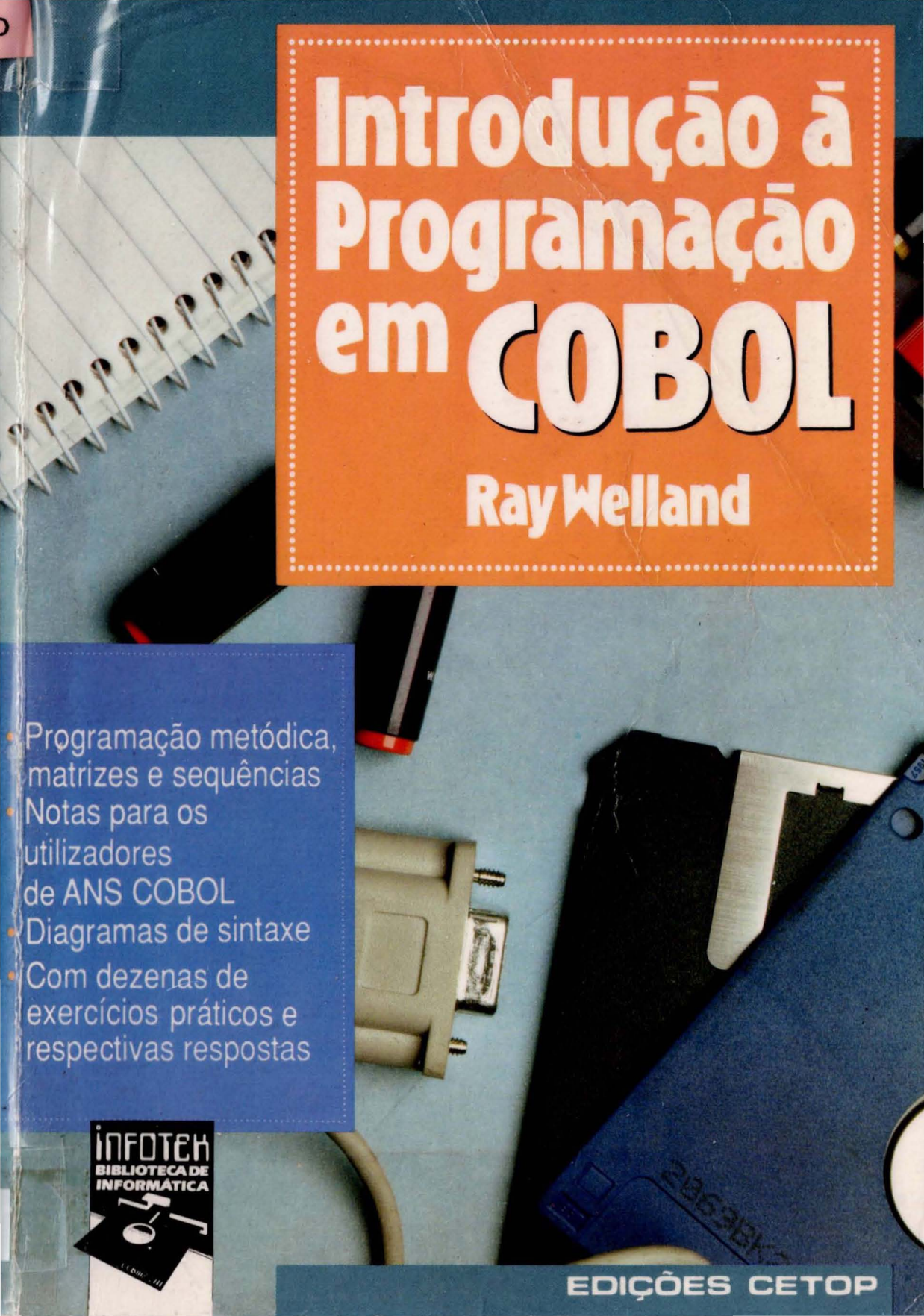
Introdução à Programação em COBOL

Ray Welland



- Programação metódica, matrizes e sequências
- Notas para os utilizadores de ANS COBOL
- Diagramas de sintaxe
- Com dezenas de exercícios práticos e respectivas respostas

INFOTEK
BIBLIOTECA DE INFORMÁTICA

EDIÇÕES CETOP

INTRODUÇÃO
À PROGRAMAÇÃO
EM COBOL

Obras publicadas nesta colecção:

1 — *Folhas de Cálculo,* D. Harrison e J. W. Yu
2 — *Introdução à Programação em Pascal,* W. Findlay e D.A. Watt
3 — *Introdução à Programação em Cobol,* Ray Welland

RAY WELLAND

INTRODUÇÃO À PROGRAMAÇÃO EM COBOL

EDIÇÕES CETOP

Título original: *Methodical Programming in Cobol*

Tradução e revisão técnica de Nuno de Azevedo

Capa de Luís Faria

Fotografia da capa de Luísa Flores

© Ray Welland, 1983

Direitos reservados por
EDIÇÕES CETOP — Centro de Ensino
Técnico e Profissional à Distância, Lda.

Nenhuma parte desta publicação pode ser reproduzida ou transmitida por qualquer forma ou por qualquer processo, electrónico, mecânico ou fotográfico, incluindo fotocópia, xerocópia ou gravação, sem autorização prévia e escrita do editor. Exceptua-se naturalmente a transcrição de pequenos textos ou passagens para apresentação ou crítica do livro. Esta excepção não deve de modo nenhum ser interpretada como sendo extensiva à transcrição de textos em recolhas antológicas ou similares donde resulte prejuízo para o interesse pela obra. Os transgressores são passíveis de procedimento judicial

Execução técnica:
Gráfica Europam, Lda.
Mem Martins

Edição nº 06 29 003
ISBN 972-641-042-8
Depósito Legal nº 52090/92

Editor: Tito Lyon de Castro

EDIÇÕES CETOP
Apartado 7
2726 Mem Martins Codex
Portugal

Índice

	Pág.
Prefácio	7
Capítulo 1 — *Introdução*	11
1.1— História e objectivos do COBOL	11
1.2 — O computador de COBOL	12
1.3 — O computador real: «hardware» e «software»	14
Capítulo 2 — *A linguagem de desenho de programas*	16
2.1— Operações básicas	16
2.2 — Selecção e repetição	18
2.3 — Programas simples	20
2.4 — Refinamento	24
2.5 — Estudo de caso	26
Exercícios 2	30
Capítulo 3 — *O programa de COBOL*	32
3.1 — Estrutura de programas	32
3.2 — Palavras reservadas e nomes definidos pelo utilizador	33
3.3 — A estrutura de dados	34
3.4 — Tipos de dados	35
Exercícios 3	37
Capítulo 4 — *Declaração de procedimento*	38
4.1 — A estrutura da Procedure Division	38
4.2 — Operações básicas em COBOL	38
4.3 — Selecção e repetição	42
4.4 — Estudo de caso	47
Exercícios 4	53
Capítulo 5 — *Mais «input» e «output»*	55
5.1 — «Input» e armazenamento de valores decimais	55
5.2 — Edição e registos de «output»	57
5.3 — Números com sinal	59
5.4 — Cabeçalhos no «output»	60
5.5 — Registos múltiplos de «output»	64
5.6 — Estudo de caso	67
Exercícios 5	73
Capítulo 6 — *Desenvolvimento de algoritmos*	75
6.1 — Validação de dados	75
6.2 — Condições de dados de classe e redefinição	76
6.3 — Nomes-condição	79
6.4 — Declarações IF encadeadas	80
6.5 — Condições compostas	83

6.6 — Campos de dados boolianos 88
6.7 — Estudo de caso 90
Exercícios 6 103

Capítulo 7 — *Erros e testes* 107
7.1 — Erros de sintaxe 107
7.2 — Verificação («debugging») 110
7.3 — Testes 113
7.4 — Estudo de caso 118
Exercícios 7 121

Capítulo 8 — *EStrutura de programa* 123
8.1 — Identification Division 123
8.2 — Environment (meio) Division 124
8.3 — Data Division 126
8.4 — Procedure Division 129
8.5 — Linhas e páginas 135
8.6 — Registos de «input» de tipos diferentes 137
8.7 — Estudo de caso 142
Exercícios 8 156

Capítulo 9 — *Ficheiros múltiplos* 159
9.1 — Ficheiros múltiplos de «input» e de «output» 159
9.2 — Actualização de ficheiros sequenciais 164
9.3 — Primeira tentativa de um algorítmo de actualização 168
9.4 — O algorítmo de «linha de balanço» 176
9.5 — Estudo de caso 181
Exercícios 9 198

Capítulo 10 — *Tabelas* 201
10.1 — Dados respetitivos num registo de «input» 201
10.2 — Tabelas na Working-Storage 208
10.3 — Itens de grupo repetidos 214
10.4 — Pesquisando uma tabela 218
10.5 — Tabelas multidimensionais 223
10.6 — Estudo de caso 225
Exercícios 10 234

Capítulo 11 — *Características adicionais do COBOL* 237
11.1 — A instrução GO TO 237
11.2 — Comunicação Interprogramas 238
11.3 — Biblioteca de «source» (fonte) e directivas de cópia 241
11.4 — Ficheiros realtivos 242
11.5 — Ficheiros indexados 245

Apêndice 1 — *O esqueleto de programa de COBOL* 249

Apêndice 2 — *Formato do programa de COBOL* 252

Apêndice 3 — *Palavras reservadas* 255

Apêndice 4 — *Sumário da sintaxe de COBOL* 258

Apêndice 5 — *Notas para utilizadores de ANS COBOL de baixo nível* 267

Respostas aos exercícios seleccionados 275

PREFÁCIO

Na altura em que é escrito este livro, a linguagem de programação comercial mais largamente utilizada é o COBOL, que existe há mais de vinte anos. Similarmente, a linguagem de programação científica mais largamente utilizada é o FORTRAN, formulado pela primeira vez em 1957. Tem sido várias vezes reivindicada a existência de melhores linguagens para ambos os campos, mas a dificuldade em introduzir uma nova linguagem reside no enorme investimento em software *já existente, escrito em COBOL e FORTRAN. Assim, parece que estas duas linguagens ainda continuarão connosco por bastante tempo.*

O objectivo deste livro é tentar combinar as novas técnicas da programação estruturada com a existente linguagem COBOL para mostrar o quão bem os programas podem ser construídos.

Os principais alvos deste livro são os alunos de computação ou processamento de dados que possam ou não ter tido uma experiência anterior em programação. As ideias foram desenvolvidas durante o ensino de COBOL como primeira linguagem de programação para alunos de Contabilidade e o ensino de COBOL a alunos de Ciência de Computação já habituados ao Pascal.

A abordagem usada neste livro consiste em separar as tarefas de desenho de programa e de codificação. Os programas são desenhados através de uma linguagem de desenho de programas abstracta que pode então ser traduzida para o COBOL através de um conjunto definido de transformações. Um dos atractivos desta abordagem é o facto de os alunos poderem aprender a programar sem serem esmagados pela massa de pormenores envolvidos mesmo num simples programa de COBOL. Será também possível desenhar um conjunto de transformações diferentes do desenho de programa para outra linguagem de programação.

Os desenhos de programa podem ser verificados pelo leitor, ou demonstrador, e discutidos antes que a codificação comece (e também marcados, se necessário). Isto significa que muitos erros lógicos podem ser descobertos antes que o programa seja de facto codificado em COBOL.

Como usar o livro

Este livro está estruturado de modo que o aluno seja capaz de desenhar programas bastante substanciais no fim do capítulo 2. Isto permitirá a realização de trabalho prático útil, enquanto elementos fundamentais do COBOL são explicados nos capítulos 3 e 4. No fim do capítulo 4, o aluno será capaz de escrever e pôr a correr programas simples de COBOL.

Os dois capítulos seguintes apresentam mais características da linguagem que dizem respeito ao modo de manusear o input[1] *e o output*[2] *e à expressão de algoritmos mais complexos. No fim do capítulo 6, todos os elementos da linguagem requeridos para escrever programas de COBOL bastante complexos terão sido apresentados ao aluno. Os capítulos 7 e 8 tencionam dar um «espaço para respirar», enquanto o aluno consolida o seu conhecimento através de uma prática de programação em COBOL.*

---

[1] Dados de entrada. *(N. do T.)*

[2] Dados de saída. *(N. do T.)*

A partir do capítulo 9, mais características da linguagem são apresentadas e o repertório do aluno, em técnicas de programação, é francamente melhorado.

Os apêndices destinam-se a dar resumos de várias características do COBOL, para uma rápida referência durante a escrita de programas.

Exercícios

A programação é essencialmente uma matéria prática, e, para tirar uma vantagem completa deste livro, o aluno deve realmente desenhar e correr alguns programas. A programação é como nadar — podem ler-se todos os livros que houver, mas até «se mergulhar» não se começa verdadeiramente a aprender.

Os exercícios dividem-se em duas categorias: exercícios de lápis e papel, para seguir de perto pontos contidos no texto, e exercícios de programação, que envolvem o desenho, escrita e execução de programas de computador completos. Recomenda-se ao aluno que faça todos os exercícios de lápis e papel e, pelo menos, um dos exercícios de programação no fim de cada capítulo.

No fim do livro apresentam-se sugestões para a solução dos problemas de lápis e papel.

A linguagem COBOL

A linguagem usada neste livro é a definida no American National Standard for COBOL em 1974, geralmente designado por ANS COBOL 74. A definição de ANS COBOL 74 é modular e dentro de cada módulo existem níveis de implementação. Portanto, o que é legitimamente descrito como um compilador ANS COBOL 74 pode não ter todas as características introduzidas neste livro. Para auxílio de leitores que usem compiladores de ANS COBOL de baixo nível, são inseridas notas no texto onde possam surgir problemas, e o apêndice 5 apresenta algumas sugestões para construções alternativas para uma programação metódica.

Agradecimentos

Gostaria de agradecer a Charles Clarke e Roy McAlister, da Divisão de Serviços de Gestão, Universidade de Strathclyde, pelo seu auxílio na utilização do sistema local para teste dos programas de COBOL contidos neste livro. Gostaria também de agradecer a influência de dois ex-colegas, Bill Findlay e David Watt, da Universidade de Glásgua, que muito me ensinaram acerca da programação estruturada.

Este livro deve muito também a outro ex-colega, John Jeacocke, da Universidade de Glásgua, que me encorajou a utilizar o estilo de ensino defendido neste livro, me ajudou a formular a linguagem de desenho e teceu muitas críticas úteis de vários planos do livro. Finalmente, gostaria de agradecer ao editorconsultor, David Hatter, pelas suas críticas construtivas, e a Alfred Waller, da Pitmans, pelo apoio prestado durante a elaboração deste livro.

Janeiro de 1983.

RAY WELLAND
Universidade de Strathclyde

ESCLARECIMENTOS

O COBOL é uma linguagem industrial e não é propriedade de nenhuma companhia ou de nenhuma organização ou grupo de organizações.

Nenhuma garantia expressa ou implícita foi feita por nenhum subscritor ou pelo comité da Linguagem de Programação CODASYL quanto à precisão e funcionamento do sistema e linguagem de programação. Além disso, nenhuma responsabilidade foi assumida por qualquer subscritor, ou pelo comité, relativamente a este facto.

Os autores e detentores de direitos de autor do material abrangido por esses direitos neste livro são: FlowMatic (marca registada da Sperry–Rand Corporation), Programação para o Univac I e II, Data Automation Systems, abrangidos por direitos de autor, 1958, 1959, Sperry–Rand Corporation; Tradutor Comercial IBM modelo n.º F28-8013, direitos de autor da IBM, 1958; Fact, DSI 27A5260-2760, direitos de autor da Minneapolis–Honeywell, 1960.

Autorizaram especialmente o uso deste material, no todo ou em parte, nas especificações de COBOL. Tal autorização abrange a reprodução e uso de especificações de COBOL em manuais de programação ou publicações similares.

NOTA DO EDITOR PORTUGUÊS

A palavra inglesa *penny* tem como plural *pence*. O *penny* é a centésima parte da libra (unidade monetária).

Na tradução desta obra utilizou-se a palavra portuguesa correspondente a *pence, pénis* (singular: péni).

Chama-se a atenção para este facto, a fim de que o leitor menos avisado não suponha ter havido erro de tradução...

Ver, por exemplo, p. 31, linhas 6, 8 e 9, p. 69, linha 39, etc.

1. IINTRODUÇÃO

Este capítulo inclui uma história muito breve do COBOL e um exame de algumas das bases do computador que são necessárias a este livro. No entanto, não se pretende que este livro seja lido isoladamente. Pressões de espaço fazem que ele não possa cobrir conceitos fundamentais de computação com alguma profundidade, embora faça jus ao seu principal tema de programação metódica em COBOL. Assim, presumir-se-á que os leitores estão familiarizados com os conceitos básicos da computação, através da frequência de aulas regulares, de leitura de base ou através do uso do seu próprio computador pessoal.

1.1 — HISTÓRIA E OBJECTIVOS DO COBOL

Nos finais da década de 1950 reconheceu-se que as linguagens de programação existentes eram inadequadas para a construção de grandes sistemas de processamento de dados comerciais. Naquela altura, a maior parte da programação era executada em linguagens *assembly*[1], o que significava que programas e pessoal qualificado estavam ligados a um tipo particular de computador. Assim, o Departamento Americano de Defesa, um dos maiores utilizadores de computadores para controlo de inventários e contabilidade, reuniu um grupo de especialistas com o objectivo de criar uma nova linguagem de programação de alto nível, adequada a aplicações comerciais de computação. Os objectivos que foram dados a este grupo, designado por Comité CODASYL, para o desenho da linguagem, são os seguintes:

a) Especificar a liguagem independentemente de qualquer fabricante ou modelo de computador, possibilitando várias alternativas e enunciada numa notação em inglês e numa forma narrativa;

b) A linguagem deveria ser extensível, de modo que pudesse ser executada em futuras séries de máquinas;

c) Deveria ser fácil de aprender, de modo que programadores relativamente inexperientes pudessem prestar uma contribuição significativa;

d) Os programas escritos na linguagem deveriam ser «autodocumentais» e «legíveis», quer por empresários, quer por pessoal não técnico.

O Comité propôs a linguagem actualmente conhecida por COBOL (COmmon Business Oriented Language), que tem vindo a ser revista regularmente, desde o seu início, em 1960. A linguagem é revista regularmente e novas propostas são publicadas no *COBOL Journal of Development.* Vários fabricantes prestaram também os seus próprios contributos às linguagens, dando origem a uma vasta série de «dialectos» da linguagem. A fim de manter a portabilidade da linguagem, uma das suas mais importantes vantagens, publicou-se, em 1968, um *American National Standard for COBOL* e um novo *standard,* em 1974. Este livro baseia-se no *standard*

---

[1] Linguagem-máquina. *(N. do T.)*

de 1974, geralmente designado por *ANS COBOL 74,* dado que esta é, vulgarmente, a versão de COBOL mais largamente utilizada. Espera-se que um novo *standard* seja acordado em 1983, mas este levará algum tempo até chegar a ser inteiramente implementado e aceite.

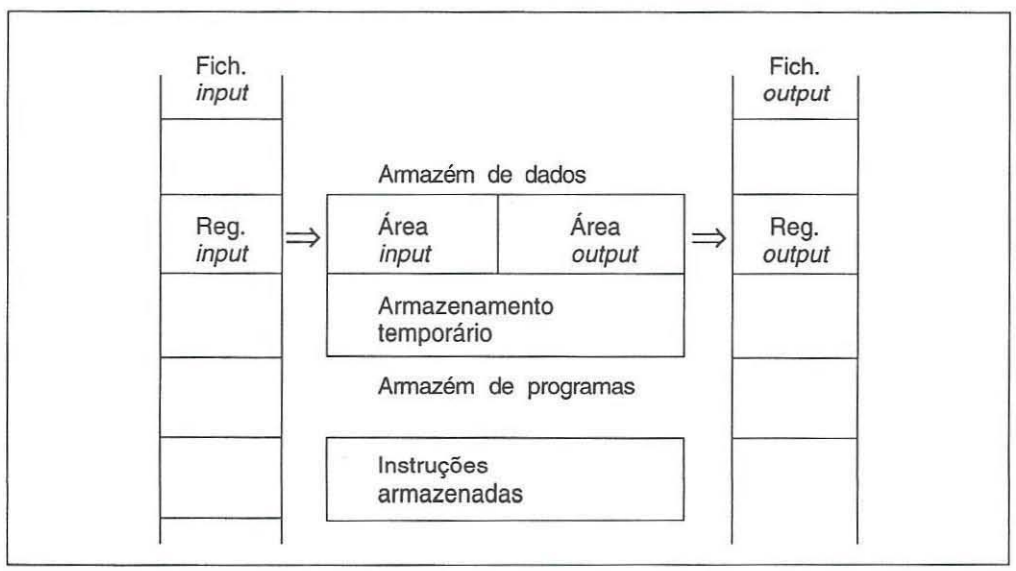
O COBOL é uma linguagem de programação muito «rica» e poderosa, tem uma enorme variedade de características destinadas a resolver os problemas mais complexos no processamento de dados comerciais. Esta fartura de facilidades pode constituir um perigo para o aluno, que se pode deixar influenciar por elas. Este livro utiliza apenas um segmento mínimo útil de iniciação ao COBOL, para que o programador em formação possa compreender os princípios básicos antes de aprender a usar aspectos mais avançados da linguagem.

1.2 — O COMPUTADOR DE COBOL

Ao desenhar uma linguagem de programação, é necessário ter um modelo conceptual do computador a usar, que é designado por máquina abstracta. De acordo com os propósitos deste livro, iremos supor que o modelo mostrado na figura 1.1 é a primeira aproximação à máquina abstracta de COBOL.

Os ficheiros de *input* e *output* dividem-se em registos. Cada registo de um ficheiro é de formato similar e, geralmente, contém dados acerca de um elemento de um grupo de «coisas» relacionadas. Por exemplo, um ficheiro poderia conter informações acerca dos empregados de uma companhia; cada empregado teria um registo no ficheiro e cada registo conteria dados semelhantes: número de empregado, nome, morada, número de apólice, salário, etc.

Acede-se a um ficheiro de tal modo que apenas um registo do ficheiro fica disponível para o programa em qualquer momento determinado, o registo actual. Inicialmente, convencionar-se-á que há um ficheiro de *input* e um ficheiro de *output*. O registo actual de *input* é retido na área de «input» e escrevem-se registos para o ficheiro de *output* via área de «output». Num programa que faça algo útil é necessário que haja um armazenamento interno para, por exemplo, valores intermédios em cálculo. Os valores retidos num tal armazenamento temporário existem apenas durante o tempo de duração do programa, e, se uma cópia dos valores for requerida, deverá ser enviada para o ficheiro de *output*. Assim, no modelo simples do computador de COBOL,



o armazém de dados divide–se em três partes: a área de *input,* a área de *output* e o armazenamento temporário.

Cada uma das áreas do armazém de dados divide–se numa hierarquia de unidades de armazenamento chamadas *registos, itens de grupo* e *itens elementares.* No nível mais baixo, o item elementar é uma posição fixa que pode reter um valor que pode ser quer um número, quer uma série de caracteres. O processamento de dados faz–se usando estes valores armazenados em operações aritméticas, transferindo–os de posição para posição e tomando decisões simples, baseadas nos valores armazenados em posições específicas.

A simples máquina precisará de um número de operações básicas que podem então ser combinadas para criar programas. Um programa é uma sequência de instruções que, na ausência de instruções que mudem o «fluxo de controlo», serão executadas na ordem em que aparecerem no armazém de programas. Obviamente, será necessário haver uma operação para que se obtenha um registo do ficheiro de *input,* tornando–o disponível na área de *input,* e uma operação complementar para colocar um registo, que tenha sido reconstruído no armazém de dados, no ficheiro de *output.* Será também necessário uma operação de transferência que permita que os dados sejam copiados de uma posição de armazenamento para outra e operações aritméticas básicas: somar, subtrair, multiplicar e dividir, que funcionem sobre os dados armazenados, para criar novos valores.

Seria possível escrever um programa simples de computador que consistisse apenas numa sequência de operações básicas, executadas uma após outra, da primeira operacão à última. Um tal programa simples de computador consistiria em três partes: início, processamento e fim. Na maioria dos programas, a parte de processamento será mais complexa do que a simples sequência de instruções, mas estas três partes básicas existirão sempre.

O início definirá um estado inicial para o computador, o processamento mudará o estado da máquina, transformando de algum modo os dados, e o fim fará uma arrumação antes de o programa acabar. Por exemplo, o início definirá o estado inicial dos ficheiros de *input* e de *output* e os valores iniciais no armazém de dados. A qualquer altura determinada, durante o processamento, pode ser preciso descobrir o estado actual da máquina, isto é, há mais alguns registos de *input?* Numa máquina simples, o fim poderá consistir em libertar os ficheiros de *input* e *output* e em imprimir uma mensagem acerca do estado final da máquina.

A maioria dos programas úteis, principalmente no processamento de dados comerciais, envolverá repetições de sequências de instruções. Por exemplo, voltando ao ficheiro «empregado», seria possível estabelecer uma sequência de operações, designada por algoritmo, para calcular o ordenado de um empregado, dados certos elementos, tais como salário, número de contribuinte, estatuto de aposentação, etc. Este algoritmo pode ser repetido para cada registo de empregado do ficheiro. Portanto, além das operações básicas acima mencionadas, será requerida uma operação especial que permita uma repetição controlada de sequências de operações. Isto significa que se terá de incluir na linguagem um método que reconheça condições que vão surgindo. Condições são questões formalizadas que podem ser respondidas, e o resultado devolvido será ou verdadeiro ou falso, um valor «booliano».

Muitos algoritmos envolverão a escolha de rumos alternativos de acção que dependem dos valores dos dados. Voltando ao exemplo do empregado, poderiam existir dois esquemas válidos de aposentação para empregados, requerendo cada um uma rotina de dedução de cálculos especial. O tipo de esquema de aposentação poderia ser distinguido através de um código especial, «A» ou «B», no registo de empregado, que precisaria de ser reconhecido pelo programa e pela rotina apropriada utilizada. Portanto, além de repetição, requer–se uma operação de selecção.

Assim, resumindo, a linguagem de desenho básica usada neste livro terá as seguintes operações:

a) Registo de *input/output:* receber e colocar;
b) Transferência de dados, dentro do armazém de dados;
c) Cálculos aritméticos simples;
d) Repetição controlada;
e) Selecção.

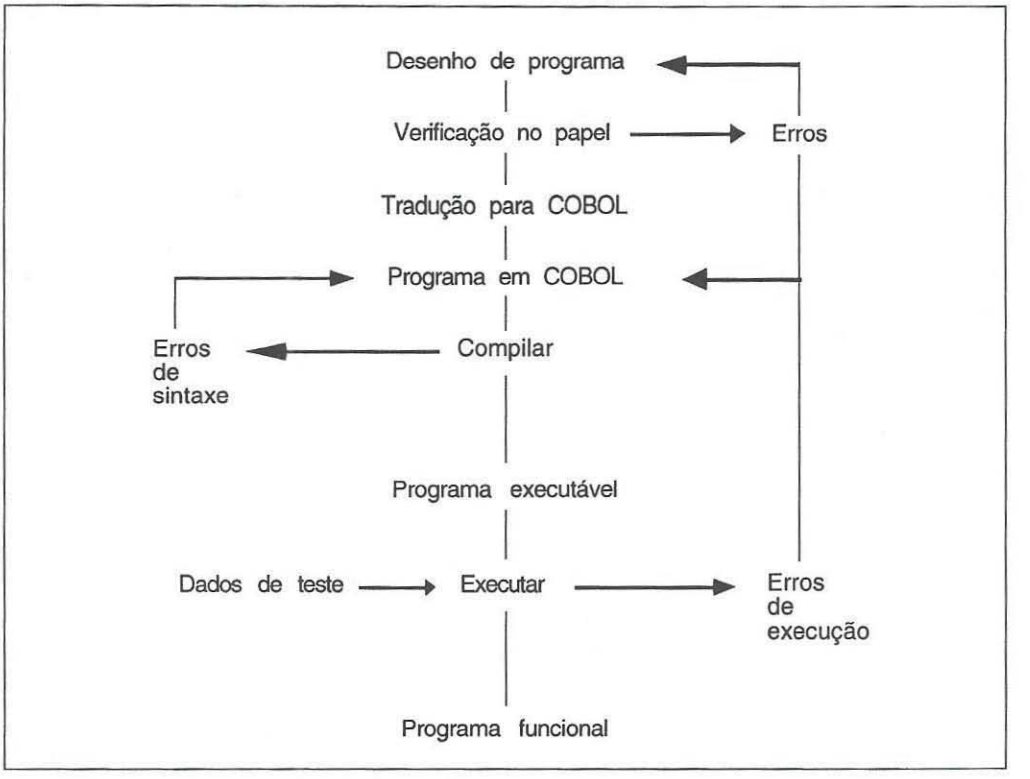


Fig. 1. 2

Para além destas operações, terão também de existir facilidades para a descrição da forma dos dados e testes das condições, para sustentar *d*) e *e*), acima mencionadas.

Estes aspectos, juntamente com um conceito adicional designado por refinamento, que mais não é do que uma técnica de construção de programas, definirão o segmento de COBOL descrito na primeira parte deste livro. Este segmento é suficientemente eficaz para escrever programas bastante complexos.

1.3 — O COMPUTADOR REAL: «HARDWARE» E «SOFTWARE»

Chama-se *máquina abstracta* à máquina mostrada na figura 1.1 e descrita na secção prévia, porque, na verdade, uma tal máquina não existe. Um programa escrito em COBOL e desenhado para correr nesta máquina abstracta terá de ser traduzido para uma linguagem de baixo nível, que pode ser executada numa máquina real. O programador não precisa de se preocupar com esta máquina real, ou hardware, dado que ela pode ser preparada para obedecer às regras para a máquina abstracta.

Existem duas fases distintas quando se põe a correr um programa de COBOL: compilação e execução. A compilação é o processo de traduzir um programa de COBOL para uma linguagem de baixo nível adequada e é executada por um programa especial que se chama compilador, que faz parte do software de um sistema particular de computador. Dado que este proces-

so de tradução é mecânico, o programa de COBOL necessita de ser escrito numa forma precisa, de acordo com certas regras de sintaxe. Assim, parte da compilação consiste em verificar se a sintaxe do programa está correcta e em identificar erros. No entanto, o facto de um programa estar sintacticamente correcto não significa que o mesmo fará algo útil. Do mesmo modo, frases em inglês podem ser sintacticamente correctas, mas sem significado: «The mat sat on the cat»[1], por exemplo. Uma vez que o programa foi compilado com êxito, precisa de ser executado e testado com uma variedade de tipos de dados de *input*, para provar que funciona. O processo de verificação de um programa relaciona-se com a posição e correcção de erros que os testes provaram existir. As fases de desenvolvimento do programa estão resolvidos na figura 1.2.

Os alunos principiantes em programação deixam-se desencorajar frequentemente porque os seus programas não resultam à primeira tentativa. Criar um programa que funcione é um processo repetitivo; geralmente, dá-se um passo para a frente e dois para trás. A primeira fase deste processo é desenhar o programa e verificar o desenho no papel, percorrendo-o manualmente. O programa pode então ser traduzido para COBOL, transformado em algo legível pela máquina, usando, talvez, um terminal de computador, e pode fazer-se uma tentativa para a sua compilação. Isto resultará, quase certamente, em erros de sintaxe, detectados pelo compilador, que terão de ser corrigidos pelo programador.

