas. Depois de aproximadamente 45 minutos, canso daquilo e resolvo ir embora. O *crupiê* agradece-me por um jogo fascinante, diz-me quanto fiz, paga as fichas que tenho (se é que fiquei com alguma) e me dá boa-noite.

Como se pode ver, pouco disso tudo tem a ver com o jogo em si. Essas interações com o *crupiê* e a banca tornam o jogo, contudo, mais divertido. Se colocarmos tudo isso em nosso sistema, ele se parecerá mais com um agradável jogo de cartas. Cada um que esteja projetando o jogo deve colocar seu gosto pessoal.

Para obtermos essas interações, devemos acrescentar as seguintes funções às de Hoyle:

- MOSTRAR AS INSTRUÇÕES DE COMO JOGAR O JOGO
- CONVERSAR COM A BANCA
- RELATAR COMO O JOGO ESTÁ INDO
- EMBARALHAR AS CARTAS QUANDO FOR PEDIDO
- DEIXAR O JOGO E MOSTRAR COMO O JOGADOR SE SAIU

Todas essas funções parecem ter a mesma importância em nossa sessão na mesa que jogar outra mão teria. A qualquer momento, porém, podemos desejar comprar fichas, ver como estamos nos saindo, reler as instruções ou jogar outra mão. Depois de terminada qualquer dessas funções, podemos escolher novamente entre qualquer dessas opções. O que sugere que talvez a função principal, JOGAR O JOGO, deva ser mudada para ESCOLHER A PRÓXIMA ATIVIDADE. Esta função incluirá as atividades listadas acima, bem como uma rodada do jogo. A única exceção é a demonstração da situação final e a mensagem de adeus no final do jogo. Este é um procedimento de término e será listado sob esta designação no diagrama da estrutura do sistema. Devem todos ter percebido que acrescentamos outra função na qual o embaralhar deve ser feito. Já que é a terceira vez que esta função aparece, EMBARALHAR AS CARTAS tornar-se-á a primeira sub-rotina utilitária; pode ser chamada de qualquer parte do sistema.

Se reconhecermos as coisas desse jeito, as principais funções podem ser representadas assim:

PROCEDIMENTOS DE INICIALIZAÇÃO
- APANHAR DOIS BARALHOS DE CARTAS

ESCOLHER A PRÓXIMA ATIVIDADE
- MOSTRAR AS INSTRUÇÕES DE COMO O JOGO É JOGADO
- CONVERSAR COM A BANCA
- RELATAR COMO O JOGO ESTÁ INDO
- EMBARALHAR AS CARTAS QUANDO SE PEDIR
- JOGAR UMA MÃO DE VINTE-E-UM (incluir todos os detalhes desenvolvidos na Figura 4.3)

PROCEDIMENTOS DE TÉRMINO
- DEIXAR O JOGO E MOSTRAR COMO O JOGADOR SE SAIU

SUB-ROTINAS UTILITÁRIAS
- EMBARALHAR AS CARTAS

Todas essas informações podem ser agora acrescentadas à rede estrutural do sistema. Na rede final, as funções que só tiveram um subelemento (como o PROCEDIMENTO DE INICIALIZAÇÃO) serão geralmente combinadas com o subelemento e receberão outro nome para indicar a coisa que fazem em lugar de serem "subdivididas" em um só item detalhado. Neste estágio, contudo, mostraremos todos os itens que desenvolvemos até aqui. Isso ilustra de maneira mais clara o que fizemos, e permite que se acrescentem funções adicionais de modo mais fácil, se preciso for. Podemos simplificar a estrutura mais adiante depois que tudo tiver sido incluído. Por enquanto, a estrutura do sistema ficará como a Figura 4.4.