Eventualmente, o programa compilará, produzindo um programa executável, que poderá ser então posto em ficheiro de *input* de dados de teste. O resultado deste teste será, muito provavelmente, um erro de execução. Os erros de execução podem dividir-se em duas categorias: erros em tempo de execução, em que o programa pára porque ocorre alguma condição inesperada, e erros lógicos, em que o programa produz os resultados errados. Corrigir um erro de execução pode implicar ter de voltar atrás e modificar o programa de COBOL, possivelmente alterar também o desenho, recompilar o programa e executá-lo então novamente. Alternativamente, os erros de execução podem ser causados por dados de teste incorrectos, que fazem que o programa produza resultados inesperados. Uma vez que o programa tenha processado com êxito um ficheiro de *input*, precisará então de ser testado com outros ficheiros de *input*, dado que, geralmente, um único ficheiro de dados de testes é insuficiente para testar todas as possibilidades.

---

[1] «O capacho sentou-se no gato.» *(N. do T.)*

2. A LINGUAGEM DE DESENHO DE PROGRAMAS

Este capítulo introduz a linguagem de desenho de programas, que será usada ao longo do livro para o desenho de algoritmos. Inicialmente, as operações de linguagem de desenho são definidas em termos do «computador de COBOL», descrito na secção 1.2. Estas definições baseiam-se na disciplina particular imposta pelo uso da estrutura específica de programas de COBOL e não são definições da construção da linguagem COBOL. As operações são então usadas para construir alguns programas simples, que são seguidos passo a passo para ilustrar como funcionam as operações.

Nas descrições das operações, qualquer coisa que apareça entre parênteses angulares, < e >, significa algo que o programador vai ter de escolher. Por exemplo, lê-próximo-<nome do ficheiro> indica que pode definir-se uma operação através da substituição do nome de um ficheiro no lugar de <nome do ficheiro>. Assim, «lê-próximo-empregado» e «lê-próximo-nota-encomenda» poderiam ser criados para ficheiros chamados «empregado» e «nota-encomenda», respectivamente. Porém, «lê-próximo-empregado» e «empregado» não poderiam ser criados a partir do padrão lê-próximo-<nome de ficheiro>.

2.1 — OPERAÇÕES BÁSICAS

As operações definidas nesta secção,são as que levam a cabo uma tarefa simples sem mudar a ordem de execução das operações.

lê-próximo-<nome do ficheiro>

Tenta ler o próximo registo do ficheiro de *input,* chamado <nome do ficheiro>, para a área de *input.* Se esse registo não existe, então a condição de fim-de-<nome do ficheiro> é tornada verdadeira. Inicialmente, fim-de-<nome do ficheiro> é falsa, e os conteúdos da área de input são indefinidos.

```
isto é,   lê-próximo-vendas
          lê-próximo-ficheiro-cliente
```

coloca-<nome do ficheiro>

Transfere o conteúdo da área de armazenamento de dados, chamada <nome de registo>, para o ficheiro de *output,* via área de *output.* Depois da transferência, a área definida por <nome do regito> é colocada a espaços. Inicialmente, a área de <nome de registo> será preenchida com espaços.

```
isto é,   coloca-detalhes
          coloca-linha-sumário
```

transfere <item-1> para <item-2>

Copia o valor armazenado em <item-1> para zona chamada <item-2>. O conteúdo de <item-1> não se modifica, mas qualquer valor contido em <item-2> é perdido. Um caso especial desta operação é:

transfere <constante> para <item-2>

onde <constante> é um valor fixo que deve ser colocado na posição <item-2>.

isto é, transfere preço–total para preço–tot–resumo
transfere 25 para média–desconto
transfere «cliente–especial» para comentário–detalhe

calcular <item–resultado> = <expressão–aritmética>

A <expressão–aritmética> é calculada e o resultado colocado na posição chamada <nome–resultado>. A expressão aritmética será construída a partir de nomes de localizações de armazenamento, constantes, e operadores aritméticos: mais, menos, multiplicar e dividir, e parênteses.

A ideia de uma «ordem de precedência» é requerida para definir o significado de uma expressão aritmética. Por exemplo, sem a definição de ordem de precedência não é possível dizer se o resultado de 2 + 3 * 4 é igual a 14 ou a 20. A ordem de precedência é assim definida.

( ) expressões entre parênteses
* / multiplicar e dividir
+ — somar e subtrair

Se dois ou mais operadores aparecem ao mesmo ao mesmo nível de precedência numa expressão aritmética, são então calculados da esquerda para a direita.

Isto significa que as expressões entre parênteses são calculadas primeiro, seguidas das multiplicações e divisões e, finalmente, adições e subtracções. Poderá ser necessário aplicar estas regras repetidamente, se as expressões contiverem parênteses «aninhados», isto é, parênteses entre parênteses...

isto é, calcular preço–total =
preço–venda * quantidade–venda
calcular preço–descontado =
(1 — média–desconto) * preço–total

Note–se que, neste contexto, o sinal de igual significa «torna–se igual a» em vez de «é igual a». Portanto,

calcular registos–total = registos–total + 1

é perfeitamente sensível e significa pegar no valor contido no item–dados total–registos, somar 1 e armazenar o resultado em registos–total.

Uma sequência de operações básicas será executada na ordem escrita, uma após outra, do princípio ao fim. Assim, a sequência:

transfere primeiro–item para segundo–item
transfere segundo–item para primeiro–item

não mudará o conteúdo do primeiro–item e segundo–item. O valor do primeiro–item será copiado para o segundo–item, e então este novo valor será novamente copiado do segundo–item para o primeiro–item, ficando ambas as posições com o mesmo valor: o valor original do primeiro–item.

2.2 — SELECÇÃO E REPETIÇÃO

Antes de considerar os detalhes das operações de selecção e repetição, torna–se necessário definir as condições utilizáveis para controlar essas operações.

Condições

As condições que serão inicialmente permitidas nas operações de selecção e repetição recaem sobre dois grupos.

As condições comparativas são da seguinte forma:

<item–nome> <operador–comparação> <valor–comparado>

em que o <operador–comparação> é um dos que se seguem:

<	menor que	não	<	não menor que
>	maior que	não	>	não maior que
+	igual	não	=	não igual

e o <valor–comparado> é outro nome–item ou uma constante.

isto é, quantidade–vendas < 20
taxa–pagamento > taxa–limite
super–código–empregado não = «A»

Note–se que os operadores «>» e «<» não são referidos, mas são equivalentes a, respectivamente, «não <» e «não >».

As condições de ficheiro testam o estado actual do ficheiro de *input* da seguinte forma:

fim–de-<nome do ficheiro> não fim–de-<nome do ficheiro>

A condição de fim–de-<nome do ficheiro> é verdadeira se não existirem mais registos disponíveis em <nome do ficheiro>, porque a última lê–próximo-<nome do ficheiro> falhou. Esta condição é falsa após um lê–próximo-<nome do ficheiro> ser feito com êxito ou, então, no princípio do programa, quando não há informação disponível. A condição não fim–de-<nome do ficheiro> é somente a conversão de fim–de-<nome do ficheiro>,

isto é, fim–de–vendas
não fim–de–ficheiro–clientes

Selecção

A operação de selecção assume duas formas:

```
if <condição> then                  if <condição> then
    <procedimento-1>                        <procedimento-3>
else                    endif
    <procedimento-2>
endif
```

(*a*) (*b*)

A operação de selecção (*a*) funciona do seguinte modo:

1) A <condição> é avaliada para dar falso ou verdadeiro;
2) Se a <condição> for verdadeira, então executar <procedimento-1>;
3) Se a <condição> for falsa, então executar <procedimento-2>;
4) Após seleccionar o procedimento adequado, continuar executando a próxima operação a seguir a endif.

A versão simplificada da operação de selecção apresentada acima, em (*b*), funciona de forma similar, excepto quando a <condição> é falsa e a operação nada executa. Assim, o <procedimento-3> é executado apenas quando a <condição> é verdadeira.

Os «procedimentos» especificados podem ser uma sequência de uma ou mais operações básicas, ou o nome de uma sequência de operações a serem executadas (ver 2.4), ou uma mistura dos dois.

Exemplos de operações de selecção:

```
if taxa–pagamento not < taxa–limite then
      calcular total–pagadores–mais elevados = total–pagadores–mais elevados + 1
      rotina–taxa–superior
else
      calcular taxa–a–pagar = taxa–pagamento * média–base
endif
if preço–casa > limite–selo then
      soma–imposto–selo
endif
```

Repetição

Inicialmente, só uma operação de repetição será introduzida, e a sua forma geral é:

```
until <condição> do
            <procedimento>
enduntil
```

Esta operação é definida do seguinte modo:

1) A <condição> é avaliada por verdadeira ou falsa;
2) Se a <condição> for verdadeira, então executa–se a operação que se segue a enduntil.
3) Se a <condição> for falsa, então executar o <procedimento> e voltar ao passo 1), acima mencionado.

Assim, o <procedimento> é executado repetidamente até que a <condição> seja verdadeira quando a operação que se segue a enduntil é executada. Se a <condição> for verdadeira quando o until é executado pela primeira vez, então toda esta operação não se efectua e a operação a seguir ao enduntil é executada imediatamente.

O <procedimento> já especificado é similar ao da selecção: uma sequência de operações básicas, o nome de uma sequência de operações, ou uma mistura dos dois. Por exemplo:

```
until fim–de–empregado do
  calcular–taxa
  ler–próximo–empregado
enduntil
```

Neste exemplo, «calcular–taxa» é o nome de uma sequência de operações a executar.

2.3 — PROGRAMAS SIMPLES

Esta secção vai analisar alguns exemplos de possíveis programas para ilustrar o material apresentado nas duas secções anteriores.

Exemplo 2.1

Um programa muito simples, mas completo, que executa algo útil é:

```
mostrar–registos–vendas
        ler–proximo–vendas
        until fim–vendas do
                transferir registo–vendas para linha–impressora
                colocar–linha–impressora
                ler–proximo–vendas
enduntil
```

Aqui está um ficheiro de *input,* chamado «vendas», que contém registos, chamados «registo–vendas», e um registo de *output,* chamado «linha–impressora», que se convenciona pertencer a um ficheiro de linhas de impressora. O programa começa por tentar ler um registo do ficheiro de vendas. Se não existirem registos neste ficheiro, então a condição fim–de–vendas será tomada verdadeira, o until não terá qualquer efeito e o programa terminará imediatamente.

Se existir, pelo menos, um registo no ficheiro, então a condição de fim–de–ficheiro será falsa e as operações entre o until e o enduntil repetir–se–ão até que a condição fim–de–vendas seja verdadeira. As operações dentro do ciclo transferem simplesmente o registo de *input* actual para o registo de *output,* transferem o registo de *output* para o ficheiro de *output* (escrevendo uma linha, neste caso) e preparam uma nova leitura ao ficheiro de vendas.

Assim, este programa simples irá escrever o conteúdo total do ficheiro de vendas na impressora, exactamente como os registos aparecem no ficheiro de *input.* Isoladamente, isto pode ser ocasionalmente útil, mas é importante porque constitui a construção básica para um grande número de programas que lêem registos de um ficheiro de *input,* processam cada um dos registos e fazem o *output* dos resultados.

A indentação apresentada no exemplo anterior não é obrigatória, mas é útil para realçar a estrutura do programa.

Exemplo 2.2

Alargando o exemplo anterior, suponha que cada registo de vendas contém o item número de um item vendido, «vendas–item–num», a quantidade do item vendido, «quantidade–vendas», e o preço por item, «preço–vendas». Modifique o exemplo 2.1 para escrever o item número, o preço total imputado e o desconto de quantidade. O desconto é 10% do preço total imputado, se a quantidade vendida for superior a dezanove itens. Antes de fazer o desenho de programa, os itens do registo de output poderiam ser assim definidos:

```
linha–impressora
        item–num–impressora          item número
        quantidade-impressora        quantidade vendida
        preço-impressora             preço por item
preço-tot-impressora          preço total imputado
        desconto-impressora          desconto dado
```

O problema não requer que os valores da quantidade vendida e do preço por item sejam imprimidos, mas imprimir estes valores para verificar que eles foram correctamente gravados é uma boa prática. É muito frustrante procurar um erro lógico num programa quando são dados incorrectos que provocam o problema.

O programador pode escolher quaisquer nomes para ficheiros, registos e itens[1], sujeito a restrições impostas pela linguagem de programação utilizada. Em COBOL, os nomes devem começar por uma letra e ser construídos a partir de letras, dígitos e hifenes. É uma boa prática escolher nomes que tenham sentido e mostrar a estrutura dos dados, isto é, todos os campos que pertençam a linha–impressora têm o sufixo «–impressora» e nomes que indicam o conteúdo do campo.

Um possível desenho de programa para o problema modificado é:

```
preço-e-desconto
        ler-próximo-vendas
until fim-vendas do
        calcular preço-total = quantidade–vendas * preço–vendas
        if quantidade-vendas > 19 then
                calcular desconto = 0.1 * preço–total
                calcular preço-total = preço-total — desconto
        else
                transferir[2] 0 para desconto
        endif
        mover vendas-campo-num para campo–num–impressora
        mover quantidade-vendas para quantidade–impressora
        mover preço-vendas para preço-impressora
        mover preço-total para preço-tot-impressora
        mover desconto para desconto-impressora
        colocar[3] linha-impressora
        ler-próximo-vendas
enduntil
```

Para ilustrar a acção deste programa, considere que o ficheiro de vendas contém alguns dados úteis e que o efeito do ler–próximovendas inicial é obter os seguintes dados na área de «input»:

Campo-vendas	quantidade-vendas	preço–vendas
12345	20	6.36

Ao entrar no ciclo until pela primeira vez, os valores das variáveis temporárias preço–total e desconto serão indefinidos:

preço-total	desconto
?	?

Depois de se realizar a operação

```
calcular preço-total = quantidade-vendas * preço–vendas
```

---

[1] A partir de agora designado por «campo». *(N. do T.)*
[2] A partir de agora designado por «mover». *(N. do T.)*
[3] A partir de agora designado por «escrever». *(N. do T.)*
Estas alterações justificam-se porque são os termos utilizados em informática no nosso país, possibilitando assim uma generalização. *(N. do T.)*

a posição de armazenamento temporário conterá agora:

preço-total	desconto
127.2	?

A operação seguinte é a selecção e, desde que a quantidade–vendas seja maior do que 19, o procedimento seleccionado começará com:

calcular desconto = 0.1 * preço-total

o que tem o seguinte efeito:

preço-total	desconto
127.2	12.72

Isto será seguido pela operação

calcular preço-total = preço-total — desconto

que toma o valor actual de preço-total, subtrai–lhe o desconto e coloca o resultado de novo na posição de armazenamento preço–total. Logo, os novos valores no armazenamento temporário serão:

preço-total	desconto
114.48	12.72

A próxima operação a realizar é a que se segue a endif

mover vendas-campo-num, para campo-num–impressora

Esta operação e as quatro acções de mover seguintes construirão uma linha–impressora com o formato:

campo-num-impressora	quantidade–impressora	preço–impressora
12345	20	6.36

preço-tot-impressora	desconto-impressora
114.48	12.72

O conteúdo da linha-impressora será então movido para o ficheiro de *output,* produzindo uma única linha de *output* impresso, possivelmente:

12345 20 6.36 114.48 12.72

e tenta-se ler o registo seguinte do ficheiro de vendas.

De que modo o formato actual do *input* e do *output* são especificados é deliberadamente deixado por definir por agora — isto será tratado no capítulo 3 e capítulos subsequentes.

Este programa utiliza posições de armazenamento temporárias, designadas por «preço–total» e «desconto», porque se requerem estes valores para futuros cálculos. Regra geral, se um valor a calcular for requerido para computação futura, deve ser armazenado numa posição de

armazenamento temporário, senão pode ser directamente colocado no registo de *output*. Isto porque a representação externa do valor, neste caso na linha–impressora, não será necessariamente adequada para computação dentro do computador.

Exemplo 2.3

A fim de ilustrar mais algumas técnicas de programação e de alargar a experiência do leitor, considere o seguinte problema. Um ficheiro de propriedades de um agente de compra e venda de propriedades contém um registo para cada propriedade à venda, sumariando os detalhes de cada uma delas, incluindo o preço, chamado «preço–propriedade». Pretende–se um programa para examinar o ficheiro e contar o número de propriedades à venda por menos de 25.000 libras e expressá–lo como a percentagem do número total de propriedades à venda. Esta é uma versão muito simples de outro tipo de problema, uma «agregação», que envolve o processamento de um ficheiro e a recolha de informação acerca de todo o ficheiro em vez de registos individuais.

Ao processar o ficheiro de propriedades, serão necessários dois contadores: um para as propriedades à venda por menos de 25.000 libras, «propriedade–barata», e outro para o número total de propriedades, «propriedade–total». O programa tem três partes: inicializar estes contadores, processar todos os registos do ficheiro e actualizar os contadores, e imprimir os resultados da contagem. O *output* do programa consistirá numa única linha, sumariando a informação recolhida:

```
linha-sumário
        barato-sumário          número de propriedades
                                        abaixo de 25.000 libras

        total-sumário           número total de propriedades
        percentagem-sumário     percentagem das propriedades «baratas»
```

Repare que o número total de propriedades será imprimido para propósitos de verificação, embora não seja estritamente necessário.

Um desenho de programa possível para este problema é:

```
sumário-propriedade
        mover 0 para propriedade-barata
        mover 0 para propriedade-total
        ler-próximo-propriedade
        until fim-propriedade do
                calcular propriedade-total = propriedade–total + 1
                if preço-propriedade < 25.000 then
                        calcular propriedade-barata = propriedade-barata + 1
                endif
                ler-próximo-propriedade
        enduntil
        mover propriedade-barata para barata–sumário
        mover propriedade-total para total–sumário
        calcular percentagem–sumário =
                propriedade–barata / propriedade–total * 100
        escrever linha-sumário
```

Trabalhar este programa em pormenor fica como exercício para o leitor, mas há um certo número de pontos neste desenho de programa que devem ser sublinhados. Aos contadores «propriedade–barata» e «propriedade–total» deve dar–se um valor inicial de zero, senão eles podem conter um valor indefinido quando o ciclo until é executado pela primeira vez. A operação if mos-

tra um exemplo de selecção sem um rumo de acção alternativo; se o preço–propriedade for menor do que 25.000 libras, então o contador «propriedade–barata» é incrementado em uma unidade, senão nada acontece neste ponto. Na parte final deste programa, «percentagem–sumário» é um exemplo de um campo que pode ser directamente calculado para *output,* porque o seu valor não é necessário para computação futura.

2.4 — REFINAMENTO

Um pequeno programa como o do exemplo 2.2, apresentado anteriormente, pode ser redigido e verificado no papel numa questão de poucos minutos. A solução pode então ser traduzida para uma linguagem de programação, compilada e testada. Se não executar correctamente, então, geralmente, é fácil detectar o erro, fazer uma alteração rápida ao programa e testá–lo de novo, e assim por diante. Existem muitos problemas triviais que podem ser resolvidos através de métodos aleatórios, porque todo o problema é suficientemente pequeno para ser compreendido como uma unidade única. No entanto, como o tamanho e a complexidade dos problemas aumenta, é necessario desenvolver um processo metódico de criar programas.

Antes de descrever uma tal metodologia, é importante considerar quais são os seus principais objectivos. O principal requisito para qualquer programa de computador é que ele seja correcto, ou seja, que produza o *output* esperado para todos os *inputs* possíveis. Assim, a metodologia deve ser estruturada de modo a maximizar a possibilidade de o programa estar correcto. Uma forma alternativa de conseguir o mesmo fim é que a metologia minimize as possibilidades de erro através do uso de técnicas «seguras» para a construção do programa.

Um aspecto muito importante do processamento de dados comerciais é que os programas são usualmente escritos na perspectiva de serem usados ao longo de vários anos. Inevitavelmente, os programas necessitarão de ser modificados devido a alteração de algumas circunstâncias, tais como emendas às leis fiscais ou revisões nas políticas das companhias. Na maior parte dos casos, um programa será modificado ou sujeito a «manutenção» por um programador diferente do seu autor original. Muitas companhias têm um grupo de programadores cujo único trabalho é a manutenção do *software* existente. O facto de um programa vir provavelmente a ser modificado por outro programador exige que o mesmo seja escrito num estilo fácil de compreender e seja simples de modificar.

É do conhecimento geral que, se um método é desenvolvido no sentido de melhorar aspectos específicos de um processo, existirá então um «compromisso» contra um ou mais aspectos do mesmo problema. Os principais objectivos que foram referidos são a correcção e a possibilidade de manutenção, e o compromisso existirá contra a eficiência na execução do programa. Os programas escritos utilizando os métodos descritos neste livro não são necessariamente as soluções mais eficientes para o problema, porque é quase sempre possível aumentar a velocidade de execução de um programa através de «acertos» cuidadosos. No entanto, a maior parte dos custos de um projecto de computador, hoje em dia, associa–se à mão–de–obra necessária para o desenvolvimento e manutenção do *software,* e, assim, a eficiência terá de ser uma consideração secundária.

Não se requer que exista apenas uma metodologia que prencha os critérios acima mencionados, mas o que importa é que qualquer metodologia escolhida seja aplicada com consistência. A técnica de desenvolvimento de programas descrita e utilizada ao longo deste livro é chamada refinamento progressivo.

O princípio básico do refinamento progressivo é que qualquer problema não trivial pode ser repartido em vários subproblemas, que poderão então ser resolvidos, partindoos também, se necessário, em posteriores subproblemas, etc. Aplicar esta ideia à programação significa que uma descrição do problema total é escrita como um algoritmo em alto nível, ligado a vários problemas.

Cada um desses subproblemas pode então ser «refinado», escrevendo-se quer a sua solução, quer um algoritmo em termos de posteriores subproblemas, melhorando-os então, etc. Assim, todo o processo vem a ser um refinamento progressivo do problema, a partir de um esboço do mesmo, para um desenho de problema completo e detalhado.

Exemplo 2.4

Para ilustrar o formato de refinamento na linguagem de desenho de programa, considere novamente o exemplo 2.2. Uma abordagem alternativa ao problema poderia ter começado com o esboço do desenho de programa da seguinte forma:

```
preço-e-desconto-2
        ler-próximo-vendas
        until fim-vendas do
                encontrar-preço-e-desconto
                produzir-linha  impressora
                ler-próximo-vendas
        enduntil
```

Deve ficar claro que este programa processa cada um dos registos do ficheiro de vendas, por analogia com o exemplo 2.1, mas o processamento actual mantém-se nesta fase.

Os nomes de refinamento, tais como «encontrar-preço-e-desconto» e «produzir-linha-impressora», são inventados pelo programador e colocados no desenho de programa no actual nível de desenvolvimento. Estes refinamentos serão divididos em sequências de operações de refinamento de nível inferior na fase de desenvolvimento seguinte. A ligação entre os dois níveis de desenho é fornecida pelo nome de refinamento que irá aparecer, sublinhado, como cabeçalho do refinamento.

Tendo decidido que «imprimir-e-desconto-2» é a forma básica correcta da solução, os problemas «encontrar-preço-e-desconto» e «produzir-linha-impressora» podem agora ser refinados da seguinte forma:

```
encontrar-preço-e-desconto
        calcular  preço-total = quantidade-vendas * preço-vendas
        if quantidadevendas > 19 then
                calcular desconto = 0.1 * preçototal
                calcular preço-total = preço-total — desconto
        else
                mover 0 para desconto
        endif
produzir-linha-impressora
        mover  campo-num-vendas  para  campo-num-impressora
        mover  quantidade-vendas  para  quantidade-impressora
        mover  preço-vendas para  preço-impressora
        mover  preço-total para  preço tot-impressora
        mover  desconto para  desconto-impressora
        escrever-linha-impressora
```

As sequências de operações especificadas nos refinamentos, «encontrar-preço-desconto» e «produzir-linha-impressora», são as mesmas especificadas no desenho de programa do exemplo 2.3.

Ao percorrer um desenho de programa que utilize refinamentos, as operações especificadas para cada um são efectivamente substituídas pelo nome do mesmo. O modo alternativo de vi-

sualizar este processo é: sempre que o nome do refinamento seja encontrado, execute as instruções nele especificadas e volte à operação seguinte, após o nome de refinamento. No fim de cada refinamento há uma operação implícita que diz para voltar atrás até ao lugar onde este refinamento foi invocado e continuar a processar a partir desse ponto.

Exemplo 2.5

O exemplo 2.3 pode também ser reformulado de modo que a estrutura geral do programa seja separada do detalhe do problema particular. Aqui está um desenho de programa alternativo:

```
sumário-propriedade-2
        inicializar-contadores
        ler-próximo-propriedade
        until fim-propriedade do
                actualizar-contadores
                ler-próximo-propriedade
        enduntil
        produzir-sumário

inicializar-contadores
        mover 0 para propriedade-barata
        mover 0 para propriedade-total

actualizar-contadores
        calcular propriedade-total = propriedade-total + 1
        if preço-propriedade < 25.000 then
                calcular propriedade-barata = propriedade-barata + 1
        endif

produzir-sumário
        mover propriedade-barata para barata-sumário
        mover propriedade-total para total-sumário
        calcular percentagem-sumário = propriedade-barata / propriedade-total * 100
        escrever-linha-sumário
```

As sequências de operações contidas nos três refinamentos correspondem às que foram desenvolvidas no exemplo 2.3.

2.5 — ESTUDO DE CASO

A ABC Carpet Company mantém um ficheiro de encomendas diárias, em que cada registo contém: um número de encomenda, o nome e morada do cliente, o número de metros quadrados de alcatifa encomendada, o preço por metro quadrado e a zona de entrega. A zona de entrega está codificada com uma simples letra «A» ou «B», indicando a distância a partir do armazém. A política da companhia relativamente aos custos das entregas é que a encomenda seja entregue grátis se o custo da alcatifa exceder as 200 libras; caso contrário, existe um preço adicional de 5 libras para a zona «A» e de 10 libras para a zona «B».

Requer-se um programa que irá processar as encomendas do ficheiro e calcular o preço da alcatifa, o preço da entrega, se existir, e o total a cobrar por cada encomenda. Após processar todas as encomendas, o programa deverá produzir os totais de venda de alcatifa, os custos de entrega e o total dos dois.

O ficheiro de *input* pode chamar-se «ficheiro-encomendas» e a estrutura de um registo, neste ficheiro, será:

```
registo-encomenda
        número-encomenda            número de encomenda
        cliente-encomenda           nome e morada do cliente
        tamanho-encomenda           tamanho da alcatifa (metros quadrados)
        preço-encomenda             preço por metro quadrado
        zona-encomenda zona de entrega «A» ou «B»
```

e o registo de *output* poderia ter a seguinte estrutura:

```
linha detalhe
        numero-detalhe              número de encomenda
        cliente-detalhe             nome e morada do cliente
        tamanho-detalhe             tamanho da alcatifa
        preço-detalhe               preço por metro quadrado
        zona-detalhe                zona de entrega «A» ou «B»
        preço-custo-detalhe         preço da alcatifa
        custo-entrega-detalhe       custo de entrega
        custo-encomenda-detalhe     custo total da encomenda
```

A quantidade de *output* pode ser reduzida se não se escrever o tamanho, preço ou zona, mas, tal como foi previamente recomendado, estes são incluídos para verificação da encomenda.

Este registo de output representa o *output* requerido para cada encomenda, mas existe também a necessidade de executar o *output* dos totais, após executar o processamento de todas as vendas. Neste caso, convencionar-se-á que os campos «detalhe-preço-alcatifa», «detalhe-preço-entrega» e «detalhe-preço-encomenda» são suficientemente grandes para comportar os totais, e, assim, o registo acima pode ser utilizado para fazer o *output* dos totais.

Este problema requer não só o processamento de registos individuais mas também a acumulação de totais; por isso, a estrutura do programa será uma combinação das estruturas utilizadas nos exemplos 2.2 e 2.3. A estrutura geral do programa será então da seguinte forma:

```
encomendas-alcatifa
        inicializar-contadores
        ler-próximo-ficheiro-encomenda
        until fim-de-ficheiro-encomenda do
                processar-encomenda
                actualizar-contadores
                ler-próximo-ficheiro-encomenda
        enduntil
        produzir-totais
```

Há três totais que são exigidos, mas, como o total de todas as encomendas pode ser calculado a partir da soma das alcatifas vendidas mais a soma dos custos de entrega, somente dois contadores são necessários. O custo total das alcatifas será acumulado em «alcatifa-total» e o total cobrado por entregas em «entrega-total». Assim, «inicializar-totais» poderá ser refinado do seguinte modo:

```
inicializar-totais
        mover 0 para alcatifa-total
        mover 0 para entrega-total
```

É agora necessário registar os nomes destes campos de dados temporários e adicionar quaisquer outros do mesmo tipo à lista à medida que o programa é desenvolvido.

```
campos-temporários
        alcatifa-total      custos de alcatifas por total de clientes
        entrega-total       custos de entregas por total de clientes
```

O processamento de uma encomenda individual, «processar-encomenda», é razoavelmente mais claro e um possível refinamento será:

```
processar-encomenda
        calcular-custo-alcatifa = tamanho-encomenda * preço-encomenda
        if preço-alcatifa not > 200 then
                encontrar-preço-entrega
        else
                mover 0 para preço-entrega
        endif
        imprimir-detalhe-encomenda
```

O cálculo do preço de entrega é deixado para posterior explicação (encontrar-preço-entrega), tal como a construção da linha de *output* (imprimir-encomenda-detalhes). Note que «preço-alcatifa» e «preço-entrega» terão de ser campos de armazenamento temporário, por serem ambos requeridos para cálculos posteriores.

A lista dos campos de dados temporários precisa de ser actualizada:

```
campos  temporários
        alcatifa-total      custos de alcatifa por total de clientes
        entrega-total       custos de entregas por total de clientes
        custo-alcatifa      custo de alcatifa por cliente
        custo-entrega       custo de entrega por cliente
```

Note que, na prática, esta lista pode ser reconstruída numa folha de papel à parte, em vez de ser repetida.

O refinamento para «actualizar-totais» é agora muito simples:

```
actualizar-totais
        calcular alcatifa-total = alcatifa-total + custo-alcatifa
        calcular entrega-total = entrega-total + custo-entrega
```

O último refinamento de todo o desenho de programa é «produzir–totais». Para distinguir a linha sumário das linhas de encomendas vulgares, uma mensagem adequada pode ser posta, digamos, dentro do campo do nome e morada do cliente, «cliente–detalhe», deixando os outros campos irrelevantes a espaços. O total para todas as encomendas pode ser directamente calculado e posto em *output* como a soma dos dois contadores «alcatifa-total» e «entrega-total». Assim, um possivel refinamento é:

```
produzir-totais
        mover «totais» para cliente-detalhe
        mover alcatifa-total para custo-alcatifa-detalhe
        mover entrega-total para custo-entrega-detalhe
        calcular custo-encomenda-detalhe = alcatifa-total + entrega-total
        escrever-linha-detalhe
```

Existem agora dois refinamentos de segundo nível de «processar-encomenda» para serem resolvidos. O cálculo de custo de entrega «encontrar-custo-entrega» é uma selecção clara do tipo:

```
encontrar-custo-entrega
        if zona-encomenda = «A» then
                mover 5 para custo-entrega
        else
                mover 10 para custo-entrega
        endif
```

Note que esta operação de selecção assume que se «zona-encomenda» não é igual a «A», então será «B»; na prática, isto pode ser uma grave suposição.

O refinamento final requerido é «imprimir-detalhes-encomenda» para contruir a linha-detalhe, e isto é bastante claro:

```
imprimir-detalhe-encomenda
        mover número-encomenda para número-detalhe
        mover cliente-encomenda para cliente-detalhe
        mover tamanho-encomenda para tamanho-detalhe
        mover preço-encomenda para preço-detalhe
        mover zona-encomenda para zona-detalhe
        mover custo-alcatifa para custo-alcatifa-detalhe
        mover custo-entrega para custo-entrega-detalhe
        calcular custo-encomenda-detalhe = custo-alcatifa + custo-entrega
        escrever-linha-detalhe
```

A execução deste programa não vai ser demonstrada passo a passo; em vez disso, aqui está um exemplo de um fecheiro de *input:*

1513	J. Stuart, Flodden	25	15.50	A
1314	R. Bruce, Bannockburn	10	5.60	A
1297	W. Wallace, Stirling Bridge	10	20.00	A
1645	J. Graham, Inverlochy	20	10.01	A
1388	J. Douglas, Otterburn	20	14.40	B
1745	C. Stuart, Prestonpans	15	12.90	B
1689	J. Graham, Killiecrankie	20	10.00	B

Percorrendo as operações especificadas no desenho de programas, seria possível o leitor idealizar que o *output* seria da seguinte forma:

1513	J. Stuart, Flodden	25	15.50	A	387.5	0	387.50
1314	R. Bruce, Bannockburn	10	5.60	A	56.00	5	61.00
1297	W. Wallace, Stirling Bridge	100	20.00	A	200.00	5	205.00
1645	J. Graham, Inver lochy	20	10.01	A	200.20	0	200.20
1388	J. Douglas, Otterburn	20	14.40	B	288.00	0	288.00
1745	C. Stuart, Prestonpans	15	12.90	B	193.50	10	203.50
1689	J. Graham, Killiecrankie	20	10.00	B	200.00	10	210.00
	Total				1525.20	30	1555.20

Este problema é, obviamente, uma versão simplificada de um verdadeiro problema e é irrealista por um certo número de razões. Uma importante suposição em «encontrar-custo-entrega» é que os dados são correctos; isto pode apenas ser presumido se algum programa prévio tiver já «validado» os dados, rejeitando quaisquer valores incorrectos. O assunto da validação de dados será tratado no capítulo 6.

A técnica que consiste em imprimir todas as linhas usando o mesmo registo pode ser utilizada apenas se as linhas de detalhe e a(s) linha(s) sumário puderem ter exactamente o mesmo formato. Uma técnica mais geral, que utiliza registos de *output* múltiplos, é discutida na secção 5.5.

O problema é irrealista porque considera apenas uma alcatifa por cliente e não prevê nenhum desconto para o cliente que encomenda duas alcatifas de 150 libras e lhe é cobrada taxa de entrega em cada uma. As regras de custo de entrega são também bastante simplistas, mas um conjunto alternativo de regras de custo poderia ser «colado» ao programa, substituindo «encontrar-custo-entrega» por um refinamento diferente. Isto revela uma vantagem da abordagem do refinamento progressivo; um subproblema identificado pode ser alterado sem afectar a estrutura total do programa.

Exercícios 2

2.1 O *input* para o sistema de pagamentos da St. Kilda Puffin Factory consiste em registos de empregado que contêm os seguintes campos de dados:

número de empregado
pagamento mensal total
desconto anual para impostos

Todos os empregados pagam 5% do seu salário para o fundo de pensões da companhia, que são deduzidos antes de se calcular o imposto. São também sujeitos a imposto em 20% do seu rendimento, depois de deduzidos os descontos para impostos e contribuições para pensões.

Desenhe um programa de pagamentos que calcule e imprima a contribuição para a pensão, o imposto a pagar e o salário líquido mensal, para cada empregado. Pode ignorar erros de arredondamento que resultem do cálculo do desconto mensal sobre o salário, contribuição de pensões, etc.

Verifique o seu desenho de programa no papel, utilizando os seguintes dados:

Número	Pagamento mensal	Desconto para impostos
12345	500 libras	1200 libras
87654	260 libras	2964 libras
23456	300 libras	4800 libras

2.2 Os dados para o censo de 1981 contêm um registo para cada família e, entre os campos de um registo, encontram-se:

o número de pessoas da família
o número de divisões da casa

Desenhe um programa que leia estes registos e que calcule:

a) A percentagem de famílias com mais de duas pessoas,
b) A percentagem de famílias onde a taxa de pessoas por divisões exceda 1.5.

2.3 O Rockal Electricity Board administra um sistema de contas onde o *input* consiste em registos que contêm os dados seguintes:

número de conta do cliente
código de tarifa

nome e morada do cliente
número de unidades consumidas no trimestre

O código de tarifa é «D» para doméstico ou «C» para comercial e as regras para calcular a quantia a cobrar são as seguintes:

— Clientes domésticos pagam uma taxa de 3 libras por trimestre e são-lhes cobrados 10 pénis por unidade para as primeiras cem unidades e 8 pénis por unidade ultrapassadas as cem unidades.
— Clientes comerciais são cobrados a 15 pénis por unidade para as primeiras duzentas unidades e 5 pénis ultrapassadas as duzentas unidades.

Desenhe um programa de contas que:

a) Calcule o preço cobrado a cada conta de cliente e o imprima juntamente com detalhes da conta;
b) Imprima o total de unidades consumidas e a quantidade total devida por todos os clientes no actual ficheiro de *input*.

3. O PROGRAMA DE COBOL

Este capítulo começa o processo de descrição de programas de COBOL. Antes, toda a estrutura do programa precisa de ser definida, e aí existem então dois tópicos principais: estruturas de dados e algoritmos. Este capítulo irá tratar da estrutura de programa e das estruturas de dados básicas, deixando os algoritmos para o capítulo 4. No fim do capítulo 4, o leitor deverá estar em condições de escrever um programa completo simples e de o codificar em COBOL.

3.1 — ESTRUTURA DE PROGRAMAS

Um programa em COBOL divide-se em quatro unidades principais, chamadas divisions [*divisões*]. Os nomes e funções destas divisões podem ser sumariadas como se segue.

IDENTIFICATION DIVISION — tal como o nome sugere, esta *division* é largamente documental e a única entrada principal é o nome do programa.

O material documental pode incluir: o nome do programador, a data em que o programa foi escrito ou em que foi compilado e comentários introdutórios sobre a matéria do programa.

ENVIRONMENT DIVISION — aqui define-se o meio em que o programa vai ser posto a correr. Inclui detalhes sobre o tipo de computador utilizado e sobre os nomes dos ficheiros actuais usados pelo programa. Esta *division* foi projectada para ser a parte do programa dependente da máquina; na teoria, esta deveria ser a única parte do programa afectada, se um programa fosse transferido de um sistema de computador para outro.

DATA DIVISION — descreve todas as estruturas de dados que são requeridas pelo programa. A *division* encontra-se dividida em duas secções prinncipais.

> A FILE SECTION, que descreve todos os ficheiros de *input/output* usados pelo programa. Corresponde às áreas de *input* e de *output* do modelo de computador simples.
>
> A WORKING-STORAGE SECTION contém as descrições de todos os campos de armazenamento temporário requeridos pelo programa.

PROCEDURE DIVISION — especifica as operações a executar nos campos de dados definidos em DATA DIVISION. Esta *division* encontra-se dividida em secções e parágrafos, que correspondem aos refinamentos do desenho de programa.

Para começar, convencionar-se-á que uma grande parte do programa pode ser aceite como *standard* e que as estruturas de dados e de algoritmos, definidas pelo leitor, podem ser simplesmente «coladas» ao esqueleto de um programa. Quando o leitor tiver ganho alguma experiência a desenhar, escrever e, assim se espera, pôr a correr programas simples de COBOL, baseado neste esqueleto de programa *standard,* discutir-se-ão mais detalhes da estrutura de programa.

O formato do esqueleto de programa *standard* é apresentado no apêndice 1. Os próximos capítulos irão concentrar-se nos aspectos importantes do programa: a descrição de campos de dados e a tradução do desenho de programa para declarações de procedimento.

3.2 — PALAVRAS RESERVADAS E NOMES DEFINIDOS PELO UTILIZADOR

No capítulo 1 foi mencionada a ideia de compilação. Este processo consiste em traduzir um programa de COBOL para uma forma muito mais simples adequada à execução directa de um computador. Este processo de tradução é realizado por um compilador, que segue um conjunto de regras de sintaxe a fim de transformar o programa de COBOL numa linguagem de nível mais baixo, Assim, ao escrever programas em COBOL, o programador deve ter em atenção estas regras sintácticas, que podem, por vezes, parecer bastante arbitrárias.

O primeiro conjunto de regras sintácticas a considerar são as que se utilizam para construir nomes num programa. Na *data division* têm de se dar nomes de dados a todos os campos de dados usados para *input/output* e para armazenamento temporário e na *procedure division* a todos os parágrafos e secções têm de ser dados nomes de procedimento únicos. O compilador deve ser capaz de distinguir os nomes escolhidos pelo programador daqueles usados para operações *standard* em COBOL e para combinações de símbolos com significados especiais.

Existe uma lista de palavras reservadas em COBOL que têm um significado especial para o compilador e que não podem ser usadas como «palavras definidas pelo utilizador», ou seja, aqueles nomes escolhidos pelo programador. A lista completa de palavras reservadas é apresentada no apêndice 3[1]. Esta é uma longa lista e duvida-se que todo o programador de COBOL consiga lembrar-se de todas elas, e duvida-se mais ainda que algum saiba para que servem todas elas. No entanto, uma boa regra prática é a que consiste em escolher nomes que tenham sentido, construídos a partir de palavras ligadas por hífenes, e, assim, muitas das palavras reservadas serão evitadas.

Formalmente, as regras para construir nomes definidos pelo utilizador em COBOL são as seguintes:

1) O nome não deve ser uma palavra reservada (ver apêndice 3).
2) Os caracteres utilizados para construir o nome devem ser do conjunto:

   A —> Z
   O —> 9
   - (hífen)

3) Um nome não deve começar ou terminar com um hífen.
4) O comprimento máximo de um nome é de trinta caracteres.
5) Um nome de campo deve conter pelo menos uma letra.

Assim, os nomes que se seguem são válidos, embora não tenham necessariamente sentido, quer como nome de campos, quer de procedimentos:

PRINT-LINHA	LINHA-MORADA-1	1-PARTE-MORADA
A-B-C-D-E	CODIGOIMPOSTOEMP	1-2-3-4A

---

[1] Ao aparecerem no texto as palavras reservadas, que estarão em inglês segundo a sua notação original, o leitor deve reportar-se ao apêndice 3, onde aquelas, por mera razão de referência, se encontram traduzidas para português. Em caso algum deve o leitor codificar programas em COBOLutilizando o equivalente português daquelas palavras reservadas. *(N. do T.)*

e nomes sem quaisquer caracteres alfabéticos, tais como:

```
12      1-13    1-13-2-3
```

seriam válidos como nomes de procedimentos, embora não recomendados como boa prática de programação.

3.3 — A ESTRUTURA DE DADOS

O resto deste capítulo irá concentrar-se numa análise inicial da *data division.* Existem dois aspectos principais a considerar sobre os dados: a sua estrutura e a sua descrição ou tipo.

Todos os campos de dados em COBOL são considerados para fazerem parte de alguma estrutura de dados e a relação entre campos de dados define-se por números de nível. Por exemplo, olhando de novo para o exemplo 2.2, uma estrutura possível para o registo de *input* seria:

```
01      REGISTO-VENDAS
          05    NUM-VENDAS
          05    QUANTIDADE-VENDAS
          05    PRECO-VENDAS
```

Isto transmite a informação de que «REGISTO-VENDAS» é uma descrição de registo, porque tem o número de nível 01, e de que os três campos seguintes são subordinados (pertencem) a REGISTO-VENDAS, porque têm números de nível mais elevados. O facto de terem números de nível iguais indica que não há uma estrutura posterior dos dados, e, logo, estes três campos são independentes uns dos outros.

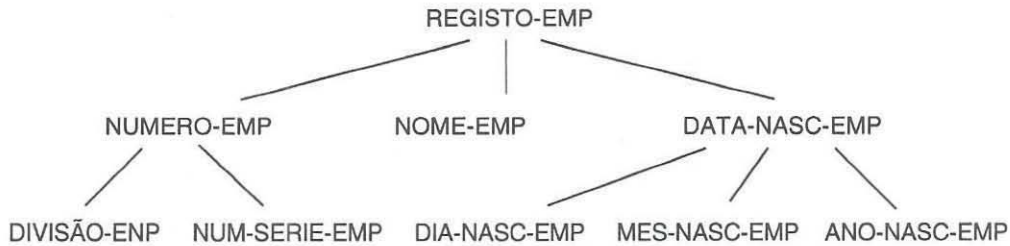
A razão para usar o número de nível 05 para os campos de dados subordinados discute-se mais tarde. O ponto importante nesta fase é que mudanças em números de nível definem estrutura; os valores numéricos de números de nível, excepto a forma 01, não são significativos.

Um modo de visualizar uma estrutura de dados em COBOL é desenhar uma «árvore», mostrando a relação hierárquica entre os campos, e, para o exemplo acima mencionado, essa árvore simples será:

Uma estrutura de registo um pouco mais complicada é:

```
01      REGISTO-EMPREGADO
          05    NUMERO-EMP
                  10    DIVISAO-EMP
                  10    NUM-SERIE-EMP
          05    NOME-EMP
          05    DATA-NASC-EMP
                  10    DIA-NASC-EMP
                  10    MES-NASC-EMP
                  10    ANO-NASC-EMP
```

Pode ver-se que REGISTO-EMPREGADO é constiuído por três partes chamadas NUMERO-EMP, NOME-EMP e DATA-NASC-EMP. Por sua vez, NUMERO-EMP é constituído por duas componentes designadas por DIVISAO-EMP e NUM-SERIE-EMP, enquanto DATA-NASC-EMP se divide em três partes: DIA-NASC-EMP, MES-NASC-EMP e ANO-NASC-EMP. A estrutura da árvore equivalente sumaria esta relação:



Um importante ponto de terminologia precisa de ser apresentado nesta fase. Os campos da árvore que não têm subordinados são chamados itens elementares, enquanto aqueles que são subdivididos se chamam itens de grupo. Neste exemplo, os itens de grupo são: REGISTO-EMPREGADO, NUMERO-EMP e DATA-NASC-EMP, e todos os outros nomes de dados se referem a itens elementares.

Os exemplos dados acima revelam uma boa prática no que respeita a números de nível e indentação. É sempre sensato utilizar números de nível que não sejam contíguos, de modo que níveis intermédios possam ser introduzidos, se necessário. O único número de nível significativo é 01, que indica o princípio de um novo registo, e o número de nível mais alto permitido é 49. O REGISTO-EMPREGADO mostrado acima teria ainda o mesmo significado se os números de nível fossem sistematicamente alterados para 11 e 21, ou 02 e 03, substituindo 05 e 10, respectivamente. Recomenda-se a indentação como ajuda visual para mostrar a estrutura dos dados.

Frequentemente, revela-se útil ser capaz de ignorar partes de um registo de *input* ou deixar áreas a espaços num registo de *output*. Em vez de inventar nomes para campos de dados simulados, existe um nome-dados especial, FILLER, que indica qualquer campo que nunca será directamente referenciado. Por exemplo, um registo de *output* pode ser definido como se segue:

```
01      PRINT-REGISTO-EMP
        05      PRINT-NUM-TRABALHO
                10      PRINT-DIVISÃO
                10      FILLER
                10      PRINT-NUM-SERIE
        05      FILLER
        05      PRINT-NOME
        05      FILLER
        05      PRINT-IDADE
```

Os FILLERs permitem ao programador descrever campos simulados que irão corresponder a áreas a espaços no registo de *output*, o que o tornará mais legível.

3.4 — TIPOS DE DADOS

Os itens elementares, definidos como os elementos indivisíveis na estrutura de dados, têm de ser descritos indicando o tipo de dados que se espera que eles contenham. Existem três tipos básicos distintos de dados no COBOL:

Tipo	Conteúdo do campo de dados
Numérico	0 —> 9 + -
Alfabético	A —> Z espaço
Alfanumérico	Qualquer carácter representável

Um item elementar deve pertencer apenas a um destes tipos básicos. A cada item elementar é dado uma *picture* [*figura*], abreviada para PIC, que define o seu tipo e tamanho. Os tipos são codificados do seguinte modo: 9 para numérico, A para alfabético e X para alfanumérico.

O código é repetido para inndicar o tamanho do campo; assim, «PIC 9999» representa quatro dígitos numéricos e «PIC AAAAA» indica cinco caracteres alfabéticos. Utiliza-se uma forma abreviada da *picture* para evitar a transcrição de longas filas de códigos para *pictures* maiores. Isto consiste em fazer seguir o código de uma contagem de repetição entre parênteses, isto é, «PIC 9(6)» equivale a <PIC 999999>, um campo numérico de seis dígitos.

O exemplo prévio do REGISTO-EMPREGADO poderia ser completado adicionando *pictures* do seguinte modo:

```
01      REGISTO-EMPREGADO.
        05      NUMERO-EMP.
                10      DIVISÃO-EMP             PIC AA.
                10      NUM-SERIE-EMP           PIC 9(5).
        05      NOME-EMP                        PIC X(25).
        05      DATA-NASC-EMP
                10      DIA-NASC-EMP            PIC 99.
                10      MES-NASC-EMP            PIC AAA.
                10      ANO-NASC-EMP            PIC 99.
```

Note que os itens de grupo não são *pictures* dadas explicitamente, porque o seu tipo é sempre alfanumérico e o seu tamanho é definido através da agregação das suas partes componentes. Assim, no exemplo acima mencionado, DIVISAO-EMP é um campo alfabético de dois caracteres e NUM–SERIE–EMP é um campo numérico de cinco dígitos; assim, NUMERO-EMP é um campo alfanumérico com sete caracteres. NOME-EMP é descrito como um campo alfanumérico de vinte e cinco caracteres, dado que pode conter outros caracteres que não aqueles definidos como caracteres alfabéticos, isto é, «SCARLET O'HARA», «HARRY S. TRUMAN». DATA-NASC-EMP é um campo alfanumérico de sete caracteres, cujas partes componentes são um dia de dois dígitos, uma abreviatura de três letras para o mês e um ano de dois dígitos. O tamanho de REGISTO-EMPREGADO pode ser então calculado somando todos estes componentes, o que dá um total de trinta e nove caracteres.

Registos de dados possíveis correspondendo a esta descrição são:

```
AA12345WHITE  JOHN                      29FEV48
BB00700BOND   JAMES                     03MAR28
CC87433O'NEILL  JAMES  ALEXANDER       15DEZ35
```

Note que os dados que correspondem aos campos numéricos têm de incluir zeros à esquerda, dado que o espaço não é um carácter numérico válido.

Voltando à estrutura PRINT-REGISTO-EMP definida previamente, esta poderia ser completada do seguinte modo:

```
01      PRINT-REGISTO-EMP.
        05      PRINT-NUM-TRABALHO.
                10      PRINT-DIVISÃO           PIC AA.
                10      FILLER                  PIC X.
                10      PRINT-NUM-SERIE         PIC 9(5).
        05      FILLER                          PIC XXX.
```

```
05      PRINT-NOME          PIC X(25).
05      FILLER              PIC X(6).
05      PRINT-IDADE         PIC 99.
```

e o *output* apareceria com o seguinte formato:

```
AA      12345   WHITE JOHN                  34
BB      00700   BOND JAMES                  54
CC      87433   O'NEILL JAMES ALEXANDER     47
```

Exercícios 3

3.1. Quais dos seguintes nomes são nomes válidos definidos pelo utilizador num programa de COBOL?

a) CONTA-EMPREGADO-1 *b*) 1-TOTAL-EMPREGADO *c*) 1.2.3
d) 1-3-5 e) 1 *f*) PIC
g) CUSTO-TOTAL-10% h) -CONTA-NUM-EMP i) AAA
j) CUSTO-TOTAL-MENOS-DESCONTO-DEZ-PORCENTO

Para aqueles nomes que pensa serem válidos, indique se eles podem ser utilizados como nomes de dados, nomes de procedimento ou ambos.

3.2. A seguinte descrição de registo é tirada de um programa de COBOL. Desenhe a árvore de estrutura correspondente, mostrando a relação entre os campos, e descubra alguns registos de dados que correspondam a esta descrição.

```
01      REGISTO-ESTUDANTE
        05      MATRIC-ESTUDANTE
                10      ANO-MAT-ESTUDANTE               PIC 99
                10      NUMERO-MAT-ESTUDANTE            PIC 9(5)
                10      VER-MAT-ESTUDANTE               PIC X
        05      NOME-ESTUDANTE                          PIC X(30)
        05      ESCOLA-ESTUDANTE
                10      AREA-ESC-ESTUDANTE
                        15      REGIAO-ESC-ESTUDANTE    PIC A.
                        15      DISTRI-ESC-ESTUDANTE    PIC XX
                10      NUM-ESC-ESTUDANTE               PIC 999
        05      FILLER          PIC XX.
        05      CODIGO-GRAU-ESTUDANTE                   PIC 999
```

3.3. Abaixo encontra-se uma descrição narrativa de um registo de *input* para um sistema de controlo de *stocks* que deve ser lido por um programa em COBOL. Transforme esta descrição numa estrutura de registo de COBOL com números de nível e com *pictures*.

A primeira parte do registo contém o número de *stock*, que tem duas componentes: o código do fabricante e a parte do número de série. O primeiro consiste em duas letras, seguidas por três dígitos e, finalmente, um carácter de verificação. A segunda contém oito dígitos. A seguir ao número de *stock* está a parte da descrição, que não excede trinta caracteres. A esta segue-se a localização binária, que consiste em duas letras e quatro dígitos, uma área a espaços de quatro caracteres e o nível de reordenação recomendado, três dígitos.

4. DECLARAÇÕES DE PROCEDIMENTO

Este capítulo dá uma primeira vista de olhos à codificação de algoritmos na *procedure division* de um programa em COBOL. As operações primitivas do desenho de programa introduzidas no capítulo 2 podem ser todas transformadas em *statements* [declarações] ou sequência de *statements* de COBOL. Nesta fase não se espera que o leitor perceba o modo em que as primitivas são implementares — isto será explicado no capítulo 8.

4.1 — A ESTRUTURA DA «PROCEDURE DIVISION»

A unidade de procedimento mais pequena em COBOL, é uma declaração que começa por um verbo, seguido de um ou mais operandos. Os nomes dos verbos foram escolhidos pelos desenhadores do COBOL para indicar as funções das operações correspondentes. O ou os operandos definem o que existe para ser manipulado. Por exemplo:

```
MOVE  PREÇO-COBRADO      TO      PRINT-PREÇO
```

é uma declaração de COBOL, MOVE é o verbo e PREÇO-COBRADO e PRINT-PREÇO são operandos, enquanto TO é um separador entre os operandos, requerido pela sintaxe da declaração *move*. O COBOL é uma linguagem muito «rica», com grande número de verbos, e apenas um pequeno segmento destes verbos será considerado neste livro.

Uma sequência de declarações pode ser combinada para formar uma frase, terminada por um ponto final. No sentido de se poder implementar o refinamento e a repetição, é necessário ter a possibilidade de identificar grupos de declarações formando uma ou mais frases. Sendo assim, a Procedure Division divide-se em *parágrafos,* tendo cada um deles um único nome a si associado. Os parágrafos podem ser reunidos em grandes unidades chamadas sections [secções], mas, de momento, estes não irão ser considerados.

4.2 — OPERAÇÕES BÁSICAS EM COBOL

As operações básicas do desenho de programas foram escolhidas de modo a serem facilmente traduzidas para COBOL. No entanto, há certas regras que devem ser observadas, e estas são salientadas abaixo. Assume-se que a tradução se baseia no esqueleto de programa apresentado no apêndice 1.

Operação ler-próximo [«get-next»]

A operação de desenho de programas:

```
ler-próximo- <nome-ficheiro>
```

traduz-se para a declaração de procedimento:

PERFORM LER-PRÓXIMO-<nomeficheiro>

onde o utilizador substitui o nome do ficheiro de *input* por <nome–ficheiro>. O nome do ficheiro de *input* deve também ser substituído ao longo do esqueleto de programa dado no apêndice 1.

Repare que o verbo PERFORM é aqui exigido e que é obrigatória a existência de hífenes na operação.

Operação escrever [«put»]

escrever–<nome–registo>

traduz-se também facilmente para:

PERFORM ESCREVER-<nome–registo>

em que <nome–registo> é um nome, escolhido pelo utilizador para um registo de *output*. Deve existir uma descrição para <nome–registo> na WORKING-STORAGE SECTION da Data Division e o nome escolhido pelo utilizador deve ser substituído por <nome–registo> no esqueleto de programa.

O uso do verbo PERFORM e o uso de hífenes deve ser agora sublinhado.

Constantes

Em COBOL, uma constante é ou numérica ou não numérica, segundo a notação utilizada para a representar. A constante não numérica é uma série de caracteres delimitados por marcas de citação, como por exemplo:

«A» «PITMAN PUBLISHING» «26 RICHMOND ST.» «12» «1.2.3»

Uma constante numérica não tem marcas de delimitação, é simplesmente um número; quer todo um número <inteiro>, quer um número decimal com ponto decimal. Qualquer constante numérica pode ser precedida de um símbolo. Exemplos de constantes numéricas são dados abaixo.

23 —15 15.5 —2,4 0.672

Note que um número decimal deve ser sempre escrito com, pelo menos, um dígito antes e um depois do ponto decimal; assim, .5 teria de ser 0.5 e 23. seria inválido em COBOL.

Há também constantes específicas especiais chamadas *literais figurativos;* são utilizadas frequentemente para representar constantes usadas. A mais útil dessas constantes é SPACES [espaços], uma constante não numérica que representa um ou mais caracteres a espaços. Esta constante utiliza–se mais frequentemente em registos de *output* que vão ser imprimidos.

Operação mover [«transfer»]

mover <campo–emissor> para <campo–receptor>

traduz–se para COBOL do seguinte modo:

MOVE <campo–emissor> TO <campo–receptor

em que <campo–emissor> deve ser o nome de um campo de dados ou de uma constante e <campo–receptor> deve ser o nome de um campo de dados. O efeito da declaração dependerá da dimensão e dos tipos dos campos envolvidos no MOVE. A principal regra é que <campo–emissor> e <campo–receptor> devem ser sempre do mesmo tipo, para que não haja problemas.

Se apenas itens elementares estiverem envolvidos na declaração MOVE, há então dois casos a considerar:

moves numéricos; aplicam–se estas regras apenas se <campo–emissor> e <campo–receptor> forem ambos numéricos. Neste ponto, apenas valores inteiros serão considerados; valores decimais serão tratados no próximo capítulo. A regra para números inteiros é que os dígitos sejam copiados do lado mais à direita para o lado mais à esquerda até que

ou o <campo–emissor> está totalmente preenchido e então quaisquer posições remanescentes no <campo–receptor> são preenchidas com zeros

ou o <campo–receptor> está totalmente preenchido, e então quaisquer dígitos remanescentes no <campo–emissor> são perdidos.

No último caso, o <campo–receptor> contém uma cópia «truncada» do <campo–emissor>.

Por exemplo, considere um campo de dados definido como se segue.

```
RESULTADO–INTEIRO   PIC 999.
```

Então a declaração MOVE, em baixo, terá os resultados indicados.

```
MOVE 35672 TO RESULTADO–INTEIRO   => 672
MOVE 345 TO RESULTADO–INTEIRO     => 345
MOVE 3 TO RESULTADO–INTEIRO       => 003
```

moves não numéricas; estas regras aplicam–se apenas se <campo–emissor> e <campo–receptor> forem ambos alfanuméricos ou de tipo alfabético. Aqui a regra é que os caracteres são copiados do ponto mais à esquerda para o ponto mais à direita até que

ou o <campo–emissor> está completamente preenchido, e então quaisquer posições remanescentes em <campo–receptor> são preenchidas com espaços,

ou o <campo–receptor> está completamente preenchido, e então quaisquer caracteres remanescentes em <campo–emissor> são perdidos, outro caso de truncagem.

Por exemplo, com um campo de dados:

```
RESULTADO–CARACT   PIC X (3).
```

as seguintes declarações de MOVE darão o resultado indicado:

```
MOVE «RAYMOND» TO RESULTADO–CARACT   => RAY
MOVE «RCW» TO RESULTADO–CARACT       => RCW
MOVE «R» TO RESULTADO–CARACT         => Rbb
```

em que b se utiliza para representar um espaço.

Talvez valha a pena repetir a regra simples: não misture tipos em declarações MOVE. Existem regras que regulam *moves* de tipo misto, mas para o programador principiante o mais razoável é executar *moves* entre campos do mesmo tipo.

É possível existir uma declaração MOVE que envolva itens de grupo; neste caso, o item ou itens de grupo serão tratados como campos não numéricos e aplicar-se-ão as regras acima mencionadas. As principais utilizações para *moves* em grupo são para fazer uma cópia exacta de um campo de dados complexo ou para inicialização. Um exemplo de inicialização é:

```
MOVE SPACES TO LINHA-IMPRESSÃO
```

que preenche, simplesmente, toda a LINHA-IMPRESSÃO com espaços, independentemente de PICs subordinadas nessa linha.

Operação calcular [calculate]

```
calcular <campo-resultado> = <expressão-aritmética>
```

traduz-se directamente para COBOL como:

```
COMPUTE <campo-resultado> = <expressão-aritmética>
```

e convém recordar algumas regras simples. O <campo-resultado> deve ser um item elementar numérico, dado que a <expressão-aritmética> deve produzir um número, quando avaliada.

Se o valor resultante da avaliação de <expressão-aritmética> for demasiado grande para ser representado em <campo-resultado>, então a truncagem pode ocorrer no fim mais significativo, no menos significativo ou em ambos. Considere o exemplo simples:

```
05      CAMPO-RESULTADO    PIC 99.
. . . . . .
COMPUTE CAMPO-RESULTADO = 25 * 25
```

o valor armazenado em CAMPO-RESULTADO, depois desta operação, será 25.

Devem existir espaços à volta dos operandos aritméticos, para que o sinal menos e o hífen possam ser distinguidos. Por exemplo,

```
COMPUTE SALARIO-LIQUIDO = SALARIO-TOTAL — DESCONTOS
```

não equivale a

```
COMPUTE SALARIO-LIQUIDO = SALARIO-TOTAL-DESCONTOS
```

por causa do espaceamento. A primeira declaração subtrai o valor de DESCONTOS ao valor de SALARIO-TOTAL e armazena o resultado em SALARIO-LIQUIDO, enquanto a segunda copia o valor de um item elementar chamado SALARIO-TOTAL-DESCONTOS para SALARIO--LIQUIDO.

Para os leitores que utilizem um compilador ANS CO-BOL que não implemente a declaração COMPUTE dá-se uma tradução alternativa da operação calcular no apêndice 5.1.

Refinamento

Quando se referencia um refinamento no desenho de programa, substitui-se o mesmo por uma declaração de COBOL do tipo:

```
PERFORM <nome-refinamento>
```

e a tradução continua do princípio ao fim do desenho de programa. A tradução do próprio refinamento consiste em criar um parágrafo chamado <nome–refinamento> e em traduzir as operações do refinamento para COBOL.