Esta é uma boa aproximação da estrutura de nosso sistema e parece conter tudo o que é necessário para descrever todo o jogo. Algumas dessas funções serão subdivididas em unidades mais detalhadas à medida que se avançar com o projeto. Conversar com a banca, por exemplo, terá que incluir a compra de mais

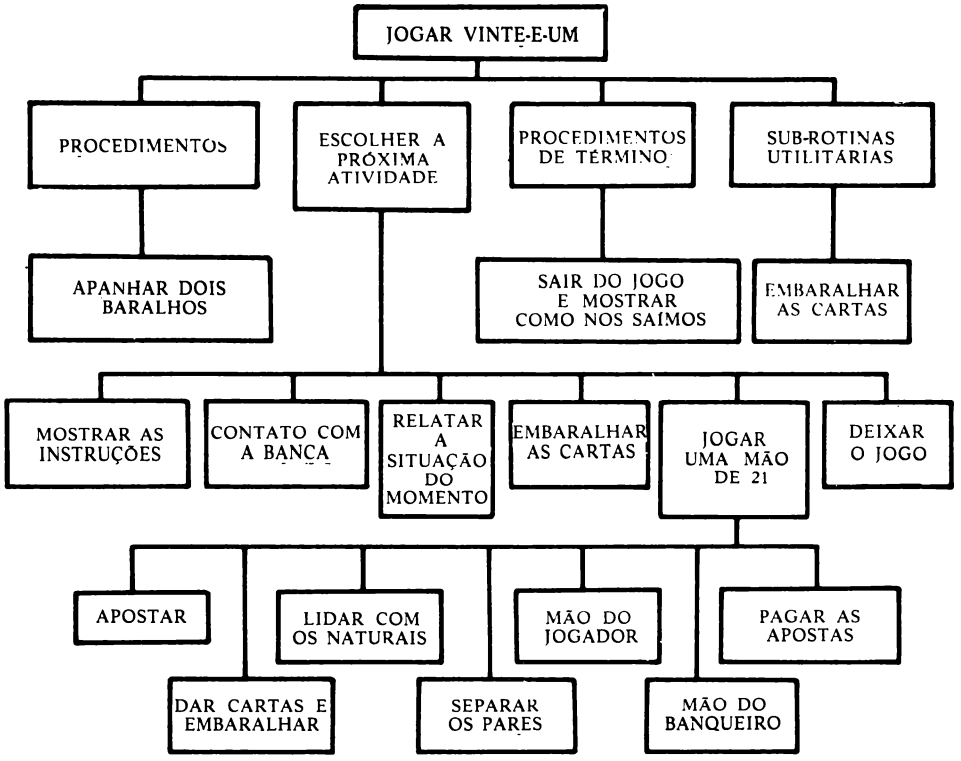



**Figura 4.4** — Diagrama da Estrutura Ampliada

fichas e a troca das restantes por dinheiro. Se já desenvolvemos as principais funções, contudo, é tempo de voltarmos atrás para considerarmos o que fizemos até aqui.

## C. Uma Olhada no que Fizemos

Sempre que se pára para analisar o projeto como um todo, é uma boa idéia ver se ainda se está satisfeito com a maneira como tudo está indo. É mais fácil corrigir as coisas agora do que mais tarde. Há muito poucas regras para este tipo de análise introspectiva. Todavia, quando começamos a nos sentir vagamente insatisfeitos com o projeto, o melhor é parar e ver o que está acontecendo para que se possa encontrar uma solução. No caso do nosso jogo de vinte-e-um, as coisas parecem correr bem e ainda não encontramos nenhum problema significativo. Até aqui, tudo bem. Então, vamos continuar.