Exemplo

Para ilustrar a tradução das operações básicas para COBOL, suponha que o seguinte faz parte de um desenho de programa:

```
. . . . .
imprimir–subtotal
ler–próximo–factura
. . . . .
        imprimir–subtotal
                mover 99999 para imprimir–numcon
                mover «subtotal» para imprimir–nomcon
                calcular imprimir–custo–total = vendas–total + vat–total
                escrever–imprimir–vendas
```

assim, a tradução para COBOL apareceria assim:

```
        . . . . .
        PERFORM  IMPRIMIR–SUBTOTAL
        PERFORM  LER–PROXIMO–FACTURA
        . . . . .
PRINT–SUBTOTAL
        MOVE  99999 TO PRINT–NUMCON
        MOVE  «SUBTOTAL» TO PRINT–NOMCON
        COMPUTE  IMPRIMIR–CUSTO–TOTAL = VENDAS + VAT–TOTAL
        PERFORM  ESCREVER–IMPRIMIR–VENDAS
```

Isto é, naturalmente, um exemplo artificial que é apenas um fragmento de um programa para mostrar a tradução das operações básicas. Um exemplo completo será dado após se considerar a selecção e a repetição na secção seguinte. As regras para mostrar as declarações de COBOL são dadas no apêndice 2.

4.3 — SELECÇÃO E REPETIÇÃO

Parte vital de qualquer problema são as estruturas que modificam a execução sequencial e linear de instruções. O refinamento é um exemplo de uma estrutura de controlo simples que permite que uma sequência de instruções inclua uma referência a outra sequência de instruções, etc. No entanto, as duas estruturas de controlo mais importantes são aquelas que permitem uma selecção entre muitos rumos de acção e uma repetição de sequências de declarações. A fim de implementar as operações de selecção e repetição introduzidas na linguagem de desenho de programas, é necessario considerar quais são as condições que podem ser avaliadas como base para a tomada de decisões.

Condições

Uma condição é uma expressão que pode ser avaliada para dar resultado verdadeiro ou falso. As condições simples introduzidas na secção 2.2 eram escolhidas de modo que pudessem ser directamente traduzidas para COBOL e os únicos problemas diziam respeito aos tipos de campos nas condições de comparação.

Em programas de principiantes, o tipo mais comum de comparação é o que envolve dois campos numéricos. Por exemplo, suponha que um campo de dados numéricos, designado por *quantidade–vendas*, vai ser testado para se ver se o seu valor é maior que dez. O desenho de programa poderia incluir a condição:

```
quantidade–vendas > 10
```

que deveria ser directamente traduzida para:

```
QUANTIDADE–VENDAS > 10
```

que compara o valor quantidade–vendas com uma constante numérica e não:

```
QUANTIDADE–VENDAS > «10»
```

a qual envolve uma constante não numérica e que resultará na realização de uma comparação de caracteres. Isto conduz ao resultado aparentemente estúpido de que «5» > «10» é verdadeiro, o que pode ser explicado usando as regras para comparações não numéricas apresentadas em baixo.

Aqui, a regra simples é comparar sempre coisas do mesmo tipo para não causar problemas. No caso de comparações numéricas, os valores de dois membros são comparados através da utilização do comparador especificado para produzir o resultado verdadeiro ou falso. Logo, a condição

```
QUANTIDADE–VENDAS > 10
```

é verdadeira se o valor armazenado em quantidade–vendas for 150 e falsa se o valor aí armazenado for 5 ou 10. A condição

```
CODIGO–CONTA NOT = 12
```

é verdadeira para todos os valores de código–conta, excepto 12. Num desenho de programa é perfeitamente aceitável escrever condições do tipo:

```
preço–factura > 25£
```

mas num programa de COBOL existem apenas constantes numéricas e, assim, a declaração equivalente seria:

```
PREÇO–FACTURA > 25
```

e, implicitamente, o programador deve depreender a unidade monetária ou outras.

Se a comparação for não numérica, os valores armazenados nos campos são comparados do seguinte modo.

a) Se os campos forem de comprimento variável, o menor é alargado adicionando-se-lhe espaços à direita para que fiquem de comprimento igual.

b) Os dois campos são comparados carácter a carácter, da esquerda para a direita, até que se encontrem dois caracteres diferentes ou até que todos os caracteres tenham sido comparados.

c) Se não existirem diferenças entre os valores, então, obviamente, os dois campos são iguais; se o não forem, podem ser arrumados através da relação entre o primeiro par de caracteres diferentes que tenham sido encontrados.

Suponha que existe um campo de dados de *input* definido deste modo:

```
APELIDO–EMPREGADO  PIC  X(10)
```

então a condição

```
APELIDO–EMPREGADO  =  «SMITH»
```

é verdadeira se apelido–empregado contiver SMITHbbbbb, mas é falsa se contiver SMITHSONbb. A condição

```
APELIDO–EMPREGADO  >  «M»
```

é verdadeira se apelido–empregado contiver THOMASbbbb, mas é falsa se contiver BRIDESbbb. Esta condição também é verdadeira se apelido–empregado contiver MAUDSLEYbb, porque A será maior de que o espaço em ordenação alfabética. Determinar o valor de uma comparação não numérica, que envolva caracteres especiais (não–alfabéticos), implicará uma consulta à «sequência comparativa» do computador utilizado.

É provavelmente mais sensato que o programador principiante utilize apenas comparações exactas de campos não numéricos, por exemplo:

```
COD–REF–PART  =  «AB»
TIPO–CONTA NOT = «X»
```

em que COD–REF–PART é um campo de dois caracteres, isto é, PIC XX e em que TIPO–CONTA é um campo de um único carácter, isto é, PIC A.

As condições de ficheiro, do tipo end–of [fim–de]–(nome–ficheiro), são directamente traduzidas para COBOL desde que a estrutura de programa *standard* apresentada no apêndice 1 seja utilizada. Por exemplo:

```
end–of–ficheiro–vendas    =>   END–OF–FICHEIRO–VENDAS
not  end–of–empregado     =>   NOT END–OF–EMPREGADO
```

Uma condição de ficheiro, como END–OF–FICHEIRO–VENDAS, é implementada usando nomes–condição de COBOL, matéria explicada na secção 6.3.

Alguns compiladores ANS COBOL de baixo nível não implementam nomes–condição; no apêndice 5.2. é apresentada uma tradução alternativa.

Selecção

O problema principal da operação de selecção em COBOL é a terminação de uma sequência de operações. Na linguagem de desenho, as sequências são claramente definidas através do uso de else [senão] e de endif [fim se], mas em COBOL utiliza–se o ponto final (.) para terminar a sequência em vez de endif.

Por exemplo, um desenho de programa pode incuir:

```
if tipo–conta = «x» then [então]
        mover «conta especial» para imprimir–mens–cont
        rotina–conta–especial
else
```

```
            mover 0 para desconto
            rotina-conta-normal
endif
escrever-imprimir-cont
ler-proximo-cont
```

Isto deveria ser assim traduzido para COBOL:

```
IF TIPO CONTA = «X»
            MOVE «CONTA ESPECIAL» TO PRINT-MENS-CONT
            PERFORM ROTINA-CONTA-ESPECIAL
ELSE
            MOVE 0 TO DESCONTO
            PERFORM ROTINA-CONTA-NORMAL.
PERFORM ESCREVER-IMPRIMIR-CONT
PERFORM LER-PROXIMO-CONTA
```

O fim da sequência IF da selecção reconhece-se pela palavra ELSE e o fim da sequência ELSE pelo ponto final, a seguir a ROTINA-CONTA-NORMAL. A indentação é uma boa prática de programação e facilita uma compreensão visual, mas não é reconhecida pelo compilador de COBOL.

De modo análogo, a sequência de operações que se segue a if, que não tem o ramo else, também é terminada com um ponto final que representa endif.

Por exemplo:

```
if quantidade-encomendada > 500 then
            calcular desconto = desconto + 0.01
            mover «grandeza» para razão-desconto-1
endif
mover desconto para imprimir-desconto
```

deveria ser traduzido para:

```
IF QUANTIDADE-ENCOMENDADA > 500
            COMPUTE DESCONTO = DESCONTO + 0.01
            MOVE «GRANDEZA» TO PRINT-RAZAO-DESCONTO-1.
TRANSFER DESCONTO TO PRINT-DESCONTO
```

Deve ter-se muito cuidado para garantir que os pontos finais sejam correctamente colocados, porque eles podem ter um grande efeito no sentido de um programa. Por exemplo:

```
IF RENDIMENTO-TRIBUTADO > 11000
            PERFORM TAXA-MAIOR.
PERFORM LER-PROXIMO-EMPREGADO.
```

tem um significado diferente de:

```
IF RENDIMENTO-TRIBUTADO > 11000
            PERFORM TAXA-MAIOR
PERFORM LER-PROXIMO-EMPREGADO.
```

No primeiro exemplo, PERFORM LER-PROXIMO-EMPREGADO é executado qualquer que seja o resultado da selecção, enquanto no segundo exemplo é executado apenas se a condição for verdadeira. A aplicação errada de pontos finais é uma das causas de erro mais comuns entre os principiantes na programação COBOL e, frequentemente, induz também em erro os programadores mais experientes.

O desenho de programa e a sua tradução para COBOL é bastante linear desde que utilizem apenas condições simples e que os pontos finais sejam utilizados correctamente. Uma técnica segura para implementar condições mais complexas envolvendo questões múltiplas consiste em usar declarações if simples e refinamento.

Por exemplo, suponha que um programa tem de seleccionar uma de três rotinas de desconto, dependendo de uma combinação de tipo de conta e da quantidade encomendada. Para registos com o tipo de conta «X» aplica-se o desconto um se a quantidade encomendada exceder 500, senão aplica-se o desconto dois; todos os outros tipos de conta são processados de acordo com o desconto três.

Isto poderia ser representado no desenho de programa do seguinte modo.

```
            if tipo-conta = «X» then
                    rotina-conta-x
            else
                    desconto-três
            endif
            . . .
rotina-conta-x
            if quantidade-encomendada > 500 then
                    descontoum
            else
                    desconto-dois
            endif
```

A tradução para COBOL é bastante linear seguindo as regras apresentadas previamente. Mais para a frente será considerado o uso de outras tecnicas de implementação de condições complexas, tais como declarações if «aninhadas» e condições compostas (ver secções 6.4 e 6.5).

Repetição

A tradução de until, usado no desenho de programa, para COBOL, utiliza uma variação da declaração PERFORM, já encontrada na sua forma mais simples para o refinamento. O until até ao enduntil é substituído por uma declaração PERFOR-MUNTIL do tipo:

```
PERFORM (corpo-ciclo) UNTIL (condição)
```

em que (corpo-ciclo) é um nome de parágrafo escolhido pelo programador ao traduzir o desenho de programa para COBOL. O corpo do ciclo são aquelas operações contidas entre do e enduntil, traduzidas para COBOL como parágrafo <corpo-ciclo>.

Por exemplo, considere a seguinte estrutura do programa total.

```
Sumário-factura
        ler-proximo-factura
        until fim-de-factura do
                processar-factura
                ler-proximo-factura
        enduntil
        imprimir-sumario
```

Isto pode ser traduzido para COBOL como se segue.

```
        SUMARIO-FACTURA.
                PERFORM LER-PROXIMO-FACTURA
```

```
            PERFORM  EXECUTAR-FACTURA  UNTIL  FIM-FACTURA
            PERFORM  IMPRIMIR-SUMÁRIO.
PROCESSAR-FACTURA.
            PERFORM  PROCESSAR-FACTURA
            PERFORM  LER-PROXIMO-FACTURA.
```

O nome EXECUTAR-FACTURA não aparece no desenho de programa, mas é inventado na frase de tradução para dar um nome ao corpo do ciclo.

Este nome adicional é desnecessário se o corpo do ciclo corresponder ao nome de uma operação simples ou de um refinamento. Por exemplo:

```
encontrar-quantidade-total
        mover 0 para quantidade-total
        until campo-numero 9999 do
                actualizar-quantidade-total
        enduntil
```

pode ser traduzido para COBOL como se segue:

```
ENCONTRAR-QUANTIDADE-TOTAL.
        MOVE 0 TO QUANTIDADE-TOTAL
        PERFORM  ACTUALIZAR-QUANTIDADE-TOTAL
                UNTIL CAMPO-NUMERO = 9999.
```

Um compilador ANS COBOL de muito baixo nível pode não incluir PERFOM ... UNTIL e uma tradução alternativa para until é apresentada no apêndice 5.3.

4.4 — **ESTUDO DE CASO**

É agora possível ilustrar o processo completo de desenho e construção de um programa de COBOL simples. Aqui está um problema.

Um ficheiro contém registos de *stock,* contendo, cada um deles, os seguintes campos:

Código *stock*	6 caracteres
Descrição	30 caracteres
Quantidade em depósito	4 dígitos

O código *stock* consiste numa unica letra de código, seguida de um número de código de cinco dígitos.

Escreva um programa que imprima detalhes de todos os registos de *stock* com código de *stock* que consiste na letra que o identifica, «A», seguida de um número de código inferior a 100, ou numa letra diferente de «A» e num número de código inferior a 10. Junte uma mensagem «baixo *stock*» aos detalhes de qualquer registo de *stock* que satisfaça os critérios acima mencionados, em que a quantidade em depósito seja inferior a vinte e cinco artigos.

O processo de desenho de programa pode começar pelo exame dos registos de *input* e de *output*. O ficheiro de *input* pode chamar-se «ficheiro-stock» e cada registo individual pode ser assim descrito:

```
registo-stock
        código-stock
                letra-c-stock
                numero-c-stock
        descrição-stock
        quantidade-stock
```

O código-stock tem de ser dividido nas duas partes que o constituem porque a letra e o número precisam de ser testados separadamente.

O registo de *output* irá requerer quatro campos para incluir a possibilidade de mensagens. Uma descrição possivel é:

```
detalhes
        codigo-detalhes
        descrição-detalhes
        quantidade-detalhes
        mensagem-detalhes
```

Em relação ao *output,* é provavelmente desnecessário dividir o código nas suas partes componentes.

A forma básica do programa é semelhante à de exemplos prévios:

```
selec-stock
        ler-proximo-ficheiro-stock
        until fim-ficheiro-stock do
                processar-registo-stock
                ler-proximo-ficheiro-stock
        enduntil
```

Este progrma irá tratar completamente os registos um por um e usará «processar-registo-stock» para seleccionar os registos que interessam. Assim, o primeiro nível de refinamento pode ser:

```
processar-registo-stock
        if letra-c-stock = «A» then
                verificar-numeros-a
        else
                verificar-outros-numeros
        endif
```

Aqui dividem-se os registos de *stock* em duas categorias: os que têm letra de código «A» e os outros. No nível seguinte existem dois refinamentos que podem ser:

```
verificar-numeros-a
        if numero-c-stock < 100 then
                detalhes-requeridos
        endif
verificar-outros-numeros
        if numero-c-stock < 10 then
                detalhes-requeridos
        endif
```

Isto deixa uma única rotina «detalhes-requeridos» para ser refinada e que deveria construir a linha de detalhe de *output* e, se necessário, juntar a mensagem «baixo *stock*». Um possível refinamento é apresentado em baixo:

```
detalhes-requeridos
        mover codigo-stock para codigo-detalhe
        mover descrição-stock para descrição-detalhe
        mover quantidade-stock para quantidade-detalhe
        if quantidade-stock < 25 then
                mover «baixo stock» para mensagem-detalhe
        endif
        escrever-detalhes
```

Isto completa o desenho de programa. Deve realçar-se que não há uma só solução correcta para nenhum programa, excepto para os problemas de programação mais vulgares. O desenho acima representado não é o único desenho possível e não é necessariamente o melhor. No entanto, existe uma possibilidade de que ele resulte devido à sua construção sistemática.

Nesta fase, o leitor deveria verificar o desenho usando alguns registos de *input* de teste. Aqui estão algumas sugestões para valores de código e de quantidade para verificação de desenho:

Código	Quantidade	Resultado esperado
A00099	0025	seleccionado
A00250	0003	não seleccionado
A00008	0024	seleccionado, stock baixo
A12345	1234	não seleccionado
B00009	0004	seleccionado, stock baixo
Q99999	9999	não seleccionado
X00008	0099	seleccionado
Y00010	0200	não seleccionado

Depois de ter verificado o desenho de programa e de lhe ter feito qualquer alteracão necessária, ele pode agora ser traduzido para COBOL. Não interessa muito o facto de a tradução começar com as descrições dos dados ou com as declarações de procedimento do programa porque, na maioria dos casos, haverá uma certa interacção entre elas à medida que o programa se desenvolve.

Será correcto, neste exemplo, anotar as descrições dos registos de *input* e *output* como ponto de partida.

```
01      REGISTO-STOCK.
        05      CODIGO-STOCK.
                        10      LETRA-C-STOCK           PIC A.
                        10      NUMERO-C-STOCK          PIC 9(5).
        05      DESCRIÇÃO-STOCK                         PIC X(30).
        05      QUANTIDADE-STOCK                        PIC 9999.
01  DETALHES.
        O5      CODIGO-DETALHE                          PIC X(6).
        O5      FILLER                                  PIC XX.
        05      QUANTIDADE-DETALHE                      PIC 9(4).
        05      FILLER                                  PIC X(5).
        05      MENSAGEM-DETALHE                        PIC X(11).
```

O único campo em que há alguma escolha é «MENSAGEM–DETALHE», que tem de ser suficientemente grande para conter qualquer mensagem que tiver de ser imprimida. Neste caso, a única mensagem é «BAIXO STOCK», que irá caber num campo de onze caracteres.

Agora, é possivel trabalhar o desenho de programa sistematicamente, codificando os níveis sucessivos de refinamento. Aplicando as regras apresentadas para a tradução da linguagem de desenho para COBOL, obtêm-se os seguintes fragmentos de programa:

```
SELECT-STOCK
        PERFORM  LER-PROXIMO-FICHEIRO-STOCK
        PERFORM  VERFICAR-STOCK  UNTIL  FIM-DE-FICHEIRO-STOCK.
VERIFICAR-STOCK.
        PERFORM  PROCESSAR-REGISTO-STOCK
        PERFORM  LER-PROXIMO-FICHEIRO-STOCK.

PROCESSAR-REGISTO-STOCK.
        IF LETRA-C-STOCK = «A»
                PERFORM  DETALHES-REQUERIDOS
        ELSE
                PERFORM  VERIFICAR—  OUTROS-NUMEROS.

VERIFICAR-NUMEROS-A
        IF NUMERO-C-STOCK < 100
                PERFORM  DETALHES-REQUERIDOS.

VERIFICAR-OUTROS-NUMEROS.
        IF NUMERO-C-STOCK < 10
                PERFORM  DETALHES-REQUERIDOS.

DETALHES-REQUERIDOS.
        MOVE  CODIGO-STOCK  TO  CODIGO-DETALHE
        MOVE  DESCRIÇÃO-STOCK  TO  DESCRIÇÃO-DETALHE
        MOVE  QUANTIDADE-STOCK  TO  QUANTIDADE-DETALHE
        IF QUANTIDADE-STOCK < 25
                MOVE «BAIXO STOCK» TO MENSAGEM-DETALHE.
        PERFORM  ESCREVER-DETALHES.
```

Para completar o programa, têm de se incorporar estas descrições de dados e os fragmentos de programa no esqueleto de programa apresentado no apêndice 1, e os nomes para o programa, o ficheiro de *input* e o registo de *output* têm de ser sistematicamente inseridos. Com a prática, a fase de tradução e a incorporação no esqueleto podem ser realizadas numa operação.

O programa completo é apresentado em baixo, na sua forma correcta, tal como se especifica no apêndice 2.

```
IDENTIFICATION  DIVISION.
PROGRAM-ID.  EXEMPLO1.

ENVIRONMENT  DIVISION.
CONFIGURATION  SECTION.
SOURCE-COMPUTER.  <nome-computador>.
OBJECT-COMPUTER.  <nome-computador>.
*       <nome-computador> deve ser substituído pelo nome
*       do computador usado para correr programas.

INPUT-OUTPUT  SECTION
FILE-CONTROL.
        SELECT  FICHEIRO-STOCK      ASSIGN  TO  <sistema-input>.
        SELECT  FICHOUT             ASSIGN  TO  <sistema-output>.
*       <sistema-input> e <sistema-output> devem estar conformes
*       com as regras do sistema utilizado.
```

```
DATA DIVISION.
FILE SECTION.
FD      FICHEIRO-STOCK
        LABEL RECORDS ARE OMITTED.
01  REGISTO-STOCK.
        05      CODIGO-STOCK.
                        10      LEIRA-C-STOCK           PIC A.
                        10      NUMERO-C-STOCK          PIC 9 (5).
        05      DESCRIÇÃO-STOCK                         PIC X(30).
        05      QUANTIDADE-STOCK                        PIC 9999.
FD      FICHOUT
        LABEL RECORDS ARE OMITTED.
01      LINHA-OUT                                       PIC X(120).

WORKING-STORAGE SECTION.
01      CAMPOS-TRABALHO.
        05      FIM-FICHEIRO-STOCK                      PIC X.
                88      FIM-FICHEIRO-STOCK    VALUE «E».
01      DETALHES.
        O5      CODIGO-DETALHE                          PIC X(6).
        05      FILLER PIC XX.
        05      DESCRIÇÃO-DETALHE                       PIC X(30).
        05      FILLER PIC XX.
        05      QUANTIDADE-DETALHE                      PIC 9(4).
        05      FILLER PIC X(5).
        05      MENSAGEM-DETALHE                        PIC X(11).

PROCEDURE DIVISION.
PROGRAMA-PRINCIPAL.
        PERFORM ESTADO-INIC
        PERFORM SELECCIONAR-STOCK
        PERFORM FECHO.
ESTADO-INIC.
        MOVE SPACE TO FIM-FICHEIRO-STOCK
        OPEN INPUT FICHEIRO-STOCK OUTPUT FICHOUT
        MOVE SPACES TO DETALHES.
FECHO.
        CLOSE FICHEIRO-STOCK FICHOUT
        STOP RUN.
LER-PROXIMO-FICHEIRO-STOCK.
        READ FICHEIRO-STOCK
                AT END MOVE «E» TO FIM-FICHEIRO-STOCK.
ESCREVER-DETALHES.
        WRITE LINHA-OUT FROM DETALHES
        MOVE SPACES TO DETALHES.
SELECCIONAR-STOCK.
        PERFORM LER-PROXIMO-FICHEIRO-STOCK
        PERFORM VERIFICAR-STOCK UNTIL FIM-DE-FICHEIRO-STOCK.
VERIFICAR-STOCK.
        PERFORM PROCESSAR-REGISTO-STOCK
        PERFORM LER-PROXIMO-FICHEIRO-STOCK.
PROCESSAR-REGISTO-STOCK.
        IF LETRA-C-STOCK = «A»
                PERFORM VERIFICAR-NUMEROS-A
        ELSE
```

```
                PERFORM  VERIFICAR-OUTROS-NUMEROS.
VERIFICAR-NUMEROS-A.
        IF NUMERO-C-STOCK < 100
                PERFORM  DETALHES-REQUERIDOS.
VERIFICAR-OUTROS-NUMEROS.
        IF NUMERO-C-STOCK < 10
                PERFORM  DETALHES-REQUERIDOS.
DETALHES-REQUERIDOS.
        MOVE  CODIGO-STOCK  TO  CODIGO-DETALHE
        MOVE  DESCRIÇÃO-STOCK  TO  DESCRIÇÃO-DETALHE
        MOVE  QUANTIDADE-STOCK  TO  QUANTIDADE-DETALHE
        IF QUANTIDADE-STOCK < 25
                MOVE «BAIXO STOCK» TO MENSAGEM-DETALHE.
        PERFORM  ESCREVER-DETALHES
```

A estrutura detalhada do esqueleto de programa é discutida no capítulo 8 e no apêndice 4 é dado um sumário das declarações de COBOL individuais.

Para testar o programa será necessário construír um ficheiro de dados de teste que pode ser baseado nas sugestões dadas anteriormente. Um ficheiro de *input* amostra possível é:

```
A00099SELECCIONADO                    0025
AOO25ONAO  SELECCIONADO               OOO3
A00008SELECCIONADO-BAIXO  STOCK       0024
A12345NAO  SELECCIONADO               1234
B00009SELECCIONADO-BAIXO  STOCK       0004
Q99999NAO  SELECCIONADO               9999
X00008SELECCIONADO                    0099
YOOO10NAO  SELECCIONADO               0200
```

A descrição do campo no registo de *stock* não é verdadeiramente testada no programa e, assim, pode ser usada para indicar os resultados esperados do caso teste. Quando a combinação do código *stock* e da quantidade em depósito for tal que o registo não deva ser imprimido pelo programa, a descrição do campo é «NAO SELECCIONADO». Se a combinação de valores significar que o registo deve ser imprimido, a descrição do campo é «SELECCIONADO» OU «SELECCIONADO-BAIXO STOCK», quando se espera uma mensagem «BAIXO-STOCK». Isto é mais vantajoso do que encher apenas o campo com dados inválidos.

Se o ficheiro acima mencionado fosse de *input* para o programa «EXEMPLO1», o *output* resultante seria:

```
A00099  SELECCIONADO                  0025
A00008  SELECCIONADO-BAIXO  STOCK     0024    BAIXO  STOCK
B00009  SELECCIONADO-BAIXO  STOCK     0004    BAIXO  STOCK
X00008  SELECCIONADO                  0099
```

Se alguma descrição de *output* contiver as palavras «NAO SELECCIONADO», obviamente o programa foi mal sucedido. Do mesmo modo, se todos os campos que se esperava que fossem seleccionados não forem incluídos no *output,* é porque existe um erro. Em qualquer destes casos, o programa terá de ser examinado, para se encontrar o erro, modificado e executado de novo se se obtiverem, então, resultados correctos. Este processo pode percorrer diversos ciclos e é designado por debugging. A matéria de testes e de *debugging* será discutida mais em detalhe no capítulo 7.

O aspecto do *output* de prova, apresentado acima, poderia ser consideravelmente melhorado com a adição de cabeçalhos nas colunas e suprimindo os zeros à esquerda na coluna da quantidade. A definição de cabeçalhos e a «edição» do *output* são dois dos assuntos tratados no capítulo seguinte.

Exercícios 4

4.1. Suponha que o programa apresentado na secção 4.4 foi incorrectamente carregado no computador e que as três últimas linhas do programa eram

```
IF QUANTIDADE-STOCK < 25
        MOVE «BAIXO STOCK» TO MENSAGEM-DETALHE
PERFORM ESCREVER-DETALHE.
```

Mostre qual seria o *output* do programa para o ficheiro de *input* de prova dado.

Esta situação ilustra um erro típico de principiantes — a falta do ponto final no fim da operação de selecção. Note que, na prática, ser-lhe-á dado o *input* incorrecto e ser-lhe-á exigido que deduza o erro de código, o que é muito mais difícil.

4.2. Um desenho de programa começa com a seguinte estrutura de início:

```
ler-proximo-ficheiro-conta
until fim-de-ficheiro-conta do
        processar-conta
        ler-proximo-ficheiro-conta
enduntil
```

Escreva a tradução deste segmento de desenho de programa para COBOL.

Este segmento de desenho é traduzido do seguinte modo para COBOL, por um programador em formação:

```
PERFORM LER-PROXIMO-FICHEIRO-CONTA
PERFORM PROCESSAR-CONTA UNTIL FIM-DE-FICHEIRO-CONTA
PERFORM LER-PROXIMO-FICHEIRO-CONTA
```

Descreva o que aconteceria se esta codificação incorrecta fosse incluída num programa de COBOL. Pode fazer algumas suposições razoáveis acerca do refinamento «processar–conta».

Exercícios de Programação 4

4.3 Um ficheiro contém registos do formato:

Identificador de cliente	2 letras 6 dígitos
Nome de cliente	20 caracteres
Data da última transação	2 dígitos 3 letras 2 dígitos
Limite de crédito	5 dígitos (libras)

Não há espaços nestes registos e, logo, registos típicos poderiam ser:

```
AA123456PRECISION CASTINGS          03ABR8200500
BC980241NUTS AND BOLTS LTD.         22JAN8308000
DD004836A. WASHER(1892) LTD.        24DEZ8100250
```

Requer-se um programa que produza uma lista destes artigos, com os campos separados por espaços, incluindo a separação das duas partes do identificador de cliente e as três partes da data. Deve juntar-se uma mensagem «limite baixo» para os detalhes de qualquer cliente cujo

limite de crédito seja inferior a 500 libras e uma mensagem «limite alto» para aqueles cujos limites de crédito excedam as 5000. Assim, os três exemplos de registo apresentados acima poderiam aparecer com esta forma:

AA	123456	PRECISION CASTINGS	03 ABR 82	00500	
BC	980241	NUTS AND BOLTS LTD.	22 JAN 83	08000	LIMITE ALTO
DD	004836	A. WASHER (1892) LTD.	24 DEZ 81	00250	LIMITE BAIXO

Pode alargar-se o problema juntando uma segunda mensagem «Não encomendas recentes» para qualquer registo cuja data de última transacção seja anterior a 1 de Janeiro de 1983.

4.4 Uma companhia de cerâmica mantém um ficheiro de encomendas pendentes com a seguinte configuração:

Número de artigo ou artigo requerido	6 dígitos
Identificador de cliente	10 caracteres
Data pedida	6 dígitos
Quantidade de artigo requerida	6 dígitos

Números de artigo com os três primeiros dígitos entre 020 e 070 identificam tipos diferentes de estatuetas de porcelana que exigem técnicas de produção especiais. Mudar a linha de produção representa uma economia apenas se a quantidade total de estatuetas requerida exceder cem.

Desenhe um programa que conte o número de encomendas pendentes de estatuetas e a quantidade total requerida. O *output* do programa precisa apenas de ser constituído por estes números. O próximo capítulo discutirá o modo de adicionar mais detalhes a este *output*.

5 — MAIS «INPUT» E «OUTPUT»

A maior parte dos programas escritos em COBOL estão relacionados com dinheiro, e, logo, é preciso ser capaz de fazer o *input,* de manipular e de fazer o *output* de valores decimais. Estes programas terão também de produzir relatórios que têm de ser apresentados numa forma aceitável para serem lidos pelas pessoas, e, por isso, requerem–se também facilidades para a formação do *output.*

O *input* e o armazenamento interno de valores decimais têm regras semelhantes e serão tratados em primeiro lugar. O *output* de valores decimais faz parte do tópico mais geral da apresentação de *output,* que inclui também a inserção de símbolos monetários e a supressão de zeros. Ao tratar de transacções financeiras, deve ser possível representar tanto os débitos como os créditos, e, assim, os valores com sinais precisam de ser considerados. Para produzir *output* legível é também necessário ter a possibilidade de juntar cabeçalhos ao *output* e, possivelmente, de produzir vários tipos diferentes de linhas de *output.*

5.1 — «INPUT» E ARMAZENAMENTO DE VALORES DECIMAIS

A descrição de um número decimal a que vai fazer–se o *input* num programa de COBOL inclui um ponto decimal implícito. Indica–se a posição do ponto, mas não se aloca uma posição de carácter ao dado de *input* inicial. Um ponto decimal implícito é representado pelo carácter «V» numa *picture.*

Por exemplo,

```
    10      QUANTIA-FACTURA     PIC 9999V99.
ou  10      QUANTIA-FACTURA     PIC 9(4)V99.
```

descreve um número que ocupa seis posições de carácter, com o ponto decimal implícito entre o quarto e o quinto dígitos. Assim, os *inputs* seguintes seriam representados deste modo:

Valor de *input* requerido	Forma inicial do *input*
23.5	002350
1.0	000100
0.03	000003
2172.21	217221

Todas as posições de dígito não utilizadas devem ser preenchidas com zeros e os números devem ser alinhados para que o ponto decimal seja implícito na posição correcta.