Num sistema grande, com diversas funções complexas, há geralmente muito mais a fazer do que um programa possa manipular facilmente. Quando isso acontece, cada módulo é geralmente dividido em diversos programas diferentes,

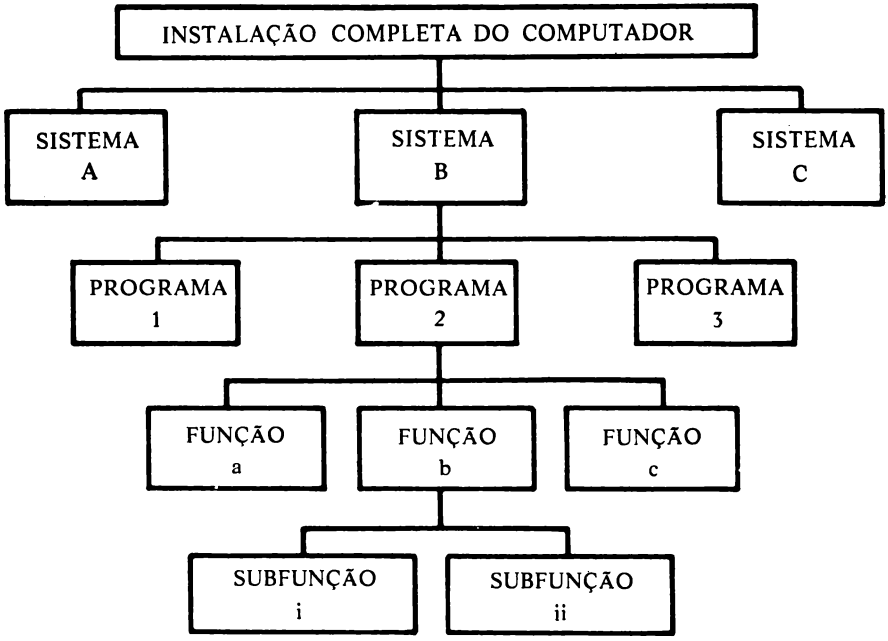



**Figura 4.5** — Hierarquia do Sistema de Computador

cada um simples o bastante para ser compreendido e pequeno o suficiente para caber na memória do computador. Tais programas, de um modo geral, têm acesso às informações através de arquivos de dados em dispositivos externos de armazenamento. Cada programa do sistema cuida de uma tarefa, do mesmo modo que cada módulo do programa cuida de sua própria função. Para aplicações muito grandes, como no caso de sistemas de contabilidade computadorizada de grandes empresas, sistemas inteiros de programas trabalham uns com os outros, cada sistema realizando uma única mas complexa tarefa. A hierarquia estrutural de uma instalação de computador completa pode parecer com essa da Figura 4.5.

O número de sistemas, programas, funções e subfunções de qualquer parte do sistema e o número de níveis adicionais de subfunções em qualquer ramificação dependerão, logicamente, dos detalhes da instalação. Mesmo um sistema pequeno, como a utilização da computação para fins pessoais, pode conter todos esses elementos.*

Uma vez definidas as principais funções, podemos decidir de quantos programas nosso sistema necessitará. Nosso "sistema" Vinte-e-Um consistirá de apenas um programa.

---

* As coisas funcionarão melhor se tudo for desenvolvido de cima para baixo, logicamente, mas em geral isso é muito difícil de se fazer porque os programas ou sistemas são geralmente acrescidos aos poucos, à medida que a pessoa se torna interessada neles após algum tempo decorrido.

## D. Uma Análise do Processo que Usamos

Enquanto paramos para fazer uma revisão do que fizemos, podemos aproveitar para fazer algumas observações a respeito de como o método top-down, que escolhemos, funcionou e o que ganhamos ao usá-lo.

A maior vantagem desta abordagem é que não se perde a noção do objetivo geral ao se cuidar dos detalhes. O que é um contraste acentuado com a forma pela qual muitos projetos pequenos são desenvolvidos. Freqüentemente quem os projeta envolve-se em partes específicas do sistema sem saber como (ou se) essas partes serão usadas realmente. Por exemplo, teria sido muito interessante no início do nosso projeto escrever um pequeno programa para embaralhar e distribuir as cartas. Podíamos não ter percebido, então, que usaríamos dois baralhos que poderiam ser embaralhados ou distribuídos independentemente. O programa original teria que ser, sem dúvida alguma, reescrito para que essas mudanças fossem introduzidas. Do modo como agora está, podemos trabalhar em qualquer parte sem nos preocuparmos se ela vai se encaixar no todo ou não.

Uma segunda vantagem do projeto top-down é que a estrutura do sistema é desenvolvida em torno de funções que devem ser executadas. Se algumas dessas funções necessitarem de modificações em algum momento, será fácil ver onde as mudanças devem ser feitas. Será fácil também mudar uma função sem mudar qualquer outra coisa mais.

A terceira vantagem é que já fizemos um trabalho de documentação sobre como o sistema irá operar melhor do que muitos programadores fazem com projetos inteiros. Quando for preciso fazer mudanças — e sempre chega esse momento — é muito importante ter um guia escrito sobre como o sistema opera, especialmente no caso de se desenvolverem diversos projetos diferentes e não se poder guardar na cabeça todos os detalhes. À medida que continuarmos desenvolvendo nosso jogo de vinte-e-um, continuaremos, igualmente, acrescentando sua documentação.

Há duas pequenas desvantagens neste método de desenvolvimento. A primeira é apenas psicológica: é realmente muito mais divertido começar logo a programar do que perder algum tempo definindo como as coisas devem ser feitas. Mas saibam que sua paciência será plenamente recompensada. Haverá menor necessidade de correções e muito menos frustração devido a mudanças no projeto. Não é gratificante deixar de lado algo que já foi feito, e os programas que são muito modificados geralmente não operam tão bem quanto aqueles que operam como originalmente concebidos. É importante ter perseverança nas etapas iniciais do projeto, pois quando se chegar a escrever o programa será mais divertido ainda.

O outro problema tem a ver com o fato de que estamos desenvolvendo o projeto pensando no programador e não no computador. No início da programação, as máquinas eram pequenas e caras. O tempo de programação, contudo, era relativamente barato, de modo que os programadores gastavam muito esforço desenvolvendo programas eficientes que fizessem o menor número possível de

exigências da máquina. Hoje em dia as coisas são bem diferentes. Até mesmo os pequenos computadores pessoais podem ter memórias razoavelmente grandes e trabalham relativamente depressa. O tempo disponível para programação, porém, tornou-se muito caro.

A programação estruturada top-down foi elaborada para facilitar o desenvolvimento de programas para o programador às custas do computador. Isso significa que, embora nosso programa de Vinte-e-Um seja fácil de ser entendido e escrito, não pode ser tão eficiente como um que seja escrito para fazer o melhor uso possível do equipamento. Para a maior parte dos programas isso não constitui um grande problema. Mas deve-se ficar atento para o fato de que não estamos desenvolvendo, necessariamente, o mais eficiente programa do ponto de vista do computador.

Devemos mencionar aqui que podem haver exceções à direção do projeto top-down. Em alguns projetos, pode ser necessário experimentar algumas funções detalhadas específicas para ver se o projeto é mesmo viável. O sistema será então desenvolvido em direção a esses módulos específicos. Trabalhar de cima e de baixo em direção ao centro geralmente resulta em ter de alterar tanto a parte superior da estrutura do sistema quanto as rotinas detalhadas antes que tudo funcione conjuntamente. Isso deve ser evitado, se possível, mas às vezes torna-se necessário.

Agora que sabemos o que o sistema basicamente vai fazer, é hora de considerar as informações requeridas pelo sistema. Isso incluirá os dados reunidos do mundo externo como entrada, os dados mantidos dentro do sistema para uso próprio e os dados a serem mostrados ao mundo externo como saída.

## III. DEFININDO O FLUXO DE DADOS NO SISTEMA

Deve ser lembrado que no início deste capítulo decidimos desenvolver primeiro as funções do sistema e depois tratar dos requisitos dos dados. Geralmente, utilizam-se funções e dados durante o desenvolvimento do projeto, indo e vindo à medida da necessidade. A seguir, vamos ver os dados requeridos pelo programa de Vinte-e-Um.