Quaisquer zonas internas de armazenamento requeridas para números decimais devem também ser descritas através do carácter que representa o ponto decimal implícito. Por exemplo,

```
10      PREÇO-TOT       PIC 999V99.
```

descreve um campo que pode reter um valor entre 0 e 999.99. Repare que, se um valor que vai ser armazenado neste campo tiver mais dígitos do que os que podem ser representados por esta *picture,* este será truncado para que caiba. A truncagem ocorre frequentemente quando se calculam valores num programa mas se define um espaço insuficiente para o resultado do cálculo. Por exemplo, suponha que existem dois campos,

```
05      QUANTIDADE-VENDA    PIC 99.
05      PREÇO-VENDA         PIC 99V99.
```

e que os seus valores correntes são QUANTIDADE-VENDA = 85 e PREÇO-VENDA = 28.50. O cálculo

```
COMPUTE PREÇO-TOT = QUANTIDADE-VENDA * PREÇO-VENDA
```

dará o resultado de 422.5O em PREÇOTOT porque o resultado está truncado — deveria ser 2422.50.

É essencial que quaisquer campos usados para conter os valores dos cálculos sejam suficientemente grandes para conter o valor máximo possível que pode ser calculado. Muito frequentemente, valores estranhos de *output* são causados não porque o «computador errou», mas devido à truncagem de valores, dada a provisão inadequada de espaço de armazenamento.

A truncagem dos dígitos menos significativos pode também ocorrer particularmente ao calcular fracções de valores. Por exemplo:

```
05 PREÇO-TOT  PIC 999V99.
. . . . . .
COMPUTE PREÇO-TOT = 0.95 * PREÇO-TOT
```

reduziria o valor de PREÇO-TOT em 5%, representando talvez um desconto especial. Suponha que PREÇO–TOT = 125.85 antes do cálculo, e então o novo valor será 119.55, quando o resultado corrente é 119.5575 e os valores menos significativos foram truncados.

Valores arredondados

Nos casos em que os dígitos menos significativos forem perdidos durante o cálculo, é possível indicar que o resultado deve ser arredondado para o valor representável mais próximo, em vez de ser truncado. No exemplo acima, a declaração

```
COMPUTE PREÇO-TOT ROUNDED = 0.95 * PREÇO-TOT
```

daria o resultado de 119.56 se o valor original de PREÇO–TOT fosse 125.85.

As regras para o arredondamento são:

a) O valor calculado é truncado para que caiba no campo resultado;
b) Se o primeiro dígito perdido for maior ou igual a 5, o dígito menos significativo do valor truncado é incrementado numa unidade, o que pode conduzir a outros valores;
c) O valor é armazenado no campo resultado.

Estas regras são também aplicadas aos verbos aritméticos alternativos discutidos no apêndice 5.1.

5.2 — EDIÇÃO DE REGISTOS DE «OUTPUT»

É de desejar que os valores impressos pelo programa sejam representados numa forma aceitável para quem recebe o *output*. As pessoas estão habituadas a ler £23.50 em vez de 002350, que poderia ser a forma «em bruto» deste valor. O processo de conversão de valores de dados internos para uma forma mais aceitável para o uso humano é chamado edição. Os caracteres de edição considerados nesta secção são:

.	ponto decimal
Z	supressão de zeros
*	supressão e substituição de zeros por * (protecção de cheques)
£	símbolo monetário
,	Vírgula (inserida em números grandes)

Ao fazer o *output* de um valor decimal na sua forma correcta, a *picture* deve conter um ponto decimal real. Por exemplo,

```
    05      PRINT-QUANTIA-FACT   PIC 9999.99.
ou  05      PRINT-QUANTIA-FACT   PIC 9(4).99.
```

Repare nas duas utilizações do «.», como ponto decimal e como indicador de fim de *picture* na sintaxe do COBOL. Devido a esta ambiguidade, é importante não deixar um espaço depois de um ponto decimal, porque ele será então tratado como indicador de fim.

Qualquer valor transferido para um campo com esta descrição será automaticamente alinhado pela posição do ponto decimal e ou truncado ou preenchido com zeros para se ajustar à descrição. Para ilustrar o efeito da descrição anterior, considere o efeito dos seguintes exemplos artificiais, com os resultantes *outputs*.

```
MOVE 23 TO PRINT-QUANTIA-FACT        =>    0023.00
MOVE 67.5 TO PRINT-QUANTIA-FACT      =>    0067.50
MOVE 87653.4 TO PRINT-QUANTIA-FACT   =>    7653.40
MOVE 0.5754 TO PRINT-QUANTIA-FACT    =>    0000.57
```

Note que existe uma inconsistência entre o *input* e o *output* de valores decimais — um campo de *output* que pode ser representado por seis caracteres, com um ponto decimal implícito, irá requerer, no mínimo, sete caracteres para *output,* por causa do ponto decimal real.

Outro ponto importante que não pode ser determinado muito frequentemente é que um campo editado é um campo resultado; uma vez que um valor tenha sido editado, não deve tentar-se utilizar esse mesmo valor em nenhum cálculo posterior. O campo PRINT-QUANTIA-FACT, definido acima, poderia ser usado à esquerda do sinal de igual numa declaração COMPUTE ou como campo receptor de uma declaração MOVE. No entanto, não deveria nunca aparecer no lado direito do sinal de igual numa declaração COMPUTE ou campo emissor numa declaração MOVE.

O leitor está acostumado à ideia de que os números são escritos num formato livre sem se precisar de zeros à esquerda a completar o número. Para conseguir este formato «natural» no *output* do programa utiliza-se o carácter «Z» para supressão de zeros nos campos, em vez de alguns ou de todos os «9». Por exemplo.

```
05      PRINT-PRECO    PIC ZZZ9.99.
05      PRINT-QUANTIDADE      PIC Z(4).
```

O «Z» indica que, se a posição correspondente não estiver preenchida com um dígito significativo, ela é preenchida com espaço em vez de um zero. Os exemplos seguintes, também artificiais, ilustram a utilização da supressão de zeros.

```
MOVE 23.6 TO PRINT-PRECO                  =>    ƀƀ23.60
MOVE 1002.03 TO PRINT-PRECO               =>    1002.03
MOVE 67012.305 TO PRINT-PRECO             =>    7O12.3O
MOVE 0.05 TO PRINT-PRECO                  =>    ƀƀƀ0.05
MOVE 206 TO PRINT-QUANTIDADE              =>    b206
MOVE 567842 TO PRINT-QUANTIDADE           =>    7842
MOVE 0 TO PRINT-QUANTIDADE                =>    ƀƀƀƀ
```

É uma boa prática deixar um dígito por suprimir em frente ao ponto decimal, para produzir um zero à esquerda para valores fraccionados, tal como se ilustra no quarto exemplo dado acima. Na maioria dos casos, os símbolos de supressão de zeros à direita do ponto decimal não têm nenhum efeito e devem ser evitados.

O símbolo de protecção de cheques, «*», funciona de modo semelhante ao da supressão de zeros, excepto que todas as posições em branco são preenchidas com asteriscos. Pretende–se isto, tal como o nome sugere, para a impressão de quantias de cheques onde os asteriscos evitam a inserção de dígitos significativos adicionais para propósitos fraudulentos. Repare que a supressão de zeros e a protecção de cheques são mutuamente exclusivos numa determinada *picture*.

O símbolo monetário «£» pode ser utilizado de duas maneiras: como símbolo monetário ou como símbolo móvel. Um valor monetário fixo é uma ocorrência única do símbolo que aparece numa posição fixa do output. Por exemplo:

```
05      PRINT  VENDAS-TOTAL   PIC £Z (4)9.99.
```

poderia ser usado do seguinte modo:

```
MOVE 123.2 TO PRINT-VENDAS-TOTAL          =>    £ƀƀ123.20
MOVE 0.5 TO PRINT-VENDAS-TOTAL            =>    £ƀƀƀƀ0.50
MOVE 656721.458 TO PRINT-VENDAS-TOTAL     =>    £56721.45
```

O símbolo monetário fica na sua posição fixa, apesar da supressão de zeros ou da truncagem.

A posição de um símbolo monetário móvel varia dentro do campo de *output*, sujeita a limites definidos pela *picture*. A descrição:

```
        05      PRINT-VENDAS-TOT     PIC £££££9.99.
ou      05      PRINT-VENDAS-TOT     PIC £(5)9.99.
```

indica que o símbolo monetário pode «flutuar» para qualquer das posições ocupadas por «£», para que preceda imediatamente o dígito mais significativo do valor a que se faz o *output*. Quaisquer posições não utilizadas à esquerda do símbolo monetário que é imprimido são preenchidas com espaços. Os exemplos seguintes ilustram o efeito do símbolo monetário móvel.

```
MOVE 621.5 TO PRINT-VENDAS-TOT            =>    ƀƀ£621.5O
MOVE 0.3 TO PRINT-VENDAS-TOT              =>    ƀƀƀƀ£0.30
MOVE 8265172.831 TO PRINT-VENDAS-TOT      =>    £65172.83
```

Repare que o símbolo monetário não pode mover–se para fora do campo, mesmo que isso signifique a truncagem do valor a ser imprimido.

Não podem usar–se símbolos monetários móveis na mesma *picture* com símbolos de supressão de zeros ou de protecção de cheques.

O último carácter de edição considerado nesta secção é a vírgula. Mais uma vez, é «natural» supor que números grandes sejam escritos com vírgulas para identificar os milhares, os milhões, etc. Podem inserir–se vírgulas nas *pictures* e podem utilizar–se em conjunto com outros símbolos de edição. A picture seguinte dá alguns exemplos da utilização das vírgulas.

```
a)  05   TOTAL-VENDA-SUMARIO      PIC £99,999.99.
b)  05   QUANTIDADE-SUMARIO      PIC ZZ,ZZ9.
c)  05   TOTAL-VAT-SUMARIO       PIC ££££,££9.99.
```

No exemplo *a*), a vírgula aparecerá independentemente do valor a que se vai executar o *output*. Em *b*) e *c*), a vírgula aparecerá apenas se for requerida — se não houver nenhum dígito significativo à esquerda da vírgula, ela será tratada como um «Z» ou como um «£», respectivamente. Aqui estão alguns exemplos da utilização destes campos de dados.

```
MOVE 23.5 TO TOTAL-VENDA-SUMARIO          =>   £00,023.50
MOVE 25 TO QUANTIDADE-SUMARIO             =>   ␣␣␣25
MOVE 1001 TO QUANTIDADE-SUMARIO           =>   b1,001
MOVE 528162 TO QUANTIDADE-SUMARIO         =>   28,162
MOVE 450 TO TOTAL-VAT-SUMARIO             =>   ␣␣␣£450.00
MOVE 18760.23 TO TOTAL-VAT-SUMARIO        =>   ␣£18,760.23
```

A melhor maneira de aprender algo cerca dos caracteres de edição é tentar usá-los em programas. Um programa simples que lesse alguns números de um registo de *input* e que os transferisse para um registo de *output* contendo campos de dados editados ajudaria a clarificar as ideias discutidas nesta secção.

5.3 — NÚMEROS COM SINAL

Uma das áreas indefinidas no *ANS COBOL 74 standard* é a representação exacta de valores de *input* com sinal. A notação da *picture* é suficientemente clara — utiliza-se o símbolo «S» para indicar que um valor tem sinal. Por exemplo:

```
05   MOVIMENTO-STOCK    PIC S999.
05   VALOR-STOCK        PIC S999V99.
```

O «S» indica que estes valores devem ser tratados como tendo sinal, mas não está definido no *standard* como representar, de facto, o registo de *output*. Uma prática comum para fazer o *input* de cartões perfurados é juntar o sinal ao último dígito do número através da multiperfuração; assim, —325 apareceria como 32N ao ser perfurado para *input* como MOVIMENTO-STOCK. No entanto, se se vai executar o *input* de valores de *input* com sinal num programa, deve consultar-se o manual do sistema local.

Quaisquer campos temporários que precisem de sinal utilizam também a notação «S» e o programador não precisa de se preocupar com o modo como o sinal vai ser representado. Convenciona-se que todos os campos sem sinal contêm valores positivos e que, se se transferir um valor negativo para um campo sem sinal, o sinal desaparecerá.

Se é possível fazer o *input* de valores com sinal e retê-los internamente, obviamente que é necessário que haja um modo de representar os sinais no *output*. O modo mais simples de o fazer consiste em usar um sinal de arrasto anexo a uma *picture* que contenha qualquer combinação válida dos sinais de edição descritos na secção prévia. Este sinal de arrasto pode ser «— », «+», «CR» (crédito) ou «DB» (débito), e as acções correspondentes a estes símbolos podem ser assim sumariadas:

Símbolo	*Output* para Valor positivo ou zero	Valor negativo
+	+	—
—	espaço	—
CR	2 espaços	CR
DB	2 espaços	DB

Suponha que existe a variável temporária:

```
04      BALANCO-CORRENTE    PIC S9 (4)V99.
```

poder-se-ia então fazer o *output* dos seus conteúdos via um campo do tipo:

```
        05      PRINT-BALANCO       PIC  ££££9.99DB.
ou      05      CLIENTE-BALANCO     PIC  £*,**9.99—.
```

5.4 — CABEÇALHOS NO «OUTPUT»

A aparência e utilidade do *output* de um programa de computador pode ser substancialmente melhorado através da adição de cabeçalhos, especialmente para *output* tabular que contenha colunas de figuras.

Para ilustrar o processo de desenho de cabeçalhos e da sua impressão de facto, considere o problema simples introduzido na secção 2.3, que, subsequentemente, veio a ser o desenho do programa mostrado no exemplo 2.4, reproduzido em baixo.

```
preço-e-desconto-2
        ler-proximo-vendas
        until fim-vendas do
                encontrar-preço-e-desconto
                produzir-linha-impressora
                ler-proximo-vendas
        enduntil

encontrar-preço-e-desconto
        calcular preço-total = quantidade-vendas * preço–vendas
        if quantidade-vendas > 19 then
        calcular desconto = 0.1 * preçototal
        calcular preçototal = preçototal — desconto
else
        mover 0 para desconto
endif

produzir-linha-impressora
        mover campo-num-vendas para campo-num-impressora
        mover quantidade-vendas para quantidade-impressora
        mover preço-vendas para preço-impressora
        mover preço-total para preço-tot-impressora
        mover desconto para descontoimpressora
        escrever-linha-impressora
```

O *output* deste programa consiste em cinco valores por linha, definidos como se segue:

```
linha-impressora
        item-num-impressora         número do artigo
        quantidade-impressora       quantidade vendida
        preço-impressora            preço do artigo
        preço-tot-impressora        preço total imputado
        desconto-impressora         desconto dado
```

Seria bom ser capaz de produzir *output* sob a forma apresentada na figura 5.1, utilizando cabeçalhos de colunas de edição e adição.

NUMERO	QUANTIDADE	PRECO	PRECO TOTAL	DESCONTO
12345	23	20.25	£465.75	£46.57
21876	5	2.5O	£7.5O	£0.00

Fig. 5.1

Para se conseguir isto, tem de ser possível definir uma linha de cabeçalho, descrevê-la em COBOL e fazer o seu *output* no ponto apropriado do programa. Presumindo-se que o cabeçalho vai aparecer uma vez no princípio do *output,* o sítio apropriado para fazer o *output* do cabeçalho é logo no princípio do processamento. Assim, o primeiro nível do desenho de programa deve ser modificado como se mostra em baixo.

```
preço-e-desconto-3
        escrever-linha-cabeçalho
        ler-proximo-vendas
        until fim vendas do
                encontrar-preço-e-desconto
                produzir-linha-impressora
                ler-proximo-vendas
        enduntil
```

O «escrever-linha-cabeçalho» deve aparecer fora do ciclo until, senão o cabeçalho será repetido em cima de cada linha imprimida.

Ao traduzir este desenho de programa para COBOL, ter-seão de definir dois registos de *output* — um para o cabeçalho e outro para o pormenor. Estes dois registos terão de ser projectados de modo a serem correctamente alinhados.

Neste, exemplo, os primeiros três valores a que se vai fazer o *output* vêm directamente do registo de *input* e são, assim, bem definidos pelo formato de *input.* Suponha que o registo de *input* é assim descrito:

```
01      REGISTO-VENDAS.
          02    NUM-ARTIGO-VENDAS       PIC 9(5).
          02    QUANTIDADE-VENDAS       PIC 99.
          02    PRECO-VENDAS            PIC 999V99.
```

então os campos de *output* poderiam ser definidos da seguinte forma:

```
O5      NUM-ARTIGO-IMPRESSORA           PIC 9(5).
O5      QUANTIDADE-IMPRESSORA           PIC Z9
O5      PRECO-IMPRESSORA                PIC ZZ9.99
```

Os outros dois campos de *output* dependem dos valores de quantidade-vendas e de preço-vendas e poderiam ser descritos como:

```
05      PRECO-TOT-IMPRESSORA            PIC £££,££9.99.
05      DESCONTO-IMPRESSORA             PIC ££££9.99.
```

Agora, que o formato dos campos de *output* foi definido, é necessário desenhar o *layout* [descrição] do *output*. A utilização de papel quadriculado é o ideal para isto; colocar um carácter em cada quadrado do papel e posicionar os campos por baixo dos cabeçalhos, recordando que se deve usar o máximo comprimento possível dos campos de *output*. No exemplo isto conduzirá, provavelmente, a um diagrama com o formato do da figura 5.2.

NÚMERO	QUANTIDADE	PRECO	PRECO TOTAL	DESCONTO
XXXXX	XX	XXX > XX	XXX,XXX.XX	XXXXX.XX
XXXXX	XX	XXX > XX	XXX,XXX.XX	XXXXX.XX

Fig. 5.2

Os comprimentos dos campos FILLER podem ser calculados a partir deste diagrama demonstrativo e pode especificar-se assim o registo de *output* como:

```
01  LINHA-IMPRESSORA.
    05      FILLER                      PIC X.
    05      NUM-ARTIGO-IMPRESSORA       PIC 9(5).
    05      FILLER                      PIC X(7).
    05      QUANTIDADE-IMPRESSORA       PIC Z9.
    05      FILLER                      PIC X(7)
    05      PRECO-IMPRESSORA            PIC ZZ9.99.
    05      FILLER                      PIC XXXX.
    05      PRECO-TOT-IMPRESSORA        PIC £££, ££9.99.
    05      FILLER                      PIC XXX.
    05      DESCONTO-IMPRESSORA         PIC ££££.9.99.
```

Agora, a correspondente linha de cabeçalho tem de ser criada, introduzindo-se assim uma nova ideia — a «cláusula VALUE».

Cláusulas «Value»

A qualquer campo temporário pode ser dado um valor inicial usando a cláusula VALUE do COBOL. Por exemplo:

```
05      REPORT-CABECALHO            PIC X(15)
        VALUE «LUCRUS E PERDAS».
```

Isto atribui o valor «LUCROS E PERDAS» ao campo chamado REPORT-CABECALHO. Este será o seu valor no início da execução do programa e só será alterado por uma declaração como por exemplo:

```
MOVE «FOLHA BALANCO» TO REPORT-CABECALHO
```

Esta instrução irá modificar o valor de REPORT-CABECALHO e o valor inicial será perdido. Pode ainda ser atribuído a um campo um valor inicial de espaço utilizando a constante «SPACES». Por exemplo:

```
05      FILLER PIC X(10)        VALUE SPACES.
```

iria criar um campo, com o nome especial FILLER, contendo dez espaços. Isto poderia não fazer sentido isoladamente, mas poderia ser muito útil numa estrutura de dados maior.

No caso especial de uma linha de cabeçalho, estes campos não necessitarão de ser alterados. Assim, todos os campos que irão formar o registo cabeçalho podem ter um nome FILLER e, como partes individuais do cabeçalho, não irão ser acedidos.

Seria possível, utilizando a cláusula VALUE, criar uma única constante monolítica para toda a linha de cabeçalho. No entanto, a exeriência mostrou que isso conduz a problemas quando for necessário executar pequenos ajustes no cabeçalho. Por isso, recomenda-se que todos os cabeçalhos sejam repartidos em pequenas unidades que possam ser alteradas individualmente. Uma possível descrição de cabeçalho é dada abaixo.

```
01      LINHA-CABECALHO.
          02      FILLER  PIC X(8)        VALUE  «NUMERO»
          02      FILLER  PIC XX          VALUE  SPACES.
          02      FILLER  PIC X(8)        VALUE  «QUANTIDADE».
          02      FILLER  PIC XX          VALUE  SPACES.
          02      FILLER  PIC X(10)       VALUE  «PRECO».
          02      FILLER  PICXX           VALUE  SPACES.
          02      FILLER  PIC X(11)       VALUE  «PRECO  TOTAL».
          02      FILLER  PIC XX          VALUE  SPACES.
          02      FILLER  PIC X(8)        VALUE  «DESCONTO».
```

O diagrama mostrado na figura 5.1 indica que a LINHA–CABECALHO e a LINHA–IMPRESSORA podem ser do mesmo comprimento, cinquenta e três caracteres. Uma contagem de números de caracteres em cada um destes registos mostra que ambos totalizam cinquenta e três caracteres. Se os totais não coincidirem, devem confrontar-se as descrições dos registos ajustando os comprimentos dos seus campos.

Logo que o programa inicie a produção de resultados, as alterações menores nas posições relativas de cabeçalhos e de campo serão visíveis. No entanto, o esforço feito no desenho inicial do *output* é, normalmente, mais do que compensado pela redução do número de tais modificações quando o programa é testado.

Tendo desenhado e descrito a linha de cabeçalho, é ainda necessário executar o seu *output* para impressora no local apropriado do programa. No programa desenhado eis a operação:

```
escrever-linha-cabeçalho
```

que traduzida para COBOL será:

```
PERFORM    ESCREVER-LINHA-CABECALHO
```

Deve então adicionar-se uma especificação de ESCREVER–LINHA–CABECALHO após ESCREVER–LINHA–IMPRESSORA no esqueleto do programa.

```
ESCREVER-LINHA-IMPRESSORA.
        WRITE  LINHA-0UT  FROM  LINHA-IMPRESSORA
        MOVE  SPACES  TO  LINHA-IMPRESSORA.
ESCREVER-LINHA-CABECALHO.
        WRITE  LINHA-OUT  FROM  LINHA-CABECALHO.
```

Note a diferença entre as duas operações de escrita, O formato geral de uma operação de escrita de uma linha constante para *ouput* é:

```
ESCREVER-<linha-constante>.
        WRITE  LINHA-OUT  FROM  <linha-constante>.
```

Isto acontece porque a <linha-constante> não precisa de ser limpa para posterior utilização.

O registo LINHA-OUT é a área de *output* do programa e deve ser descrita, como parte da descrição do ficheiro de *output,* na FILE SECTION da DATA DIVISION. Uma descrição de LINHA-OUT está incluída no esqueleto de programa no apêndice 1. Os registos LINHA-IMPRESSORA e LINHACABECALHO devem ser descritos na WORKINGSTORAGE SECTION do programa.

Frequentemente, torna-se necessário introduzir linhas em branco no *output* para o tornar mais legível. Isto pode ser feito através da definição de uma <linhaconstante> com o seguinte formato:

```
01      LINHA-EM-BRANCO      PIC X   VALUE  SPACE.
```

A operação de escrita correspondente seria, então, assim descrita:

```
ESCREVER-LINHA-EM-BRANCO.
        WRITE  LINHA-OUT  FROM  LINHA-EM-BRANCO.
```

Então sempre que tenha de se efectuar o *output* de uma linha em branco, é só necessário:

```
PERFORM  ESCREVER-LINHA-EM-BRANCO.
```

Note que todos os registos de *output* que serão escritos na impressora terão os seus comprimentos restringidos ao comprimento da linha de impressão. As mais usuais são as de cento e vinte caracteres, que é o convencionado em todos os exemplos deste livro. Registos que contenham um número de caracteres inferior ao máximo permitido são completados com espaços no decurso das operacões de escrita (a tratar em MOVE não numéricos). Esta é a razão por que LINHAS-EM-BRANCO pode somente ser definida como uma constante contendo um só espaço.

5.5 — REGISTOS MÚLTIPLOS DE «OUTPUT»

O *output* de cabeçalhos descrito na secção anterior é um caso especial do problema da definição de *output* múltiplo de registos de um dado ficheiro de *output.* Na generalidade, existem três tipos de registos de *output,* que podem ser identificados pelo seu conteúdo:

a) Somente dados variáveis, tais como as típicas linhas de detalhe utilizadas em exemplos anteriores:

b) Somente constantes, tais como cabeçalhos e linhas em branco;

c) Uma mistura de constantes e variáveis, tais como um cabeçalho contendo número de página.

Nesta secção será desenvolvido um grupo de regras simples para utilização num ficheiro de *output* que requeira uma mistura de qualquer número de registos deste tipo.

Para registos que só contenham dados variáveis, cada registo definido deveria ter uma operação de escrever-<nome-registo> criada para ele. Esta operação deve seguir o formato mostrado no apêndice 1:

```
ESCREVER-<nomeregisto>.
        WRITE  LINHA-OUT  FROM  <nome-registo>
        MOVE  SPACES  TO  <nome-registo>.
```

Esta construção garante que os conteúdos de <nome-registo> são sempre limpos para a construção da próxima linha de output.

Para garantir que o estado inicial de cada registo está correcto, cada registo dever ser colocado a espaços no fim do parágrafo ESTADO–INIC, no esqueleto *standard* do programa.

Por exemplo, suponha que um programa deveria efectuar o *output* de duas linhas diferentes de detalhe chamadas «linha–det–1» e «linha–det–2». Parte do desenho de programa poderia ser:

Exemplo 5.1

```
if tipo-custo < 10 then
        imprimir-linha-1
else
        imprimir-linha-2
endif
. . .
imprimir-linha-1
. . .
escrever-linha-det-1

imprimir-linha-2
. . .
escrever-linha-det-2
```

O programa de COBOL correspondente conteria descrições para LINHA–DET–1 e LINHA–DET–2 e o esqueleto do programa poderia ser modificado como se segue:

```
PROCEDURE DIVISION.
MAIN-PROGRAM.
        PERFORM ESTADO-INIC
        PERFORM <processamento-principal>
        PERFORM FECHO.
ESTADO-INIC.
        MOVE SPACE TO FIM-<nome-ficheiro-input>
        OPEN INPUT <nome-ficheiro-input> OUTPUT FICHOUT
        MOVE SPACES TO LINHA-DET-1 LINHA-DET-2.
FECHO.
        CLOSE <nome-ficheiro-input> FICHOUT
        STOP RUN.
LER-PROXIMO-<nome-ficheiro-input> FICHOUT
        READ <nome-ficheiro-input>
                AT END MOVE «E» TO FIM-<nome-ficheiro-input>.
ESCREVER-LINHA-DET-1.
        WRITE LINHA-OUT FROM LINHA-DET-1
        MOVE SPACES TO LINHA-DET-1.
ESCREVER-LINHA-DET-2.
        WRITE LINHA-OUT FROM LINHA-DET-2
        MOVE SPACES TO LINHA-DET-2.
```

O fragmento de desenho de programa mostrado no exemplo 5.1 poderia ser traduzido para COBOL da seguinte forma:

```
EXEMPLO-5-1.
        IF TIPO-CUSTO < 10
                PERFORM IMPRIMIR-LINHA-1
        ELSE
                PERFORM IMPRIMIR-LINHA-2.
```

```
. . .
IMPRIMIR-LINHA-1.
        . . .
        PERFORM  ESCREVER-LINHA-DET-1
IMPRIMIR-LINHA-2.
        . . .
        PERFORM  ESCREVER-LINHA-DET-1.
```

Num outro programa poderiam existir duas linhas de informação a ser escritas ao mesmo tempo e o desenho de programa poderia conter:

Exemplo 5.2

```
preencher-linhas-det
escrever-linha-det-1
escrever-linha-det-2
```

Neste exemplo convenciona-se que «preencher-linhas-det» transfere dados para «linha-det-1» e «linha-det-2». Traduzindo este fragmento de desenho para COBOL, teríamos:

```
EXEMPLO-5-2.
        PERFORM  PREENCHER-LINHAS-DET
        PERFORM  ESCREVER-LINHA-DET-1.
        PERFORM  ESCREVER-LINHA-DET-2.
```

Como LINHA-DET-1 e LINHA-DET-2 são independentes, podem ser prenchidas ao mesmo tempo e sair em sequência para o *output*.

As regras para registos contendo apenas constantes foram apresentadas na secção anterior. Não há necessidade de modificar ESTADO-INIC e a operação escrever-<linha-constante> é definida como:

```
ESCREVER-<linha-constante>.
        WRITE  LINHA-OUT  FROM  <linha-constante>.
```

Quando se trabalha com registos contendo quer constantes, quer variáveis, estes serão tratados como casos especiais da linha constante, Porém, um ponto importante a recordar é que o programador é responsavel por assegurar que as partes variáveis do registo são correctamente inicializadas e reformatadas todas as vezes que o registo é utilizado.

Por exemplo, considere a descrição de registo que se segue para um cabeçalho de página.

```
01      CABECALHO-PRINCIPAL.
        02      FILLER                  PIC X(47)       VALUE  SPACES.
        02      FILLER                  PIC X(23)
                VALUE «CONTA LUCROS E PERDAS».
        02      FILLER                  PIC X(47)       VALUE  SPACES.
        02      NUM-PAG-PRINC           PIC ZZ9.
```

A operação escrever-cabeçalho-principal deveria ser definida como segue:

```
ESCREVER-CABEÇALHO-PRINCIPAL.
        WRITE  LINHA-OUT  FROM  CABECALHO-PRINCIPAL.
```

Sempre que esta operação é utilizada, deve-se atribuir um valor a NUM–PAG–PRINC, por exemplo:

```
PRINT-CABECALHO.
        MOVE CONT-PAG TO NUM-PAG-PRINC
        PERFORM ESCREVER-CABECALHO-PRINCIPAL.
```

5.6 — ESTUDO DE CASO

É agora possível escrever um programa completo de COBOL para o estudo do caso introduzido na secção 2.5. A especificação para o programa era:

> A ABC Carpet Company mantém um ficheiro de encomendas diárias em que cada registo contém: um número de encomenda, o nome e morada do cliente, o número de metros quadrados de alcatifa encomendada, o preço por metro quadrado e a zona de entrega. A zona de entrega está codificada com uma simples letra «A» ou «B», indicando a distância a partir do armazém. A política da companhia relativamente aos custos das entregas é que a encomenda seja entregue grátis se o custo da alcatifa exceder as 200 libras, caso contrário existe um preço adicional de 5 libras para a zona «A» e de 10 libras para a zona «B».
>
> Requer-se um programa que irá processar as encomendas do ficheiro e calcular o preço da alcatifa, o preço da entrega, se existir, e o total a ser cobrado por cada encomenda. Após processar todas as encomendas, o programa deverá produzir os totais de venda da alcatifa, os custos de entrega e o total dos dois.

O desenho de programa desenvolvido na secção 2.5 vai ser ligeiramente modificado porque vai agora ser possível incluir cabeçalhos para o *output* e a escrita de um registo separado contendo os totais.