A análise inicial do fluxo de informação de um sistema começa pela definição de tudo que desejamos que o sistema produza como saída. A partir daí podemos definir todos os dados necessários para obtermos a saída e as funções de que precisamos para desenvolvermos isto. Finalmente, definimos qual informação é necessária como entrada e qual pode ser incluída no programa como consulta. No caso do programa de Vinte-e-Um, já definimos a maior parte disso com as regras oficiais do jogo e com os diálogos baseados numa sessão imaginária do jogo. É tempo agora de vermos a saída do sistema de um modo mais detalhado.

Podemos começar trabalhando com os requisitos de informação do sistema, do mesmo modo como começamos a trabalhar com os requisitos funcionais do sistema — começar a escrever tudo que nos vier à mente. Para manter-se alguma

ordem durante este processo, podemos escrever cada função procurando ver as informações que devem ser mostradas. A lista a seguir é uma primeira tentativa de identificar a saída do sistema:

Começar o jogo (iniciação)
- Mostrar a tela recepcionando o jogador.

Selecionar a próxima atividade
- Mostrar as escolhas e solicitar a próxima coisa a fazer.

Mostrar instruções
- Imprimir tela-padrão (ou telas) com as regras.

Contato com a banca
- Dizer se estamos comprando ou trocando as fichas por dinheiro.
- Mostrar o total que ganhamos ou perdemos.
- Indicar quando chegar o limite do jogador ou quando a banca quebrar.
- Solicitar o número de fichas para comprar ou trocar por dinheiro.

Relatar o progresso do jogo.
- Mostrar o total ganho ou perdido.
- Mostrar o número de rodadas jogadas.
- Mostrar o número de rodadas ganhas ou perdidas.
- Mostrar a percentagem de rodadas ganhas.
- Mostrar a média ganha ou perdida em cada rodada.

Embaralhar as cartas.
- Imprimir uma mensagem simples dizendo que as cartas estão sendo embaralhadas.

Jogar uma rodada de vinte-e-um.
- Pedir que as apostas sejam colocadas.
- Indicar quando não houver dinheiro suficiente para cobrir uma aposta ou quando uma aposta estiver abaixo do mínimo ou acima do máximo
- As cartas devem ser mostradas quando distribuídas, algumas viradas para cima e outras para baixo.
- Indicar que o banqueiro tem um natural.
- Perguntar se deve ser separado algum par.
- Perguntar se os jogadores querem outra carta.
- Mostrar o jogo do banqueiro.
- Mostrar o total perdido ou ganho na mão atual.

Deixar o jogo (término).
- Mostrar o relatório de situação do jogo.
- Mostrar a mensagem de adeus.

A partir desta lista podemos identificar os seguintes itens de dados que devem estar disponíveis ao programa.

Começar o jogo.
- Mensagem de "Bem-vindo ao jogo".

Selecionar a próxima atividade.
- Mensagem "As seguintes escolhas estão disponíveis."
- Perguntar "e agora?"

Mostrar instruções.
- Mensagem das regras.

Contato com a banca.
- Pergunta "Comprar ou trocar por dinheiro?"
- Quantia total de dinheiro perdida ou ganha.
- Limite de crédito do jogador.
- Limite de crédito do banqueiro.
- Pergunta "Quanto para comprar ou trocar?"

Relatar o progresso do jogo.
- Quantia total de dinheiro ganha ou perdida.
- Número total de mãos jogadas.
- Número total de mãos ganhas ou perdidas.
- Percentagem de mãos ganhas ou perdidas.
- Média de ganhos e perdas por mão.

Embaralhar as cartas.
- Mensagem "As cartas foram embaralhadas."