Para incluir cabeçalhos, ir-se-á incluir no nível superior do desenho de programas um refinamento chamado «print–cabeçalhos» fora do ciclo principal de processamento. Assim, o nível superior do desenho de programa seria:

```
encomendas-alcatifa
        inicializar-totais
        print-cabeçalhos
        ler-proximo-ficheiro-encomenda
        until fim-de-ficheiro-encomenda do
                processar-encomenda
                actualizar-totais
                ler-proximo-ficheiro-encomenda
        enduntil
        produzir-totais
```

Neste caso, seria apropriado ter um cabeçalho onde constasse o nome da companhia e cabeçalhos para as linhas de detalhe do *output*. Se tiverem de existir linhas em branco a separar os cabeçalhos, o refinamento de «print–cabeçalhos» seria:

```
print-cabeçalhos
        escrever-cabeçalho-pagina
        escrever-linha-em-branco
        escrever-cabeçalho-detalhe
        escrever-linha-em-branco
```

Na Data Division do programa de COBOL, serão necessárias definições para «cabeçalho-–página», «cabeçalho–detalhe» e «linha–em–branco» e as linhas correspondentes das operações de escrita terão de ser adicionadas à Procedure Division.

Neste estado, presume-se que só existe um tipo único de cabeçalhos e que o *output* não necessitará de página de continuação com novos cabeçalhos. Os problemas de paginação serão discutidos no capítulo 8.

A versão original de «produzir-totais», dada na secção 2.5, previa que o *output* do sumário iria utilizar o mesmo registo que as linhas de detalhe. Nesta secção, o *output* dos totais será executado num registo de output separado, «linha–sumário». Este registo irá conter uma mistura de constantes e de variáveis; a parte constante será uma mensagem adequada e a parte variável os valores dos três totais a serem escritos.

Assim, o novo registo de *output* seria:

```
linha-sumario
        TOTAIS TODAS ENCOMENDAS     mensagem fixa
        custos-alcatifa-sumario     total custos alcatifas
        custos-entrega-sumario      total custos todas entregas
        custos-encomenda-sumário    total todos custos
```

Separar a linha de sumário do resto do detalhe através de uma linha em branco iria também melhorar o aspecto do *output*. Assim, o novo refinamento de «produzir-totais» é:

```
produzir-totais
        escrever-linha-em-branco
        mover alcatifa-total para custo-alcatifa-sumario
        mover entrega-total para custo-entrega-sumario
        calcular custo-encomenda-sumario = alcatifa-total + entrega-total
        escrever-linha-sumario
```

Deve adicionar-se então à Procedure Division do programa um parágrafo contendo «escrever-linha-sumario», recordando que este tipo de registo irá conter uma parte constante e uma parte variável.

Segue-se um desenho completo do programa, já com estas modificações:

```
encomendas-alcatifa
        inicializar-totais
        print-cabeçalhos
        ler-proximo-ficheiro-encomenda
        until fim-de-ficheiro-encomenda do
                processar-encomenda
                actualizar-totais
                ler-proximo-ficheiro-encomenda
        enduntil
        produzir-totais

inicializar-totais
        mover 0 para alcatifa-total
        mover 0 para entrega-total

print-cabeçalhos
        escrever-cabeçalhos-pagina
        escrever-linha-em-branco
        escrever-cabeçalho-detalhe
        escrever-inha-em-branco
```

```
processar-encomenda
        calcular-custo-alcatifa = tamanho-encomenda * preço-encomenda
        if preço-alcatifa not > 200 then
                encontrar-preço-entrega
        else
                mover 0 para preço-entrega
        endif
        print-detalhe-encomenda

actualizar-totais
        calcular alcatifa-total = alcatifa-total + custo-alcatifa
        calcular entrega-total = entrega-total + custo-entrega

produzir-totais
        escrever-linha-em-branco
        mover alcatifa-total para custo-alcatifa-sumario
        mover entrega-total para custo-entrega-sumário
        calcular custo-encomenda-sumario = alcatifa-total + entrega-total
        escrever-linha-sumario

encontrar-custo-entrega
        if zona-encomenda = «A» then
                mover 5 para custo-entrega
        else
                mover 10 para custo-entrega
        endif

print-detalhe-encomenda
        mover numero-encomenda para numero-detalhe
        mover cliente-encomenda para cliente-detalhe
        mover tamanho-encomenda para tamanho-detalhe
        mover preço-encomenda para preço-detalhe
        mover zona-encomenda para zona-detalhe
        mover custo-alcatifa para custo-alcatifa-detalhe
        mover custo-entrega para custo-entrega-detalhe
        calcular custo-encomenda-detalhe =
        custo-alcatifa + custo-entrega
        escrever-linha-detalhe
```

Antes de tentar traduzir este desenho para COBOL, o formato do registo de *input* deve ser definido. Uma descrição possível para este registo é apresentada em baixo.

Número encomenda	4 dígitos
Nome e morada cliente	30 caracteres
Tamanho alcatifa (metros quad.)	2 dígitos
Preço por metro quadrado	4 dígitos (pénis)
Zona entrega	1 letra («A» ou «B»)

O programa completo para este desenho pode, a partir de agora, ser desenvolvido.

```
IDENTIFICATION  DIVISION.
PROGRAM-ID.  ALCATIF.
ENVIRONMENT  DIVISION.
CONFIGURATION  SECTION.
SOURCE-COMPUTER.  IBM-S34.
```

```
OBJECT-COMPUTER.  IBM-S34.
*         <nome-computador> deve ser substituído pelo nome
*         do computador usado para correr os programas.
INPUT-OUTPUT  SECTION.
FILE-CONTROL.
          SELECT  FICHENC  ASSIGN  TO  DISK-FICH-ENC.
          SELECT  FICHOUT  ASSIGN  TO  PRINTER-FICHOUT.
*         <sistema-input> e <sistema-output> devem estar conformes
*         com as regras do sistema utilizado.
DATA  DIVISION.
FILE  SECTION.
FD        FICH-ENC
          LABEL  RECORDS  ARE  STANDARD.
01        REGISTO-ENCOMENDA
          05        NUMERO-ENCOMENDA              PIC  9999.
          05        CLIENTE-ENCOMENDA             PIC  X(30).
          05        TAMANHO-ENCOMENDA             PIC  99.
          05        PRECO-ENCOMENDA               PIC  99V99.
          05        ZONA-ENCOMENDA                PIC  A.
FD        FICHOUT
          LABEL  RECORDS  OMITTED.
01        LINHA-OUT                               PIC  X(120).
WORKING-STORAGE  SECTION.
01        VECTOR-ESTADO.
          05        FIM-FICHEIRO-ENCOMENDA        PIC  X.
                    88        FIM-DE-FICHEIRO-ENCOMENDA  VALUE  «E».
01        CAMPOS-TEMPORARIOS.
          05        ALCATIFA-TOTAL                          PIC  9(6)V99.
          05        ENTREGA-TOTAL                           PIC  999.
          05        CUSTO-ENTREGA                           PIC  99.
          05        CUSTO-ALCATIFA                          PIC  9(4)V99.
01        CABECALHOS-PAGINA.
          05        FILLER  PIC  X(36)        VALUE  SPACES.
          O5        FILLER  PIC  X(18)        VALUE  «ABC  CARPET  COMPANY».
01        LINHA-EMBRANCO        PIC  X        VALUE  SPACE.
01        CABECALHOS-DET.
          05        FILLER  PIC  X(6)         VALUE  «NUMERO».
          O5        FILLER  PIC  XXXX         VALUE  SPACES.
          05        FILLER  PIC  X(25)        VALUE
                    «NOME  E  MORADA  DO  CLIENTE».
          05        FILLER  PIC  X(5)         VALUE  SPACES.
          O5        FILLER  PIC  XXXX         VALUE  «TAM.».
          05        FILLER  PIC  XX           VALUE  SPACES.
          05        FILLER  PIC  X(5)         VALUE  «PRECO».
          O5        FILLER  PIC  XX           VALUE  SPACES.
          05        FILLER  PIC  XXXX         VALUE  «ZONA».
          05        FILLER  PIC  X(5)         VALUE  SPACES.
          05        FILLER  PIC  X(6)         VALUE  «ALCAT.».
          05        FILLER  PIC  XX           VALUE  SPACES.
          05        FILLER  PIC  X(5)         VALUE  «ENTR.».
          05        FILLER  PIC  X(7)         VALUE  SPACES.
          05        FILLER  PIC  X(5)         VALUE  «TOTAL».
01        LINHA-DET.
          05        FILLER                    PIC  X.
          05        NUMERO-DETALHE            PIC  9(4).
```

```
    05      FILLER                          PIC XXX.
    05      CLIENTE-DETALHE                 PIC X(30).
            05      FILLER                  PIC XXX.
            05      TAMANHODETALHE          PIC Z9.
            05      FILLER                  PIC XXX.
            05      PRECODETALHE            PIC Z9.99.
            05      FILLER                  PIC XXX.
            05      ZONADETALHE             PIC A.
            05      FILLER                  PIC X(6).
            05      CUSTO-ALCATIFA-DETALHE  PIC ££££9.99.
            05      FILLER                  PIC XXX.
            05      CUSTO-ENTREGA-DETALHE   PIC ££9.
            05      FILLER                  PIC X(6).
            05      CUSTO-NCOMENDA-DETALHE  PIC £(5)9.99.
01  LINHA-SUMARIO.
            05      FILLER                  PIC X(18)  VALUE SPACES.
            05      FILLER                  PIC X(23)  VALUE
                    «TOTAIS TODAS ENCOMENDAS».
            05      FILLER                  PIC X(19)  VALUE SPACES.
            05      CUSTO-ALCATIFA-SUMARIO  PIC ££££,££9.99.
            05      FILLER                  PIC XX     VALUE SPACES.
            05      CUSTO-ENTREGA-SUMARIO   PIC £££9.
            05      FILLER                  PIC XXXX   VALUE SPACES.
            05      CUSTO-ENCOMENDA-SUMARIO PIC ££££,££9.99.
    PROCEDURE DIVISION.
    PROGRAMA-PRINCIPAL.
            PERFORM ESTADO-INIC
            PERFORM ENCOMENDA-ALCATIFA
            PERFORM FECHO.
    ESTADO-INIC
            MOVE SPACE TO FIM-FICHEIRO-ENCOMENDA
            OPEN INPUT FICH-ENC OUTPUT FICHOUT
            MOVE SPACES TO LINHA-DET.
    FECHO.
            CLOSE FICH-ENC FICHOUT
            STOP RUN.
    LER-PROXIMAO-FICHEIRO-ENCOMENDA.
            READ FICH-ENC
                    AT END MOVE «E» TO FIM-FICHEIRO-ENCOMENDA.
    ESCREVER-DETALHES.
            WRITE LINHA-OUT FROM LINHA-DET
            MOVE SPACES TO LINHA-DET
    ESCREVER-CABECALHOS.
            WRITE LINHA-OUT FROM CABECALHOS-PAGINA.
    ESCREVER-LINHA-EM-BRANCO.
            WRITE LINHA-OUT FROM LINHA-EM-BRANCO.
    ESCREVER-CABECALHOS-DET.
            WRITE LINHA-OUT FROM CABECALHOS-DET.
    ESCREVER-LINHA-SUMARIO.
            WRITE LINHA-OUT FROM LINHA-SUMARIO.
    ENCOMENDA-ALCATIFA.
            PERFORM INICIALIZAR-TOTAIS
            PERFORM PRINT-CABECALHOS
            PERFORM LER-PROXIMO-FICHEIRO-ENCOMENDA
    PERFORM TRATAR-ENCOMENDA UNTIL FIM-DE-FICHEIRO-ENCOMENDA
```

```
        PERFORM  PRODUZIR-TOTAIS.
INICIALIZAR-TOTAIS.
        MOVE 0 TO ALCATIFA-TOTAL
        MOVE 0 TO ENTREGA-TOTAL.
PRINT-CABECALHOS.
        PERFORM  ESCREVER-CABECALHOS
        PERFORM  ESCREVER-LINHA-EM-BRANCO
        PERFORM  ESCREVER-CABECALHOS-DET
        PERFORM  ESCREVER-LINHA-EM-BRANCO.
TRATAR-ENCOMENDA.
        PERFORM  PROCESSAR-ENCOMENDA
        PERFORM  ACTUALIZAR-TOTAIS
        PERFORM  LER-PROXIMO-FICHEIRO-ENCOMENDA.
PROCESSAR-ENCOMENDA.
        COMPUTE  CUSTO-ALCATIFA =
                TAMANHO-ENCOMENDA * PRECO-ENCOMENDA
        IF CUSTO-ALCATIFA NOT < 200
                PERFORM  ENCONTRAR-CUSTO-ENTREGA
        ELSE
                MOVE 0 TO CUSTO-ENTREGA.
        PERFORM  PRINT-ENCOMENDA-DETALHE.
ACTUALIZAR-TOTAIS.
        COMPUTE  ALCATIFA-TOTAL =
                ALCATIFA-TOTAL + CUSTO-ALCATIFA
        COMPUTE  ENTREGA-TOTAL =
                ENTRE-GATOTAL + CUSTO-ENTREGA.
PRODUZIR-TOTAIS
        PERFORM  ESCREVER-LINHA-EM-BRANCO
        MOVE  ALCATIFA-TOTAL  TO  CUSTO-ALCATIFA-SUMARIO
        MOVE  ENTREGA-TOTAL  TO  CUSTO-ENTREGA-SUMARIO
        COMPUTE  CUSTO-ENCOMENDA-SUMARIO =
                ALCATIFA-TOTAL + ENTREGA-TOTAL
        PERFORM  ESCREVER-LINHA-SUMARIO
ENCONTRAR-CUSTO-ENTREGA.
        IF ZONA-ENCOMENDA = «A»
                MOVE 5 TO CUSTO-ENTREGA
        ELSE
                MOVE 10 TO CUSTO-ENTREGA.
PRINT-ENCOMENDA-DETALHE.
        MOVE  NUMERO-ENCOMENDA  TO  NUMERO-DETALHE
        MOVE  CLIENTE-ENCOMENDA TO CLIENTE DETALHE
        MOVE  TAMANHO-ENCOMENDA  TO  TAMANHO-DETALHE
        MOVE   ZONA-ENCOMENDA  TO  ZONA-DETALHE
        MOVE   CUSTO-ALCATIFA  TO  CUSTO-ALCATIFA-DETALHE
        MOVE   CUSTO-ENTREGA  TO  CUSTO-ENTREGA-DETALHE
        COMPUTE  CUSTO-ENCOMENDA-DETALHE =
                CUSTO-ALCATIFA + CUSTO-ENTREGA
        PERFORM  ESCREVER-DETALHES.
```

Um exemplo do *output* deste programa é mostrado na figura 5.3.

NUMERO	NOME E MOR. DO CLI.	TAM.	PRECO	ZONA	ALCAT.	ENTR.	TOTAL
1514	J.STUART, FLODEN	25	15.50	A	£387.50	£0	£387.50
1314	R.BRUCE, BANNOCKBURN	10	5.60	A	£56.00	£5	£61.00
1297	W.WALLACE, STL.BRIDGE	10	20.00	A	£200.00	£5	£205.00
1645	J.GRAHAM, INVERLOCHY	20	10.01	B	£200.20	£0	£200.20
1388	J.DOUGLAS, OTTERBURN	20	14.40	B	£288.00	£0	£288.00
1745	C.STUART, PRESTONPANS	15	12.90	B	£193.50	£10	£203.50
1689	J.GRAHAM, KILLIECRAN.	20	10.00	B	£200.00	£10	£210.00
	TOTAIS TODAS ENCOMENDAS				£1,525.20	£30	£1,555.20

Fig. 5.3

Exercício 5

5.1 O registo de *input* de um programa de COBOL contém o seguinte campo:

```
05      SALARIO-BASE-MENSAL PIC 9(4)V99
```

Escreva a forma como iria carregar, neste campo, os valores que se seguem: 23, 27.5, 133.42 e 2,125.

5.2. Um programa contém as seguintes descrições de dados:

```
01      CAMPOS-TEMPORAIS
        05      DESCONTO-DADO       PIC 99V99.
        05      PRECO-TOTAL         PIC 9(4)V99.
        05      PRECO-FINAL         PIC 9(4)V99.
```

e na Procedure Division do mesmo programa aparecem as seguintes declarações:

```
COMPUTE DESCONTO-DADO = 0.155 * PRECO-TOTAL
COMPUTE PRECO-FINAL = PRECO-TOTAL — DESCONTO
```

Para cada um do seguintes valores de PRECO-TOTAL:

10.00 20.50 37.25 1500.00

calcule os valores correspondentes de DESCONTO-DADO e PRECO–FINAL.

5.3. Explique o efeito das *pictures* seguintes:

```
05      CAMPO-EDITADO-1     PIC Z(5)9.
05      CAMPO-EDITADO-2     PIC £(5)9.99.
05      CAMPO-EDITADO-3     PIC £**,***.99.
05      CAMPO-EDITADO-4     PIC £££,££9.99DB.
05      CAMPO-EDITADO-5     PIC ZZ, ZZ9+.
```

mostrando o *output* que irá ser produzido se cada um dos valores seguintes for movido para cada um dos campos acima: 87654321, 0.756 e —253.

Exercícios de programação 5

5.4. O *input* para o sistema de pagamentos da St. Kilda Puffin Factory consiste em registos de empregado que contém os seguintes campos:

número de empregado	2 letras seguidas de 4 dígitos
pagamento mensal total	5 dígitos (pénis)
desconto anual para impostos	4 dígitos (libras)

Todos os empregados pagam 5% do seu salário para o fundo de pensões da companhia, que são deduzidos antes de se calcular o imposto. São também sujeitos a impostos em 20% do seu rendimento, depois de deduzidos os descontos para impostos e contribuições para pensões.

Desenhe um programa de pagamentos que calcule e imprima a contribuição para a pensão, o imposto a pagar e o salário líquido mensal para cada empregado. O *output* deve ser apresentado com o seguinte formato:

THE ST. KILDA PUFFIN FACTORY

NUM. EMP.	DESC.	PAG. TOT.	PENSAO	IMPOSTO	SAL. LIQ
AA1023	£1,200	£500.00	£25.00	£75.00	£400.00
BC2048	£2,321	£482.96	£24.14	£50.08	£408.74

Investigue o efeito da truncagem e do arredondamento no cálculo da contribuição para pensões, do imposto a pagar e do salário líquido. [O problema 5.4 apareceu anteriormente nos exercícios do capítulo 2. Deve também ser possível implementar os outros dois exercícios desse capítulo no caso de serem requeridas futuras ideias para programas.]

5.5. O National Savings Bank of Ruritania sumaria o estado de cada uma das contas dos seus clientes no fim de cada mês, produzindo um registo de tipo:

Número de conta	8 dígitos
Nome de conta	25 caracteres
Balanço inicial	6 dígitos (pénis, com sinal)
Quantia total de débitos	5 dígitos (pénis)
Quantia total de créditos	5 dígitos (pénis)

A política do banco no que respeita a despesas de serviço é:

Se o balanço final for crédito e maior do que 25 libras, a conta do cliente é creditada com 1% de juro no balanço final. Se o balanço final for débito, o cliente paga uma despesa de serviço de 2% de juro no balanço final. Para contas entre zero e 25 libras não se cobra nenhum serviço nem se paga nenhum juro.

Escreva um programa que leia os registos de conta, calcule o balanço final de cada conta e imprima a quantia cobrada ou o juro adicionado, se houver algum. O *output* deve incluir cabeçalhos adequados e campos monetários correctamente editados.

6 — DESENVOLVIMENTO DE ALGORITMOS

Este capítulo trata de aumentar o número de técnicas disponíveis para solucionar problemas mais complexos do que os explorados até agora.

O repertório de condições disponíveis é aumentado pela introdução de condições de classe para verificar os conteúdos de campos de dados e nomes–condição que permitem que uma só condição seja associada a um grupo de valores. Tanto as condições de classe como os nomes–condição são muito úteis para o grupo de problemas que dizem respeito à validação de dados — a verificação de erros no *input* original, «em bruto», antes do seu processamento.

As condições complexas podem ser implementadas pela combinação de condições simples numa só, e serão consideradas duas possibilidades: declarações if encadeadas e a criação de condições dentro de uma só declaração if.

À medida que os programas se vão tornando mais complexos, torna–se muitas vezes evidente a necessidade de mais declarações, e explica–se o modo como elas são implementadas.

Finalmente, um pequeno estudo de um caso, sobre validação de dados de *input,* é incluído para ilustrar a utilização de algumas das construções escritas.

6.1 — VALIDAÇÃO DE DADOS

O COBOL é uma linguagem criada para o processamento de dados comerciais, e isto implica que pelo menos alguns programas tenham de lidar com *input* em «bruto» nalgumas forma do intercâmbio homem/máquina. Alguns erros irão ocorrer nos dados de *input* se não se tomar em conta o método utilizado durante a transcrição desses dados em bruto para formatos passíveis de serem lidos pela máquina. Num meio de programação comercial, é essencial que o maior número de erros possível seja detectado antes do processamento.

Falhas na detecção dos erros conduzem, por vezes, a situações que os jornais diários adoram, por exemplo, quando uma velha pensionista recebe uma conta de gás de 6 milhões de libras. E também se deseja que possíveis erros sejam detectados antes de provocarem um erro na execução do programa. Não será boa política que um programa que processe dez mil registos de *input* falhe após ter processado nove mil, devido a um simples erro de dactilografia.

Algumas detecções de erros podem ser feitas pelo uso de simples condições que verifiquem distâncias entre valores ou que forneçam verificações «racionais». Por exemplo, espera–se que, em condicões normais, o número de unidades de electricidade consumidas por uma dona de casa seja, digamos, três mil unidades por período de contagem. Assim, o programa de conta poderia incluir uma verificação do tipo:

```
if unidades-usadas not > 3000 then
        conta-normal
else
        verificar-unidades
endif
```

O procedimento «verificar–unidades» poderia criar um comentário para todos os clientes cujo número de unidades gastas fosse «anormal». O *input* pode então ser verificado, no sentido de se detectar qualquer erro na transcrição dos dados originais ou se o contador foi forçado para trás. Na prática, o programa terá também de fornecer uma sub–rotina que permita o processamento de uma conta anormal, mas real.

Para apoiar a detecção dos erros de dados, o COBOL fornece dois tipos adicionais de condições, chamadas *condições de classe* e *nomes–condição.*

6.2 — CONDIÇÕES DE CLASSE E REDEFINIÇÃO

A condição de classe é utilizada para avaliar se os conteúdos correntes de um campo particular são do tipo definido. Uma das situações mais comuns é garantir que os conteúdos de um campo são numéricos, antes de os utilizar numa operação aritmética. A utilização de valores não numéricos em operações aritméticas irá normalmente resultar em erros de execução.

As duas condições de classe fornecidas em COBOL são alfabética e numérica e têm a seguinte forma:

<campo> numerico	<campo> não numerico
<campo> alfabetico	<campo> não alfabetico

A condição «<campo> numerico>» é verdadeira se o <campo> contiver somente caracteres do grupo definido como numérico, ou seja, os dígitos de 0 a 9 e os sinais das operações, sendo falso o contrário. Um campo contendo espaços ou um ponto decimal implícito não é numérico. A condição «<campo> não numerico» é somente uma conversão da condição numérica.

A condição «<campo> alfabetico» é verdadeira se o <campo> contiver apenas caracteres do grupo definido como alfabético, ou seja, letras de A a Z e espaço. Note que um campo contendo pontuação não é alfabético.

A tradução destas condições para COBOL é perfeitamente linear, por exemplo:

```
if preço-venda não numerico then
if código-tarifa alfabetico then
```

ficará:

```
IF PRECO-VENDA NOT NUMERIC
IF CODIGO-TARIFA ALPHABETIC
```

Considerando o problema de dados incorrectos, ele conduz–nos à questão relacionada da execução de *output* de valores inválidos de campos numéricos. Isto é uma ligeira digressão do tema principal deste capítulo, mas pode perfeitamente ser discutido nesta altura.

Suponha que um campo de *output* foi descrito como:

```
05      PRINT-PRECO-ARTIGO  PIC £££9.99.
```

e que se pressupõe que irá ser utilizado para o *output* de um campo definido no *input* como:

```
05      PRECO-VENDA-ARTIGO PIC 999V99.
```

O efeito da declaração:

```
MOVE PRECO-VENDA-ARTIGO TO PRINT-PRECO-ARTIGO
```

é claramente definido se o valor de PRECO–VENDA–ARTIGO for numérico. Porém, se o registo de *input* contiver «ABCDE», que significado terá a edição? Alguns sistemas de COBOL poderão tentar executar a edição, produzindo uma versão mutilada dos dados originais, que poderão até aparecer para indicar que o valor original era numérico. Outros sistemas de COBOL irão dar, neste ponto, um erro de execução e o programa será mal sucedido. Ora, nenhuma destas alternativas é aceitável.

Há duas soluções lineares para este programa: ou ter dois registos de *input,* um para as linhas «normais» e outro para as linhas com erro, ou permitir descrições alternativas para campos particulares de *output.* A primeira solução é preferível, mas é melhor implementada se se utilizarem dois ficheiros de *output,* questão que será considerada no capítulo 9. A segunda, menos satisfatória, pode ser implementada através da utilização da cláusula REDEFINES do COBOL, que será aqui explicada e utilizada no estudo de caso no fim do capítulo.

Cláusula «Redefines»

A cláusula REDEFINES permite ao programador dar dois nomes e descrições à mesma parte de um registo de *input* ou de *output.* As duas alternativas utilizadas devem definir campos exactamente do mesmo comprimento. No exemplo dado previamente seria possível escrever:

```
05  PREÇO-VENDA-ARTIGO                                  PIC 999V99.
05  PREÇO-VENDA-ERRO  REDEFINES  PREÇO-VENDA-ARTIGO     PIC X(5)
.....
05  PRINT-PREÇOARTIGO                                  PIC £££9.99
05  PRINT-PREÇO-ERRO  REDEFINES  PRINT-PREÇO-ARTIGO     PIC X(7).
```

Ao nível do desenho de programa, o uso de redefinição pode ser indicado ligando os nomes alternativos para a mesma área com o símbolo especial «==». Por exemplo:

```
preço-venda-artigo  ==  preço-venda-erro
print-preço-artigo  ==  print-artigo-erro
```

O desenho de programa poderia então incluir o refinamento:

```
produzir-preço-artigo
        if preço-venda-artigo numérico then
                mover preço-venda-artigo to print-preço-artigo
        else
                mover preço-venda-erro  to print-preço-erro
        endif
```

A parte correspondente da Procedure Division do programa seria então:

```
PRODUZIR-PRECO-ARTIGO.
        IF  PRECO-VENDA-ARTIGO  NUMERIC
                MOVE  PREÇO-VENDA-ARTIGO  TO  PRINT-PRECO-ARTIGO
        ELSE
                MOVE  PRECO-VENDA-ERRO  TO  PRINT-PRECO-ERRO.
```

Se o valor de PRECO-VENDA-ARTIGO for numérico, ele será editado para *output,* caso contrário, se for inválido, será escrito em *output* na sua forma «em bruto», sendo inspeccionado e corrigido para posterior reentrada como *input* no programa.

Na prática, a separação de dados válidos e inválidos será acompanhada pela colocação de uma marca de erro e a produção de uma mensagem, que irá tornar a estrutura um pouco mais complexa do que a mostrada acima.

As regras de utilização da redefinição podem ser resumidas como se segue:

a) A descrição alternativa (campo de redefinição) deve ser o primeiro campo, com o mesmo número de nível do campo a ser redefinido (campo redefinido);
b) O campo redefinido e o redefinição devem ter o mesmo comprimento;
c) Os dois campos são ligados pela cláusula REDEFINES como se segue:

<campo-redefinição> REDEFINES <campo-redefinido>

É possível que, quer os campos redefinidos, quer os de redifinição, sejam um item de dados de um grupo. Isto é bastante útil para o alinhamento das colunas de *output*. No exemplo apresentado, dado que os valores incorrectos são transferidos pela utilização de um *move* não numérico, os valores inválidos serão preenchidos a espaços e não serão alinhados com os valores válidos. Por exemplo, usando PRODUZIR–PRECO–ARTIGO, uma mistura de valores válidos e inválidos poderia aparecer assim:

```
...     £23.26      ...
...     ABCDE       ...
...     £0.65       ...
...     £123,20     ...
...     )%%))       ...
```

Algumas pessoas poderiam argumentar que isto é perfeitamente aceitável porque realça os valores inválidos, enquanto outras responderiam que isto «estraga a aparência do *output*».

O problema do alinhamento pode ser ultrapassado fazendo–se do item de redefinição um item grupo, como por exemplo:

```
05      PRINT-PRECO-ARTIGO                          PIC £££9.99.
05      PRINT-PRECO-INVALIDO  REDEFINES  PRINT-PRECO-RTIGO.
          10      FILLER                            PIC XX.
          10      PRINT-PRECO-ERRO                  PIC X(5).
```

Agora, dado que PRECO-VENDA-ARTIGO e PRINT-PRECO–ERRO são do mesmo tamanho, não haverá preenchimento de espaços — eles serão absorvidos pelo FILLER. Utilizando PRODUZIR–PRECO–ARTIGO com a mistura de valores válidos e inválidos, tal como num exemplo prévio, o *output* aparecerá agora como se segue:

```
...     £23,26      ...
...     ABCDE       ...
...     £0.65       ...
...     £123.20     ...
...     )%%))       ...
```

Note que a segunda regra para as cláusulas REDEFINES se mantém ainda neste caso — o comprimento total de PRINT–PRECO–INVALIDO (sete caracteres) é exactamente o mesmo de PRINT–PRECO–ARTIGO.

6.3 — NOMES-CONDIÇÃO

Um nome-condição permite que um conjunto de valores seja associado a um campo particular. Suponha que o número de conta legal numa determinada organização deve situar–se entre 1000 e 4000, podendo talvez ser estes os únicos números que tenham sido atribuídos. Será bom escrever num desenho de programas:

```
if num-conta-correcto then
```

A condição «num-conta-correcto» poderia então ser definida de modo a ser verdadeira se o número de conta fosse legal e falsa no caso contrário. Pode fazer–se isto em COBOL, utilizando um nome–condição associado a um campo. A descrição da estrutura de dados apropriada para este caso seria:

```
05      NUM-CONTA                              PIC 9999.
        88      NUM-CONTA-CORRECTO  VALUE  1000  THRU  4000.
```

Isto define um nome-condição NUM-CONTA-CORRECTO, que é verdadeiro se o valor do campo associado NUM–CONTA se encontra no espaço entre 1000 e 4000 (inclusive) e falso no caso contrário. Na Procedure Division do programa é então possível escrever:

```
IF   NUM-CONTA-CORRECTO
```

Este exemplo pode parecer muito simples, já que seria possível alcançar o mesmo efeito utilizando uma combinação de condições simples, como por exemplo:

```
IF NUM-CONTA NOT < 1000 AND NUM-CONTA NOT > 4000
```

(As combinações de condições simples serão discutidas nas secções 6.4 e 6.5.) No entanto, o conjunto de valores associados ao nome–condição pode ser qualquer combinação de distâncias entre valores e de valores simples. Quanto mais complexo for o conjunto de valores, mais atractivo se torna o uso de nomes–condição. Outra das vantagens dos nomes–condição é que a definição da condição pode ser facilmente alterada, modificando–se o nivel 88 na Data Division em vez da parte de algoritmo do programa.

No exemplo, um valor especial de NUM-CONTA = 9999 pode ser adicionado ao conjunto de valores válidos, caso em que a descrição dos dados fica:

```
05      NUM-CONTA              PIC 9999
        88      NUM-CONTA-CORRECTO  VALUE  1000  THRU  4000,  9999.
```

Pode ser que, mais tarde, se decida adicionar uma nova distância entre os valores, digamos de 5000 a 6500, e de novo a descrição pode ser alterada para:

```
05      NUM-CONTA              PIC 9999.
        88      NUM-CORRECTO  VALUE 1000 THRU 4000, 9999,
                                      5000 THRU 6500.
```

Neste último exemplo, a combinação equivalente de condições simples não é tão fácil de escrever.

As vírgulas entre valores, ou distâncias de valores, não são estritamente necessárias, mas podem ser incluídas para melhorar a legibilidade.

Como exemplo da utilização de um nome-condição, suponha que parte de um identificador é uma letra de código, cujos valores válidos são de A a H, de J a N, P, de R a T e de V a X (todas

as distâncias, incluindo os pontos finais). Então, no desenho de programa, é possível escrever a condição da forma:

if letra-cod-válida then

A condição «letra–cod–válida» será verdadeira se letra–cod contiver um dos valores mencionados acima e falsa no caso contrário. Para implementar este ponto, a descrição de dados apropriada é:

```
05      LETRA–COD                PIC A.
        88      LETRA–COD–VALIDA
                VALUE «A» THRU «H», «J» THRU «N», «P»
                «R» THRU «T», «V» THRU «X».
```

E agora é possível escrever na Procedure Division do programa:

IF LETRA–COD–VALIDA

Um campo de dados pode ter mais do que um nome–condição a si associado. Por exemplo:

```
05      C-APELIDO.
        10      C-PRIM-APELIDO          PIC A.
                88      C-APELIDO-GRUPO-1    VALUE «A» THRU «K».
                88      C-APELIDO-GRUPO-2    VALUE «L» THRU «N».
                88      C-APELIDO-GRUPO-3    VALUE «O» THRU «Z».
        10      C-REST-APELIDO          PIC X(29).
```

É possível, neste exemplo, escrever na Procedure Division do programa qualquer das condições:

```
C-APELIDO-GRUPO-1    NOT  C-APELIDO-GRUPO-1
C-APELIDO-GRUPO-2    NOT  C-APELIDO-GRUPO-2
C-APELIDO-GRUPO-3    NOT  C-APELIDO-GRUPO-3
```

ou utilizar estas condições como partes de condições compostas, a serem discutidas na secção 6.5.

Alguns compiladores ANS COBOL podem não implementar nomes–condição; traduções alternativas são discutidas no apêndice 5.2.

6.4 — DECLARAÇÕES «IF» ENCADEADAS

Uma operação if que aparece entre o if e o endif de outra instrução if é designada como estando «aninhada» dentro dessa outra operação if. Teoricamente, as operações if podem estar «aninhadas» ou encadeadas em qualquer nível (if dentro de if dentro de if...), mas, na prática, demasiados níveis de encadeamento tornam obscuro o significado de um programa. Recomenda–se que não se utilizem mais do que três níveis de encadeamento num programa.

A forma como as declarações if encadeadas são interpretadas em COBOL envolve a percepção de uma simples regra acerca do encadeamento de IFs e de ELSEs. Uma vez estabelecida esta regra, podem formular–se linhas mestras para tradução de estruturas if do nível do desenho para COBOL. Considere o seguinte extracto de um programa de COBOL:

```
EXEMPLO-6-1.
        IF QUANTIDADE-ENCOMENDADA NOT > QUANTIDADE-LIMITE
                IF TIPO-CONTA = «W»
                        PERFORM PREPARAR-FACTURA
                ELSE
                        PERFORM QUANTIDADE-MAIOR.
```

Este segmento de COBOL pode ser interpretado de duas maneiras, tal como se demonstra nos exemplos 6.1 e 6.2.

```
Exemplo 6.1
    if quantidade-encomendada not > quantidade-limite then
            if tipo-conta = «W» then
                    preparar-factura
            else
                    quantidade-maior
            endif
    endif
```

```
Exemplo 6.2
    if quantidade-encomendada not > quantidade-limite then
            if tipo-conta = «W» then
                    preparar-factura
            endif
            else
                    quantidade-maior
    endif
```

O fragmento de COBOL, mostrado no EXEMPLO-6-1, será interpretado tal como se demonstra no exemplo 6.1 porque, em COBOL, a regra é «um ELSE encaixa no IF por encaixar imediatamente precedente». Logo, o ELSE do exemplo será projectado de modo a encaixar em IF TIPO-CONTA... e não em IF QUANTIDADE-ENCOMENDADA... apesar da indentação. A tradução do fragmento de desenho apresentado no exemplo 6.2 para COBOL iria requerer a utilização de refinamento. O extracto correspondente de COBOL seria:

```
EXEMPLO-6-2.
    IF QUANTIDADE-ENCOMENDADA NOT > QUANTIDADE-LIMITE
            PERFORM VERIFICAR-TIPO-CONTA
    ELSE
            PERFORM QUANTIDADE-MAIOR.
    VERIFICAR-TIPO-CONTA.
            IF TIPO-CONTA =«W»
                    PERFORM PREPARAR-FACTURA.
```

Considere o seguinte fragmento de desenho:

```
Exemplo 6.3
    if tipo-cliente = «W» then
            if quantidade-encomendada > 100 then
                    valor-desconto
            else
                    desconto-trans
            endif
    else
            cliente-não-trans
    endif
```

Isto pode ser directamente traduzido para COBOL, porque todos os ifs têm elses e, logo, não podem existir nenhumas ambiguidades ao fazer encaixar os ELSEs em IFs. A tradução para COBOL poderia ser:

```
EXEMPLO-6-3.
    IF TIPO-CLIENTE = «W»
            IF QUANTIDADE-ENCOMENDADA > 100
                    PERFORM VALOR-DESCONTO
            ELSE
                    PERFORM DESCONTO-TRANS
    ELSE
            PERFORM CLIENTE-RETALHO.
```

Algumas estruturas if encadeadas que contenham ifs sem elses podem ser directamente traduzidas para COBOL, por exemplo:

```
Exemplo 6.4
    if salario-tributavel > limite-máximo then
            deducao-taxa-maior
    else
            if salario-tributavel > 0 then
                    deducao-taxa-normal
            endif
    endif
```

Isto pode ser traduzido para COBOL como:

```
EXEMPLO-64.
    IF SALARIO-TRIBUTÁVEL > LIMITE-MAXIMO
            PERFORM DEDUCAO-TAXA-MAIOR
    ELSE
            IF SALARIO-TRIBUTAVEL > 0
                    PERFORM DEDUCAO-TAXA-NORMAL.
```

Existe apenas um ELSE que encaixa perfeitamente no único IF que o precede; logo, a tradução é correcta e sem ambiguidades.

É possível estabelecer um conjunto de linhas mestras para a tradução de estruturas IF para o COBOL, através do estudo destes exemplos. Existem três casos a considerar.

a) Todas as operações if são completas, isto é, todos os ifs tem elses correspondentes: a estrutura pode directamente ser traduzida para COBOL;

b) As únicas operações if incompletas encontram-se no fim da estrutura, de modo que dois ou mais endifs aparecem juntos: a estrutura pode directamente ser traduzida para COBOL;

c) Existem operações if incompletas no meio da estrutura; cada operação if incompleta deve ser substituída por uma referência a um parágrafo onde essa operação if possa ser desenvolvida.

Finalmente, uma linha mestra extra: em caso de dúvida acerca de declarações IF, utilize o refinamnto para que tudo fique absolutamente claro. Por exemplo, o exemplo 6.3 pode igualmente ser traduzido como:

```
EXEMPLO-6-3A.
        IF TIPO-CLIENTE = «W»
                PERFORM VERIFICAR-QUANTIDADE
```

```
            ELSE
                    PERFORM-RETALHO.
            ....
VERIFICAR-QUANTIDADE.
            IF QUANTIDADE-ENCOMENDADA > 100
                    PERFORM  VALOR-DESCONTO
            ELSE
                    PERFORM  DESCONTO-TRANS.
```

A declaração IF encadeada é considerada uma característica de alto nível do *ANS COBOL 74* e, logo, pode não ser implementada em alguns compiladores ANS COBOL de baixo nível. Neste caso, a tradução de um if encadeado, na linguagem de desenho, irá sempre requerer a utilização do refinamento, tal como se demonstrou no exemplo 6.3 e na sua tradução EXEMPLO-6-3A.

Uma implementação de baixo nível de ANS COBOL que não inclua declarações IF encadeadas, provavelmente não incluirá também condições compostas (ver próxima secção), e isto significa que o único mecanismo de selecção será a simples declaração IF com refinamento.

6.5 — CONDIÇÕES COMPOSTAS

Inicialmente, apenas condições simples foram introduzidas no desenho de linguagem, suficientes para muitos problemas. No entanto, muitos problemas têm accções que dependem de uma combinação de duas ou mais condições. Um modo de tratar combinações de condições é usar declarações IF encadeadas, tal como foi discutido na secção 6.4.

Uma abordagem alternativa consiste em construir condições compostas através da combinação de condições simples, usando os operadores lógicos and (e) e or (ou) juntamente com parênteses, se necessário. As combinações de condições também podem ser postas na negativa, através da utilização do operador not (não), mas raramente isto é necessário.

O operador and combina duas condições, de modo que a condição composta seja verdadeira apenas se ambas as condições conjugadas forem verdadeiras. Por exemplo:

Exemplo 6.5

```
if tipo-conta = «C» and num-conta < 9000 then
        conta-liente-especial
endif
```

Nesta estrutura, o refinamento «conta-cliente-especial» é executado se e apenas se ambas as condições simples forem verdadeiras. A tradução para COBOL é perfeitamente linear:

```
EXEMPLO-6-5.
        IF TIPO-CONTA = «C» AND NUM-CONTA < 9000
                PERFORM  CONTA-CLIENTE-ESPECIAL.
```

Devido ao facto de esta tradução ser tão linear, exemplos posteriores serão discutidos utilizando-se apenas a linguagem de desenho.

O exemplo mostrado no exemplo 6.5 é equivalente à estrutura if encadeada apresentada no exemplo 6.6.

Exemplo 6.6

```
if tipo-conta = «C» then
        if num-conta < 9000 then
```

```
                    conta-cliente-especial
            endif
endif
```

Se se adiciona um else ao exemplo 6.5, as operações especificadas no membro else serão realizadas em todas as combinações de condições, o que faz que a condição composta seja falsa. Por exemplo:

```
Exemplo 6.7
if tipo-conta = «C» and num-conta < 9000 then
        conta-cliente-especial
else
        todas-outras-contas
endif
```

A estrutura apresentada no exemplo 6.7 é equivalente à estrutura if encadeada apresentada no exemplo 6.8.

```
Exemplo 6.8
if tipo-conta = «C» then
        if numero-conta < 9000 then
                conta-cliente-especial
        else
                todas-outras-contas
        endif
else
        if num-conta < 9000 then
                todas-outras-contas
        endif
endif
```

A terceira operação if é, obviamente, redundante e pode ser substituída por uma referência linear a «todas–outras–contas», apresentada na estrutura do exemplo 6.9.

```
Exemplo 6.9
if tipo-conta = «C» then
        if num-conta < 9000 then
                conta-cliente-especial
        else
                todas-outras-contas
        endif
else
        todas-outras-contas
endif
```

Olhando para os exemplos 6.7, 6.8 e 6.9, percebemos por que é que as estruturas if encadeadas e as condições compostas são úteis. Uma ligeira alteração ao exemplo 6.9 é apresentada no exemplo 6.10 — como pode ele ser formulado de modo a começar com uma condição composta?

```
Exemplo 6.10
if conta-tipo = «C» then
        if num-conta < 9000 then
                conta-cliente-especial
        else
```

```
                    conta-trans-especial
            endif
else
            todas-outras-contas
endif
```

O facto de se expressar a lógica do exemplo 6.10, iniciado por uma condição composta, dá origem à estrutura bastante desordenada apresentada no exemplo 6.11.

Exemplo 6.11

```
if tipo-conta = «C» e num-conta < 9000 then
            conta-cliente-especial
else
            if tipo-conta = «C» e num-conta not < 9000 then
                        conta-trans-especial
            else
                        todas-outras-contas
            endif
endif
```

Assim, neste caso, a estrutura if encadada é mais apropriada. Isto porque a selecção não é uma escolha entre alternativas, mas uma escolha entre três rumos de acção.

Uma linha mestra simples consiste neste aspecto: se uma combinação de condições conduz a uma escolha entre apenas dois rumos de acção (um dos quais talvez nada execute), uma condição composta é apropriada; caso contrário, uma estrutura if encadeada será melhor.

Tendo tratado o and com alguma profundidade, iremos aplicar comentários semelhantes à utilização do operador lógico or. Se duas operações forem combinadas pelo uso de or, a condição será verdadeira se uma das condições conjugadas for verdadeira ou se ambas forem verdadeiras. A isto chama-se «ou inclusivo». Os exemplos 6.12 e 6.13 ilustram a utilização de or.

Exemplo 6.12

```
if nível-stock < nível-reorganiz or estado-stock = «X» then
            criar-ordem
endif
```

Exemplo 6.13

```
if num-mes < 1 ou num-mes > 12 then
            erro-em-mes
else
            verificar-dias
endif
```

Geralmente, and e or podem ser definidas tal como se demostra na «tabela de verdades» da figura 6.1.

c1	c2	c1 and c2	c1 or c2
V	V	V	V
V	F	F	V
F	V	F	V
F	F	F	F

V — verdadeiro.
F — falso.

Fig. 6.1

Os leitores que utilizem um compilador ANS COBOL de baixo nível que não implemente condições compostas necessitarão de apresentar todas as condições compostas, utilizando declarações IF encadeadas ou campos boolianos (ver apêndice 5.4).

Testabilidade de condições compostas

Pode surgir um problema relacionado com condições compostas se duas ou mais condições simples a serem combinadas não forem testáveis independentemente. Ao avaliar a condição:

```
c1 and c2
```

se c1 é avaliado como falso, obviamente que não há necessidade de avaliar c2, dado que o resultado deve ser falso. Do mesmo modo, ao avaliar:

```
c1 or c2
```

se c1 for verdadeiro, não há necessidade de avaliar c2. No entanto, alguns compiladores não tiram proveito destas optimizações e podem ainda avaliar c2, o que pode conduzir a problemas.

Por exemplo, considere:

```
if preço-vendas numerico and preço-vendas < 500 then
```

Neste caso, quando preço-vendas não for numérico, a condição «preço-venda < 500» é ainda avaliada, e isto pode causar um erro de execução porque esta condição não está definida para valores não numéricos de preço-vendas. Assim, é aconselhavel combinar apenas condições simples testáveis independentemente numa condição composta.

O exemplo apresentado acima estaria escrito mais correctamente como estrutura if encadeada:

```
if preço-vendas numerico then
if preço-vendas < 500 then
```

Isto garante que a condição «preço-vendas < 500» seja avaliada apenas se preço-vendas for numérico.

Todos os exemplos usados têm mostrado a utilização de condições compostas em operações if, mas é também possível utilizá-los em ciclos until. Nos ciclos until podem também surgir problemas com a verificabilidade das condições. Uma armadilha comum consiste em escrever um ciclo until do tipo:

```
until código-trans = 2 or fim-de-transacções do
```

em que «código-trans» faz parte de um registo de *input* de um ficheiro «transacções». O programa será mal sucedido neste ponto, quando «fim-de-transacções» for verdadeiro porque a condição «código-trans = 2» é indefinida (não há nenhum registo de *input* para testar). Trocar a ordem das condições pode não ser suficiente para resolver o problema porque, conforme já foi explicado, alguns compiladores não optimizam a utilização de condições, mas avaliam todas as condições simples de uma condição composta. Uma solução para este tipo de problema será discutida na secção 6.6.

Avaliação de condições compostas

É possível criar condições compostas, combinando mais do que duas condições simples, utilizando and e or, casos em que terão de ser formuladas regras para a avaliação de condições compostas. Isto é semelhante ao problema de definição de uma ordem de precedências para operadores aritméticos.

Exemplo 6.14

```
if nivel-stock < nivel-reorganiz and cod-reorganiz = «R»
        or nivel-stock = 0 then
```

A condição composta apresentada no exemplo 6.14 pode ser interpretada de dois modos, avaliando o and antes do or ou vice-versa. Em COBOL, a regra consiste em avaliar os ands antes dos ors. Assim, a condição composta do exemplo 6.14 interpreta-se avaliando

```
nivel-stock < nivel-reorganiz and cod-reorganiz = «R»
```

para verdadeiro ou falso e, depois, combinando este resultado com o valor de nível-stock = 0, utilizando o operador or. Podem utilizar-se parênteses para alterar a ordem de avaliação de uma condição composta, estabelecendo de novo o paralelo com aquilo que se pratica com as expressões aritméticas. Assim,

```
if c1 and c2 or c3 then
```

é equivalente a:

```
if (c1 and c2) or c3 then
```

mas não equivalente a:

```
if c1 and (c2 or c3) then
```

Uma linha mestra simples consiste em utilizar parênteses para tornar o sentido claro, em caso de dúvida.

Tal como acontece com as estruturas if encadeadas, é possível construir condições compostas de grande complexidade, mas na prática, condições compostas muito complexas tendem a obscurecer o significado dos programas e são indício de um desenho de programa pobre.

A utilização do operador lógico not indica também, frequentemente, um desenho de programa pobre. Este operador pode ser utilizado para fazer a negativa de condições simples ou compostas. Aqui estão três maus exemplos de como usar o operador not:

```
if not (cliente = «A») then
if not (nivel-stock > 0 and estado-stock = «X») then
if not (tipo-cliente = «W» or quantidade-encomendada > 100) then
```

Estes três exemplos podem ser escritos de novo, de modo mais claro, da maneira seguinte:

```
if tipo-cliente not = «A» then
if nivel-stock not > 0 or estado-stock not = «X» then
if tipo-cliente not = «W» and quantidade-encomendada not > 100 then
```

O segundo e terceiro exemplo ilustram a utilização das leis de De Morgan, que podem ser formalmente assim enunciadas:

not (c1 and c2) = not c1 or not c2
not (c1 or c2) = not c1 and not c2

Para sumariar esta secção, sugerem-se as seguintes simples linhas mestras:

a) Podem utilizar-se condições compostas quando uma selecção entre duas vias for requerida, baseada numa condição complexa;
b) As condições simples que formam uma condição composta devem todas ser testáveis individualmente;
c) A utilização de mais do que dois operadores lógicos numa condição composta obscurece o significado de um programa e é índicio de um desenho de programa pobre;
d) Geralmente, a utilização do operador lógico not é desnecessária e, também ele, torna um programa mais díficil de entender.

6.6 — CAMPOS DE DADOS BOOLIANOS

À medida que os algoritmos se tornam mais complexos, atinge-se um ponto em que as combinações de condições e estruturas if encadeadas são inadequadas para expressar a estrutura de um problema. O que muitas vezes acontece é que um número de condições necessita de ser testado e consequentes acções tomadas, e, então, mais tarde, aquando do programa, é necessário «recordar» quais eram os valores das condições.

Muitas linguagens de programação têm um tipo especial de campo, chamado booliano (ou lógico), que pode ser usado para este fim. Um campo de dados do tipo booliano é tal que só pode conter os valores verdadeiro ou falso, e, assim, pode ser formatado para indicar que «isto aconteceu» ou «isto não aconteceu».

Em COBOL não existe nenhuma definição de campo correspondente ao booliano, mas a ideia básica pode ser simulada utilizando-se nomes-condição. Num desenho de programa podem escrever-se declarações da seguinte forma:

mover verdadeiro para <campo-booliano>
mover falso para <campo-booliano>
if <campo-booliano> then
if not <campo-booliano> then
until <campo-booliano> do

A última destas declarações tem um aspecto que se assemelha fortemente a:

until fim-de-<nome-ficheiro> do

A condição fim-de-<nome-ficheiro> é um exemplo de um campo pseudobooliano. Olhando para o esqueleto *standard* do programa, temos um exemplo de como implementar um campo booliano em COBOL.

Geralmente, para implementar um campo booliano em COBOL, deve-se:

a) Escolher um nome para o campo diferente do do campo booliano, mas, de preferência, relacionado com este;
b) Escolher um valor para verdadeiro e, usando o nome do campo booliano como nome-condição, juntá-lo ao campo de dados escolhidos em *a*);
c) Adicionar esta estrutura ao VECTOR-ESTADO.

Por exemplo, suponha que um campo booliano chamado REGISTO-VALIDO deve ser implementado; então, a seguinte estrutura viria a propósito:

```
05      VALIDACAO-REGISTO                PIC X.
        88      REGISTO-VALIDO           VALUE «V»
```

Para activar ou desactivar um campo de dados booliano deve mover-se um valor para o campo com o qual o campo booliano está associado. Assim:

```
mover verdadeiro para registo-valido
```

seria:

```
MOVE «V» TO VALIDACAO-REGISTO
```

porque o valor «V» foi escolhido para representar a condição verdadeiro. Para alterar o valor de registo-válido:

```
mover falso para registo-valido
```

que seria:

```
MOVE «F» TO VALIDACAO-REGISTO
```

Note que, nesta implementação, qualquer valor diferente de «V» pode ser utilizado para mudar o valor de REGISTO-VALIDO para falso.

Para testar o valor dum campo booliano, só se necessita de utilizar o seu nome como nome-condição. Assim, no exemplo acima utilizado, é possível escrever:

```
IF  REGISTO-VALIDO
IF  NOT  REGISTO-VALIDO
```

na Procedure Division do programa. Um exemplo da utilização de um campo booliano é dado na próxima secção.

Um campo de dados booliano pode ser utilizado para resolver o problema do relacionamento de condições num ciclo until discutido na secção anterior sobre «testabilidade de condições compostas». A estrutura:

```
until cod-trans = 2 or fim-de-transaccoes do
        processar-transaccoes
        ler-proximo-transaccoes
enduntil
```

não é segura porque a condição «cod-trans = 2» não é testada quando «fim-de-transacções» é verdadeiro. Um teste de sequência seguro pode ser assegurado pela introdução de um campo booliano «grupo-completo», como se segue:

```
activar-grupo-completo
until grupo-completo do
        processar-transaccoes
        ler-proximo-transaccoes
        activar-grupo-completo
enduntil
.....
activar-grupo-completo
        if fim-de-transaccoes then
                mover verdadeiro para grupo-completo
        else
```

```
                if cod-trans = 2 then
                        mover verdadeiro para grupo-completo
                else
                        mover falso para grupo-completo
                endif
endif
```

Um exemplo do uso do método acima definido é dado no estudo de caso da secção 8.7. Esta técnica pode também ser utilizada, como alternativa para condições compostas, implementando qualquer ciclo onde a condição de paragem dependa de duas ou mais condições.

Os campos boolianos são muito úteis, mas existe somente um tipo restrito numa classe mais geral de variáveis de estado. Como um número de nomes-condição pode ser atribuído a um simples campo em COBOL, é possível definir uma variável de estado com três ou mais estados de identificação. Um exemplo do uso de mais do que uma variável de estado é dado nos exercícios 8.2.

6.7 — ESTUDO DE CASO

Este estudo de caso pretende reunir algumas das ideias discutidas ao longo deste capítulo. Aqui está um problema que integra validação de dados e alguns processamentos simples.

O sistema de cobranças de chamadas telefónicas da Rockall Telecom Corporation recebe registos de *input* do seguinte formato:

Código área	3 caracteres
Número telefone	6 dígitos
Nome e morada do cliente	40 caracteres
Custo do aluger	4 dígitos (pénis)
Impulsos gastos	4 dígitos
Impulsos com utilização de operador	3 dígitos

O código da área consiste em ou duas letras seguidas dum dígito ou uma letra seguida de dois dígitos.

Pretende-se que o programa verifique a validade destes dados e que processe os registos válidos calculando o valor que se deve cobrar a cada cliente. Devem ser executadas as seguintes validações nos registos:

a) O código da área está no formato correcto;
b) O custo do aluguer, os impulsos gastos e os impulsos com utilização do operador são todos números válidos;
c) O custo do aluguer não é menor do que 8.50 libras.

Se algum destes testes falhar, o detalhe do registo deve ser impresso com uma ou mais mensagens de erro adequadas.

Para cada um dos registos, calcular a quantia a cobrar pela adopção dos seguintes elementos:

a) O custo do aluguer;
b) O custo dos impulsos gastos; 2 pénis por impulso e 3 pénis por cada impulso com utilização do operador;
c) VAT 12,5%.

O *output* do programa deverá consistir num mapa tabular dos detalhes de cada registo, ao qual se adicionará, para além da quantia a cobrar, uma mensagem relevante de erro.

A estrutura geral da solução poderia ser:

```
validar–e–cobrar
        imprimir–cabeçalhos
        ler–próximo–fich–tel
        until fim–fich–tel do
                mover falso para presença–erro
                verificar–registo
                if not presença–erro then
                        calcular–conta
                endif
                imprimir–linha–cobranca
                ler–proximo–fich–tel
        enduntil
```

A estratégia geral desta solução é:

- Assumir que não existem erros no registo que está a ser processado, tomando falso o campo booliano «presença–erro»;
- Verificar a validade do campo booliano «presença–erro» pela movimentação até si do valor verdadeiro, se forem detectados no registo um ou mais erros;
- Testar o valor de «presença–erro» e, se não houver erros detectados, calcular preço a cobrar;
- Imprimir os detalhes do presente registo e ler o próximo.

Existem então quatro refinamentos a considerar: imprimir–cabeçalhos, verificar–registo, calcular–conta e imprimir–linha–cobrança. O refinamento de «imprimir–cabeçalhos» será um procedimento linear semelhante ao utilizado para o estudo de caso anterior (secção 5.6). Por exemplo:

```
imprimir–cabecalhos
        escrever–cabecalho–pag
        escrever–linha–em–branco
        escrever–cabecalhos-det
        escrever–linha–em–branco
```

O mais complexo dos refinamentos é «verificar–registo» e pode ser repartido em quatro tarefas mais pequenas:

```
verificar–registo
        verificar–cod–area
        verificar–aluguer
        verificar–aluguer
        verificar–impulsos
        verificar–imp–oper
```

Antes de se proceder ao refinamento de «calcular–conta» é necessário definir nomes para os campos do registo de *input*. Iremos usar a estrutura que se segue para esta solução:

```
registo–tel
        tel–area        código area
        tel–numero      número telefone
```

```
tel–cliente      nome e morada do cliente
tel–aluguer      custo do aluguer
tel–impulsos     impulsos gastos
tel–imp–oper     impulsos com utilização de operador
```

O processamento de registos válidos em calcular–contas é perfeitamente linear:

```
calcular–conta
        calcular conta–base=
                tel–aluguer + 0.02 * tel–impulsos + 0.03 * tel–imp–oper
        calcular conta–cobranca = 1.125 * conta–base
```

em que se convenciona que conta–base é um campo de dados temporários e conta–cobrança é já parte do registo de *output*. Nesta altura é necessário começar–se a construção da lista dos campos temporários:

```
campos–temporários
        conta–base      conta a cobrar antes da adição do VAT
```

Para refinar «imprimir–linha–cobrança» será necessário fazer a primeira tentativa para definição do registo de *output* — que terá de ser aumentado quando as mensagens de erro tiverem de ser adicionadas, fruto do processo de verificação. Supondo que todos os registos de *input* devem ser impressos, e para a verificação, a estrutura básica do registo de *output* poderia ser:

```
linha–cobranca
        cod–area        código área
        cod–numero      número telefone
        cod–cliente     nome e morada do cliente
        cod–aluger      custo do aluguer (edição, pénis–libras)
        cod–impulsos    impulsos gastos (edição)
        cod–imp–oper    impulsos com utiliz. de oper. (edição)
        conta–cobranca  total a cobrar adicionando o VAT
```

O refinamento de imprimir–linha–cobrança será, obviamente:

```
imprimir–linha–cobranca
        mover tel–area para cob–area
        mover tel–numero para cob–numero
        mover tel–cliente para cob–cliente
        mover tel–aluguer para cob–aluger
        mover tel–impulsos para cob–impulsos
        mover tel–imp–oper para cob–imp–oper
        escrever–linha–cobranca
```

É agora necessário voltar ao procedimento de verificação e refinar os seus quatro procedimentos: verificar–cod–area, verificar–aluguer, verificar–impulsos e verificar–imp–oper.

Olhemos primeiro para «verificar–cod–area», que terá de verificar os caracteres individuais de código da área, e, então, na descrição do registo de *input* terá de ser repartido para o permitir. O campo «tel–area» terá de ser um item–grupo constituído por três itens elementares: tel–área–1, tel–área–2 e tel–rea–3. Assim, o registo de *input* «registo–tel» ficará:

```
registo-tel
        tel-area
                tel-area-1
```

```
            tel-area-2
            tel-area-3
      tel-numero
```

Na generalidade, o refinamento de «verificar–cod–area» teria o seguinte formato:

```
verificar-cod-area
      if tel-area-1 invalido then
            erro-area
      else
            if tel-area-2 invalido then
                  erro-area
            else
                  if tela-area-3 invalido then
                        erro-area
                  endif
            endif
      endif
```

onde «erro-area» é um refinamento que irá definir a acção apropriada para um valor inválido do código área.

Os testes estão ligados numa só estrutura, visto que, se uma parte do código for inválida, será escusado testar as restantes partes.

Da descrição do código área pode deduzir–se que tel–area–1 deve ser alfabético, tel–area–2 alfabético ou numérico e tel–area–3 deve ser numérico. Assim, a condição «tel–area–1 invalido» pode ser escrita «tel–area–1 não alfabetico» e «tel–area–3 invalido» como «tel–area–3 não numerico».

Porém, a construção informal

```
if tel-area-2 invalido then
      erro-area
```

representa uma condição composta e pode ser escrita de várias formas equivalentes, por exemplo:

a)
```
if tel-area-2 not alfabetico
      and tel-area-2 not numerico then
erro-area
```

b)
```
if not (tel-area-2 alfabetico or tel-area-2 numerico) then
erro-area
```

c)
```
if tel-area-2 not alfabetico then
if tel-area-2 not numerico then
      erro-area
endif
```

Baseados nas linhas mestras fornecidas no fim da secção 6.5, seria escolhida a primeira possibilidade. Assim, um possível refinamento de verificar–cod–area seria:

```
verificar-cod-area
      if tel-area-1 not alfabetico then
            erro-area
      else
            if tel-area-2 not alfabetico
```

```
                    and tel-area-2 not numerico then
                            erro-area
            else
                    if tel-area-3 not numerico then
                            erro-area
                    endif
            endif
    endif
```

A primeira aproximação ao refinamento «erro-area» é:

```
erro-area
        mover verdadeiro para presença-erro
```

Colocar em «presença-erro» o valor verdadeiro irá assegurar que, após todas as verificações terem sido efectuadas, nenhuma tentativa de cálculo neste registo com erros será efectuado. Se um registo contém vários erros, então «presença–erro» deve ser tornado verdadeiro em mais do que um dos refinamentos. Isto não interessa na medida em que só é necessário distinguir entre ausência total de erros e existência de, pelo menos um erro — o número de erros é irrelevante.

O refinamento de «verificar–aluguer» ilustra o ponto acerca das condições relacionadas (ver 6.5). O campo «tel–aluguer» deve ser verificado no sentido de se o seu conteúdo é numérico e, também, se excede o mínimo permitido de 8,50 libras. A segunda condição só é testada se a primeira for verdadeira. Assim, a forma básica do refinamento poderia ser:

```
verificar-aluguer
        if tel-aluguer not numerico then
                mover verdadeiro para presença-erro
        else
                if tel-aluguer < 8.5 then
                        mover verdadeiro para presença-erro
                endif
        endif
```

A primeira aproximação aos refinamentos «verificar–impulsos» e «verificar–impoper» é perfeitamente linear:

```
verificar-impulsos
        if tel-impulsos not numerico then
                mover verdadeiro para presença-erro
        endif

verificar-imp–oper
        if tel-imp-oper not numerico then
                mover verdadeiro para presença-erro
        endif
```

Tendo decidido da estrutura básica do desenho de programa, existem dois pontos posteriores a ser considerados: a construção de mensagens de erro e a impressão de campos de dados não numéricos que tenham ocorrido em verificações de campos supostos numéricos.