Jogar uma rodada de vinte-e-um.
- Pergunta "Quanto apostar?"
- Valor das fichas do jogador.
- Aposta mínima.
- Aposta máxima.
- Cartas — o valor e o naipe devem ser mostrados.
  - O baralho restante.
  - O jogo do banqueiro.
  - O jogo do jogador.
  - a segunda mão do jogador (separada).
- O verso da primeira carta da banca.
- Mensagem "O banqueiro tem um natural."
- Mensagem "Você ganhou."
- Mensagem "Você perdeu."
- Pergunta "Quer separar seu par?"
- Pergunta "Deseja outra carta?"
- Total ganho ou perdido durante esta mão.

Deixar o jogo
- Mensagem de "Adeus".

Infelizmente, esta lista é organizada por funções do sistema em lugar de o ser por qualquer critério derivado dos itens de dados. Teríamos maior conhecimento sobre a informação que usaríamos se esta lista fosse agrupada por tipos similares de informação. Ao se olhar esta lista, os itens de dados parecem se dividir em quatro grupos gerais.

O primeiro grupo inclui qualquer mensagem que faça uma pergunta ao jogador, tal como:

"O QUE VOCÊ DESEJA FAZER A SEGUIR?"

# CONHEÇA OS LIVROS DA

# EDITORA CAMPUS LTDA.

## SIM • Estou interessado

Desejo receber informações dos lançamentos da Editora Campus nas seguintes áreas*:

| | | |
|---|---|---|
| ☐☐INFORMÁTICA | ☐☐ESTATÍSTICA | ☐☐ECONOMIA |
| ☐☐MATEMÁTICA | ☐☐FÍSICA | ☐☐ADMINISTRAÇÃO |
| ☐☐ENGª ELETRÔNICA | ☐☐QUÍMICA | ☐☐CIÊNCIAS SOCIAIS |
| ☐☐ENGª MECÂNICA | ☐☐PSICOLOGIA | ☐☐ARTE |
| ☐☐ENGª CIVIL | ☐☐PSICANÁLISE | ☐☐OUTRAS |

* *Favor indicar com dois X, no máximo, até duas áreas de interesse prioritário e com um X as áreas de interesse secundário*

Livro Adquirido.....

Livraria..... Estado.....

Nome.....

Endereço .....

Estado..... Cidade..... CEP. .....

Profissão ..... Função .....

Instituição .....

**IMPORTANTE: É imprescindível a indicação de Livro Adquirido e Profissão.**

**NOSSOS LIVROS ENCONTRAM-SE**
**EM TODAS AS BOAS LIVRARIAS**

DOBRE AQUI

SELO

**Editora Campus Ltda.**

**LIVROS CIENTÍFICOS E TÉCNICOS**
CAIXA POSTAL 13007 CEP 20262
RIO DE JANEIRO RJ BRASIL
Endereço Telegráfico: CAMPUSRIO

COLE

# COMO CONSTRUIR UM PROGRAMA

Embora já existam no mercado inúmeros programas prontos, facilmente acessíveis, pelo menos quatro motivos diferentes podem levá-lo a querer escrever seus próprios programas:

- é possível que ninguém tenha ainda criado um programa para aquela determinada tarefa que você deseja realizar
- mesmo havendo este programa, talvez ele não seja exatamente o que você imaginava
- é mais fácil compreender e melhor usar o seu computador quando você pode controlá-lo com seus próprios programas
- criar um programa bem escrito pode ser um projeto agradável e gratificante

Destinado ao leitor sem experiência em computação, este livro vai torná-lo capaz de desenvolver seus próprios programas a partir de uma idéia original, passando pelas fases do projeto e da implementação, até o momento de rodá-lo. Assim, o processo "top-down" (descendente) de programação estruturada é detalhado passo a passo, as técnicas e recursos são revelados e dicas e sugestões sobre erros e "grilos" são apresentadas, juntamente com valiosas informações sobre os processos de verificação.

ISBN 85-7001-222-5
(Edição original: ISBN 0-88056-068-1 dilithium Press, Oregon, USA.)