A forma mais simples de se lidar com mensagens de erro é adicionar quatro campos separados a «linha–cobrança», para reterem as mensagens de erro resultantes dos testes dos quatro campos de dados, a saber: tel–área, tel–aluguer, tel–impulsos e tel–impoper. Poderão então ser chamados: cob–mens–área, cobmens–aluguer, cob–mens–impulsos e cobmens–imp–oper, respectivamente. Agora, os refinamentos convenientes devem ser modificados para carregarem esses campos quando necessário.

O primeiro refinamento a ser modificado é «verificar–aluguer», onde se podem adicionar duas mensagens de erro:

```
verificar-aluguer
        if tel-aluguer not numerico then
                mover verdadeiro para presença-erro
                mover «aluguer invalido» para cob-mens-aluguer
        else
                if tel-aluguer < 8.5 then
                        mover verdadeiro para presença-erro
                        mover «aluguer abaixo do minimo» para cob-mens-aluguer
                endif
        endif
```

As alterações aos outros refinamentos afectados são lineares:

```
verificar-impulsos
        if tel-impulsos not numerico then
                mover verdadeiro para presença-erro
                mover «impulsos invalido» para cob-mens-impulsos
        endif

verificar-impoper
        if tel-impoper not numerico then
                mover verdadeiro para presença-erro
                mover «impulsos op. invalido» para cob-mens-imp-oper
        endif
erro-area
        mover verdadeiro para presença-erro
        mover «area invalido» para cob-mens-area
```

Finalmente, para completar o desenho de programa, quais os campos supostos numéricos, cujo *output* seria normalmente editado, que precisariam de ter campos de *output* alternativos?

Neste exemplo existem três campos nessas condições: tel–aluguer, tel–impulsos e tel–imposer. Podem ser definidos três campos de *input* alternativos: tel–aluguer–erro, tel–impulsos–erro e tel–imp–oper–erro; juntamente com três campos de *output* alternativos correspondentes: cob–aluguer–erro, cob–impulsos–erro e cob–imp–oper–erro. O desenho de programa necessitará então de ser modificado, de modo que estes campos alternativos sejam correctamente referenciados.

```
verificar-aluguer
        if tel-aluguer not numerico then
                mover verdadeiro para presença-erro
                mover «aluguer invalido» para cob-mens-aluguer
                mover tel-aluguer-erro para cob-aluguer-erro
        else
                mover tel-aluguer para cob-aluguer
                if tel-aluguer < 8.5 then
                        mover verdadeiro para presença-erro
                        mover «aluguer abaixo do minimo» para cob-mens-aluguer
                endif
        endif
```

```
verificar-impulsos
        if telimpulsos not numerico then
                mover verdadeiro para presença-erro
                mover «impulsos invalido» para cob-mens-impulsos
                mover tel-impulsos-erro para cob-impulsos-erro
        else
                mover tel-impulsos-erro para cob-impulsos
        endif

verificar-impoper
        if tel-imp-oper not numerico then
                mover verdadeiro para presença-erro
                mover «impulsos-op. invalido» para cob-mens-imp-oper
                mover tel-imp-oper-erro para cob-imp-oper-erro
        else
                mover tel-imp-oper para cob-imp-oper
        endif
```

Tendo executado estas modificações, o refinamento «imprimir–linha–cobrança» pode ser reduzido a:

```
imprimir-linha-cobrança
        mover tel-area para cob-area
        mover tel-numero para cob-numero
        mover tel-cliente para cob-cliente
        escrever-linha-cobrança
```

O processo de desenvolvimento do desenho de programa foi apresentado num certo número de fases de modo a realçar e discutir certos aspectos. Na prática, o desenho de programas completo poderia ser construído sem que fosse necessário explorar todos estes passos intermédios explicitamente.

Antes de apresentar o programa de COBOL para este problema, decerto que seria útil desenhar todas as versões finais dos refinamentos num desenho de programas único, que pode ser então estudado e verificado como uma só unidade.

A versão final do registo de *input* é:

```
registo-tel
        tel-area
                tel-area-1
                tel-area-2
                tel-area-3
        tel-numero
        tel-cliente
        tel-aluguer == tel-aluguer-erro
        tel-impulsos == tel-impulsos-erro
        tel-imp-oper == tel-imp-oper-erro
```

A descrição completa do registo de *output* é:

```
linha-cobrança
        cob-area
        cob-numero
        cob-cliente
        cob-aluguer == cob-aluguer-erro
        cob-impulsos == cob-impulsos-erro
```

```
cob-imp-oper == cob-imp-oper-erro
conta-cobrança
cob-mens-area
comens-aluguer
cob-mens-inpulsos
cob-mens-imp-oper
```

Apenas um campo temporário foi referenciado durante o desenho de programa:

```
campos-temporarios
        conta-base
```

A versão final do desenho de programa já completo é dada a seguir:

```
validar-e-cobrar
        imprimir-cabeçalhos
        ler-proximo-fich-tel
        until  fim-fich-tel do
                mover falso para presença-erro
                verificar-registo
                if not presença-erro then
                        calcular-conta
                endif
                imprimir-linha-cobrança
                ler-proximo-fich-tel
        enduntil

imprimir-cabeçalhos
        escrever-cabeçalho-pag
        escrever-linha-em-branco
        escrever-cabeçalhos-det
        escrever-linha-em-branco

verificar-registo
        verificar-cod-area
        verificar-aluguer
        verificar-impulsos
        verificar-imp-oper

calcular-conta
        calcular contabase =
                tel-aluguer + 0.02 * tel-impulsos + 0.03 * tel-imp-oper
        calcular conta-cobrança = 1.125 * conta-base

imprimir-linha-cobrança
        mover tel-area para cob-area
        mover tel-numero para cob-numero
        mover tel-cliente para cob-cliente
        escrever-linha-cobrança

verificar-cod-area
        if tel-area-1 not alfabetico then
                erro-area
        else
                if tel-area-2 not alfabetico
```

```
                                    and tel-area-2 not numerico then
                            erro-area
                    else
                            if tel-area-3 not numerico then
                                    erro-area
                            endif
                    endif
            endif

    erro-area
            mover verdadeiro para presença-erro
            mover «area invalido» para cob-mens-area

    verificar-aluguer
            if tel-aluguer not numerico then
                    mover verdadeiro para presença-erro
                    mover «aluguer invalido» para cob-mens-aluguer
                    mover tel-aluguer-erro para cob-aluguer-erro
            else
                    mover tel-aluguer para cob-aluguer
                    if tel-aluguer < 8.5 then
                            mover verdadeiro para presença-erro
                            mover «aluguer abaixo do minimo» para cob-mens-aluguer
                    endif
            endif

    verificar-impulsos
            if tel-impulsos not numerico then
                    mover verdadeiro para presença-erro
                    mover «impulsos invalido» para cob-mens-impulsos
                    mover tel-impulsos-erro para cob-impulsos-erro
            else
                    mover tel-impulsos para cob-impulsos
            endif

    verificar-imp-oper
            if tel-imp-oper not numerico then
                    mover verdadeiro para presença-erro
                    mover «impulsos op. invalido» para cob-mens-imp-oper
                    mover tel-imp-oper-erro para cob-imp-oper-erro
            else
                    mover tel-imp-oper para cob-imp-oper
            endif
```

A fase seguinte do processo de desenvolvimento do programa é a descrição dos registos de *input* e *output*. Ao descrever o registo de *output* e os cabeçalhos para este exemplo, depressa se torna óbvio que, tal como especificado, existe demasiada informação para caber numa linha de impressora de cento e vinte caracteres. Como solução, sugere-se que se reduzam os comprimentos das mensagens e que se utilize uma linha dupla de cabeçalhos de coluna para comprimir mais informação numa página. Qualquer solução que requeira que as linhas de detalhe sejam divididas em duas linhas de *output* dá origem a um *output* ilegível.

As mensagens são encurtadas, substituindo-se «xxxxxx invalido» por «invalido» e «aluguer abaixo do minimo» por «abaixo do minimo». O tipo de erro é então incluído nos cabeçalhos dos campos mensagem. «Escrever-cabeçalhos-det» do desenho tem de ser depois refinado para:

escrever-cabeçalhos-det
escrever-cabeçalhos-det-1
escrever-cabeçalhos-det-2

para executar o *output* das duas linhas dos cabeçalhos de coluna.

A versão final do programa é apresentada em baixo e o exemplo do *output* apresentado na figura 6.2 a seguir ao programa.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. TEL-CONTA.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. <nome-computador>.
OBJECTCO-MPUTER. <nome-computador>.
*       <nome-computador> deve ser substituído pelo nome
*       do computador usado para correr programas.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
SELECT FICH-TEL ASSIGN TO <sistema-input>.
SELECT FICHOUT ASSIGN TO <sistema-output>.
*       <sistema-input> e <sistema-output> devem estar conformes
*       com as regras do sistema utilizado.
DATA DIVISION.
FILE SECTION.
FD      FICH-TEL
        LABEL RECORDS OMITTED.
01      REGISTO-TEL.
        05      TEL-AREA.
        10      TEL-AREA-1                                      PIC X.
        10      TEL-AREA-2                                      PIC X.
        10      TEL-AREA-3                                      PIC X.
        05      TEL-NUMERO                                      PIC 9(6).
        05      TEL-CLIENTE                                     PIC X(40).
        05      TEL-ALUGUER                                     PIC 99V99.
        05      TEL-ALUGUER-ERRO REDEFINES TEL-ALUGUER          PIC XXXX.
        05      TEL-IMPULSOS                                    PIC 9999.
        05      TEL-IMPULSOS-ERRO REDEFINES TEL-IMPULSOS        PIC XXXX.
        05      TEL-IMP-OPER                                    PIC 999.
        05      TEL-IMP-OPER-ERRO REDEFINES TEL-IMP-OPER        PIC XXX.
FD      FICHOUT
        LABEL RECORDS OMITTED.
01      LINHA-OUT                                               PIC X(120).
WORKING-STORAGE SECTION.
01      VECTOR-ESTADO
        05      FIM-FICH-TEL                    PIC X.
                88      FIM-FICH-TEL                    VALUE«E».
        05      AVISO-ERRO                      PIC X.
                88      PRESENCA-ERRO                   VALUE «V».
        01      CAMPOS-TEMPORARIOS.
        05      CONTA-BASE                      PIC 999V99.
01      CABECALHOS-PAGINA.
        05      FILLER          PIC X(46)       VALUE SPACES.
        05      FILLER          PIC X(28)       VALUE
                «ROCKALL TELECOM. CORPORATION».
O1      LINHA-EM-BRANCO         PIC X           VALUE SPACES.
```

```
01  CABECALHOS-DET-1.
    05      FILLER                  PIC X(63)       VALUE  SPACES.
    05      FILLER                  PIC X(5)        VALUE«UNID.».
    05      FILLER                  PIC X           VALUE  SPACE.
    05      FILLER                  PIC X(5)        VALUE  «UNID.».
    05      FILLER                  PIC X(11)       VALUE  SPACES.
    05      FILLER                  PIC X(35)       VALUE
     «<———————————— ERROS DETECTADOS ————————————>»

01  CABECALHOS-DET-2.
    05      FILLER                  PIC XXXX        VALUE  «AREA».
    05      FILLER                  PIC XX          VALUE  SPACES.
    05      FILLER                  PIC X(6)        VALUE  «NUMERO».
    05      FILLER                  PIC X(9)        VALUE  SPACES.
    05      FILLER                  PIC X(25)       VALUE
    «NOME E MORADA DO CLIENTE».
    05      FILLER                  PIC X(1O)       VALUE  SPACES.
    05      FILLER                  PIC X(6)        VALUE  «ALUG.»
    05      FILLER                  PIC XX          VALUE  SPACES.
    05      FILLER                  PIC XXXX        VALUE  «IMP.».
    05      FILLER                  PIC XX          VALUE  SPACES.
    05      FILLER                  PIC XXX         VALUE  «OP.».
    05      FILLER                  PIC XXX         VALUE  SPACES.
    05      FILLER                  PIC X(6)        VALUE  «CUSTOS».
    05      FILLER                  PIC XXXX        VALUE  SPACES.
    05      FILLER                  PIC XXXX        VALUE  «AREA».
    05      FILLER                  PIC X(5)        VALUE  SPACES.
    05      FILLER                  PIC X(6)        VALUE  «ALUGU.».
    05      FILLER                  PIC XX          VALUE  SPACES.
    05      FILLER                  PIC X(7)        VALUE  «IMPULS.».
    05      FILLER                  PIC XX          VALUE  SPACES.
    05      FILLER                  PIC X(8)        VALUE  «OPERADOR».
01  LINHA-COBRANCA.
    05      FILLER                  PIC X.
    05      COB-AREA                PIC XXX.
    05      FILLER                  PIC XX.
    05      COB-NUMERO              PIC 9(6).
    05      FILLER                  PIC XX.
    05      COB-CLIENTE             PIC X(40).
    05      FILLER                  PIC XX.
    05      COB-ALUGUER             PIC ££9.99.
    05      COB-ALUGUER-ERRADO  REDEFINES  COB-ALUGUER.
            10  FILLER             PIC XX.
            10 COB-ALUGUER-ERROPIC  XXXX.
    05      FILLER                  PIC XX.
    05      COB-IMPULSOS            PIC ZZZ9.
    05      COB-IMPULSOS-ERRO  REDEFINES  COB-IMPULSOS
    PIC XXXX.
    05      FILLER                  PIC XX.
    05      COB-IMP-OPER            PIC ZZ9.
    05      COB-IMP-OPER-ERRO  REDEFINES  COB-IMP-OPER
                                    PIC XXX.
    05      FILLER                  PIC XX.
    05      CONTA-COBRANCA          PIC £££9.99.
    05      FILLER                  PIC XXX.
```

```
        05      COB-MENS-AREA              PIC X(7).
        05      FILLER                PIC XX.
        05      COB-MENS-ALUGUER      PIC X(7).
        05      FILLER                PIC XX.
        05      COB-MENS-IMPULSOS     PIC X(7).
        05      FILLER                PIC XX.
        05      COB-MENS-IMP-OPER     PIC X(7).
PROCEDURE DIVISION.
POGRAMA-PRINCIPAL.
        PERFORM ESTADO-INIC
        PERFORM VALIDAR-COBRAR
        PERFORM FECHO.
ESTADO-INIC.
        MOVE SPACE TO FIM-FICH-TEL
        OPEN INPUT FICH-TEL OUTPUT FICHOUT
        MOVE SPACES TO LINHA-COBRANCA.
FECHO.
        CLOSE FICH-TEL FICHOUT
        STOP RUN.
LER-PROXIMO-TEL.
        READ FICH-TEL
                AT END MOVE «E» TO FIM-FICH-TEL.
ESCREVER-LINHA-COBRANCA.
        WRITE LINHA-OUT FROM LINHA-COBRANCA
        MOVE SPACES TO LINHA-COBRANCA.
ESCREVER-CABECALHOS-PAG.
        WRITE LINHA-OUT FROM CABECALHO-PAG.
ESCREVER-LINHA-EM-BRANCO.
        WRITE LINHA-OUT FROM LINHA-EM-BRANCO.
ESCREVER-CABECALHOS-DET-1.
        WRITE LINHA–OUT FROM CABECALHO–DET-1.
ESCREVER–CABECALHOS–DET-2.
        WRITE LINHA–OUT FROM CABECALHO–DET-2.
VALIDAR–COBRAR.
        PERFORM PRINT–CABECALHOS.
        PERFORM LER–PROXIMO–FICH–TEL
        PERFORM VERIFICAR–E–IMPRIMIR UNTIL FIM–FICH–TEL.
PRINT–CABECALHOS.
        PERFORM ESCREVER–CABECALHOS–PAG
        PERFORM ESCREVER–LINHA–EM–BRANCO
        PERFORM ESCREVER–LINHA–EM–BRANCO
        PERFORM ESCREVER–LINHA–EM–BRANCO
ESCREVER–CABECALHOS-DET.
        PERFORM ESCREVER–CABECALHOS–DET-1
        PERFORM ESCREVER–CABECALHOS–DET-2.
VERIFICAR–E–IMPRIMIR.
        MOVE «F» TO AVISO–ERRO
        PERFORM VERIFICAR–REGISTO
        IF NOT PRESENCA–ERRO
                PERFORM CALCULAR–CONTA.
        PERFORM IMPRMIR–LINHA–COBRANCA
        PERFORM LER–PROXIMO–FICH–TEL.
VERIFICAR–REGISTO.
        PERFORM VERIFICAR–COD–AREA
```

```
        PERFORM  VERICAR-ALUGUER
        PERFORM  VERIFICAR-IMPULSOS
        PERFORM  VERIFICAR-IMP-OPER.
CALCULAR-CONTA.
        COMPUTE  CONTA-BASE = TEL-ALUGUER
                + 0.02 * TEL-ALUGUER + 0.03 * TEL-IMP-OPER
        COMPUTE  CONTA-COBRANCA = 1.125 * CONTA-BASE.
IMPRIMIR-LINHA-COBRANCA.
        MOVE TEL-AREA TO COB-AREA
        MOVE TEL-NUMERO TO COB-NUMERO
        MOVE TEL-CLIENTE TO COB-CLIENTE
        PERFORM  ESCREVER-LINHA-COBRANCA.
VERIFICAR-COD-AREA.
        IF TEL-AREA-1 NOT ALPHABETIC
                PERFORM  ERRO-AERA
        ELSE
                IF TEL-AREA-2 NOT ALPHABETIC
                        AND TEL-AREA-2 NOT NUMERIC
                PERFORM  ERRO-AREA
        ELSE
                IF TEL-AREA-3 NOT NUMERIC
                        PERFORM  ERRO-AREA.
ERRO-AREA.
        MOVE «V» TO PRESENCA-ERRO
        MOVE «INVALIDO» PARA COB-MENS-AREA.
VERIFICAR-ALUGUER.
        IF TEL-ALUGUER NOT NUMERIC
                MOVE «V» TO PRESENCA-ERRO
                MOVE «INVALIDO» PARA COB-MENS-ALUGUER
                MOVE TEL-ALUGUER-ERRO TO COB-ALUGUER-ERRO
        ELSE
                MOVE TEL-ALUGUER TO COB-ALUGUER
                IF TEL-ALUGUER < 8.5
                        MOVE «V» TO PRESENCA-ERRO
                        MOVE «ABAIXO DO MINIMO» PARA COB-MENS-ALUGUER.
VERIFICAR-IMPULSOS.
        IF TEL-IMPULSOS NOT NUMERIC
                MOVE «V» TO PRESENCA-ERRO
                MOVE «INVALIDO» PARA COB-MENS-IMPULSOS
                MOVE TEL-IMPULSOS-ERRO TO COB-IMPULSOS-ERRO
        ELSE
                MOVE TEL-IMPULSOS TO COB-IMPULSOS.
VERIFICAR-IMP-OPER.
        IF TEL-IMP-OPER NOT NUMERIC
                MOVE «V» TO PRESENCA-ERRO
                MOVE «INVALIDO» PARA COB-MENS-INP-OPER
                MOVE TEL-IMP-OPER-ERRO TO COB-IMP-OPER-ERRO
        ELSE
                MOVE TEL-IMP-OPER TO COB-IMP OPER.
```

Um exemplo do *output* deste programa é apresentado na figura 6.2.

ROCKALL TELECOM CORPORATION

AREA	NUMERO	NOME E MORADA DO CLIENTE	ALUG.	UNID. IMP.	UNID. OP.	CUSTOS	<——ERROS DETECTADOS——> AREA	ALUG.	IMPULS.	OPERADOR
9BC	123458	PRIMEIRO CODIGO AREA INVALIDO	£12.50	75	50		INVALIDO			
A*3	123457	SEGUNDO CODIGO AREA INVALIDO	£12.50	50	5		INVALIDO			
ABC	123456	TERCEIRO CODIGO AREA INVALIDO	£12.50	25	0		INVALIDO			
AR1	234567	ALUGUER NAO NUMERICO	))!)	100	75			INVALIDO		
AR2	234568	ALUGUER ABAIXO DO MINIMO	£ 8.49	125	100			AL.BAIXO		
AD1	345678	IMPULSOS NAO NUMERICO	£9.50	ABCD	150				INVALIDO	
AO1	345679	UNIDADES OPERADOR NÃO NUMERICO	£15.65	1000	)*)					INVALIDO
ZZ9	999999	VALORES MAXIMOS EM TODOS OS CAMPOS———	£99.99	9999	999	£371.18				
VA1	567893	ALUGUER MINIMO	£8.50	1	1	£9.61				
VA2	567890	IMPULSOS E IMPULSOS COM OPERADOR A ZEROS	£9.50	0	0	£10.68				
VA3	567891	IMPULSOS COM OPERADOR A ZEROS	£18.15	150	0	£23.79				
VA4	567892	UTILIZADOS EM CALCULOS TODOS OS CAMPOS	£19.14	1252	115	£52.68				
3*A	456789	TODOS OS CAMPOS INVALIDOS	ABCD	EFGH	IJK		INVALIDO	IVALIDO	IVALIDO	IVALIDO

Fig. 6.2

Exercícios 6

6.1. A construção seguinte aparece num desenho de programa:

```
if modelo-em-stock then
        verificar-part-disponivel
endif
```

O valor do nome–condição «modelo–em–stock» seria verdadeiro para os seguintes valores do seu campo associado «id–part–modelo»:

4 a 9 (inclusive), 12 a 16 (inclusive), 20 e 24

a) Escreva descrições de dados para «id–part–modelo» e para «modelo–em–stock» e a tradução do desenho acima para COBOL;
b) Demonstre como é que o fragmento de desenho acima poderia ser expresso utilizando ifs encadeados e ifs compostos e refinamento se se não pudessem utilizar nomes–condição. Traduza este novo fragmento de desenho para COBOL.

6.2. Um registo de *input* para um programa de COBOL contém um campo de um só carácter, chamado TIPO–CONTA, que só tem dois valores, «W» (venda a grosso) ou «R» (retalho). A determinada altura do processamento de um registo, requerem–se rotinas diferentes para lidar com registos que se relacionem com a venda a grosso, venda a retalho e tipos de conta errados. Supõe–se que o seguinte parágrafo extraído de um programa de COBOL executa a rotina apropriada para o valor de TIPO–CONTA.

```
PROCESSO–TIPO–CONTA.
        IF TIPO–CONTA = «W»
                PERFORM  ROTINA–GROSSO
        IF TIPO–CONTA = «R»
                PERFORM  ROTINA–RETALHO
        IF (TIPO–CONTA NOT = «W»)
                        OR (TIPO–CONTA NOT = «R»)
                PERFORM  TIPO–CONTA–ERRO.
```

Explique o que executa realmente este fragmento de programa de COBOL quando executado para vários valores de TIPO–CONTA. Apresente um fragmento de desenho de programa equivalente ao extracto de COBOL acima. Construa uma versão correcta do desenho de programa e codifique-o em COBOL, utilizando declarações IF encadeadas.

Exercícios de programação 6

6.3. A Picardy Rose Company expede ramos a clientes para todas as ilhas do Reino Unido e necessita de um sistema computadorizado simples para a facturação. O *input* para o programa de facturação irá consistir em registos com o seguinte formato:

Número de cliente	6 dígitos
Designação de cliente	1 letra (ver abaixo)
Código de distância	1 letra
Quantidade encomendada	5 dígitos
Preço por ramo	4 dígitos (pénis)

Requer–se que o programa valide os registos através das seguintes operações:

a) A designação do cliente é «W» (grossista) ou «R» (retalhista);
b) O código de distância é «A», «B» ou «C»;
c) A quantidade encomendada é um número válido e não menor do que o mínimo permitido de cinco ramos;
d) O preço é um número válido.

Se algum destes testes falhar, o detalhe do registo deve ser imprimido com uma ou mais mensagens de erro apropriadas, no caso de ser detectado mais do que um só erro.
Para cada registo válido calcule a quantia a cobrar ao cliente, aplicando as seguintes regras:

a) Fazer um desconto de 7% sobre o preço de aquisição para uma entrega com o código de distância «A» e de 3,5% para uma com o código «B»;
b) Fazer um desconto adicional de 4% sobre o preço de aquisição para qualquer encomenda que exceda cem ramos, independentemente da distância;
c) Um cliente retalhista tem de ter VAT de 15% adicionado à quantia a cobrar.

O *output* do programa deverá consistir numa listagem de detalhes de cada registo juntamente com quaisquer mensagens de erro apropriadas ou a quantia a cobrar. Todos os campos de *output* devem ser editados e o mapa deve incluir um cabeçalho com o nome da companhia e cabeçalhos adequados para as colunas de *output*.

6.4. A Instant Death (Agriculture) Company aluga a agricultores equipamento para pulverização de sementeiras, com base diária, e vende também os químicos necessários à pulverização.
O *input* para o programa de contas de clientes da companhia consiste em registos com o seguinte formato:

Código equipamento	1 letra e 3 dígitos
Preço aluguer/dia	6 dígitos (pénis)
Preço pulverizante/litro	4 dígitos (pénis)
Número de cliente	2 letras e 3 dígitos
Número de dias de aluguer	3 dígitos
Litros de pulverizante utilizados	5 dígitos

O código de equipamento é composto por duas partes: uma letra de código indicando o tipo de equipamento e um número de série de três dígitos. O equipamento está agrupado em três categorias, de acordo com a letra de código,

Categoria um: «A», «B» e «C»;
Categoria dois: «F» e «G»;
Categoria três: «W», «X», «Y» e «Z».

Estes nove códigos são os únicos permitidos.
Devem ser executadas as seguintes validações para cada registo:

a) O número de cliente é válido;
b) A letra código do equipamento faz parte das nove permitidas;
c) O número de dias de aluguer e os litros de pulverizante utilizados são números válidos.

Se alguma destas verificações falhar, o detalhe do registo deve ser imprimido juntamente com a respectiva mensagem de erro.
Para cada registo válido, calcule o valor a pagar pelo cliente, adicionando os seguintes elementos:

a) O custo de aluguer base menos um desconto de 5%, se o aluguer for para mais de dez dias;

b) Uma taxa de seguro dependendo do tipo de equipamento: categoria um 5 libras por dia, categoria dois 7 libras e categoria três 11 libras, também por dia;

c) Um preço pela quantidade de pulverizante utilizado.

O *output* do programa deve consistir numa listagem dos detalhes das encomendas, juntamente com a quantia a cobrar ou com mensagens de erro relevantes. O mapa deve incluir cabeçalhos adequados e edição de campos.

7. ERROS E TESTES

O estudante que tenha trabalhado firmemente ao longo deste livro terá adquirido um número considerável de ideias em programação e a linguagem COBOL ter–lhe-á sido «atirada» na altura em que atinge este capítulo. Espera–se que ele tenha também tentado desenhar, codificar e executar alguns programas de COBOL ou que esteja no processo de desenvolvimento de um ou mais programas.

O objectivo deste capítulo é dar–lhe um «espaço para respirar» para que consolide a sua experiência em programação. Introduzem–se aqui algumas ideias sobre o desenvolvimento de programas, mas não se inclui muito mais matéria adicional de COBOL. No entanto, isto não significa que este capítulo seja um extra opcional!

Tendo desenhado um programa, verificado o desenho e traduzido este para COBOL, a próxima fase no processo de desenvolvimento é tentar compilar um programa (ver secção 1.3). Para o programador principiante, a primeira compilação produz, geralmente, um grande número de erros de sintaxe. Estes têm de ser identificados, as modificações executadas e o programa tem de ser de novo compilado. Gradualmente, o número de erros de sintaxe é reduzido, até que se consegue uma compilação «limpa». É muito difícil formular regras para lidar com erros de sintaxe — a experiência é o melhor professor. No entanto, na primeira secção deste capítulo apresentam–se algumas linhas mestras gerais que permitem discutir erros de sintaxe.

Quando um programa foi compilado com sucesso, precisa de ser testado para verificar pequenas falhas (debugging). Um bug (pequena falha) é um erro que origina falhas no programa ou produz resultados errados. Este tipo de verificação é um processo de localizar falhas num programa, tentar corrigi–los e testá–los para verificar se a correcção resultou. Este processo pode percorrer vários ciclos. Sendo assim, na próxima secção serão apresentadas algumas pistas para a verificação de programas.

Relacionado com este tipo de verificação existe o problema dos testes. A execução de um programa precisa de ser testado através de um conjunto variado de dados de teste e os resultados produzidos pelo programa comparados com os resultados esperados. A terceira secção deste capítulo analisa como desenhar dados de teste de modo a aumentar a probabilidade da detecção de falhas no programa.

7.1 — ERROS DE SINTAXE

O objectivo da compilação é traduzir um programa de COBOL, de uma linguagem de alto nível, para uma linguagem de baixo nível que possa ser executada pelo computador. Este processo de tradução é mecânico e depende do facto de o programa de COBOL estar sintacticamente correcto, isto é, escrito de acordo com as regras sintácticas do COBOL, resumidas no apêndice 4. Se o programador desobedecer a estas regras de sintaxe, o compilador produzirá uma ou várias mensagens de erro, tentando identificar esse mesmo erro ou erros. O compilador tentará então «ressarcir–se» do erro tentando continuar a compilação do programa.

É impossível produzir um conjunto de regras para identificação e correcção de erros de sintaxe, mas podem ser definidas algumas linhas mestras gerais para o auxílio do principiante. Ge-

ralmente, o maior problema que se depara a um programador é que múltiplas mensagens de erro podem resultar de um só erro no programa. Por exemplo, aqui está um extracto de uma listagem de um programa de COBOL:

```
154     PERFORM  VERIFICAR–AREA–RECORD
...
167     VERIFICAR  AREA  RECORD.
```

Os números à esquerda são números de linha adicionados pelo compilador à listagem do programa para as referenciar. Neste caso particular, a linha 167 vai produzir uma série de mensagens de erro do seguinte formato:

a) VERIFICAR deveria ser seguido dum ponto final;
b) AREA é uma palavra reservada utilizada irregularmente;
c) RECORD é também uma palavra reservada utilizada irregularmente.

Adicionalmente, podem existir outras mensagens de erro, dependendo do quão confuso está o compilador na altura em que atinge o fim desta linha. No fim do programa existirá outra mensagem de erro, indicando que o nome do parágrafo VERIFICAR–AREA–RECORD, referenciado na linha 154, não foi definido. Todas estas mensagens resultam de um mesmo erro — falta de hífen.

É muito fácil ter um grande número de mensagens de erro devido a um simples erro em COBOL; a experiência ajudará a definir os mais significativos. Outro problema vulgar que provoca uma cascata de erros é um cabeçalho de divisão incorrecto ou a sua falta. Por exemplo, a falta de um ponto final a seguir à última entrada na ENVIRONEMENT DIVISION pode fazer que o cabeçalho da DATA DIVISION «desapareça», provocandlo resultados caóticos, dado que todas as entradas na DATA DIVISION serão tratadas como pertencendo à ENVIRONMENT DIVISON. Desastres semelhantes podem ser causados pela escrita incorrecta de Environment e de Procedure.

Outros pontos que, de acordo com a experiência do autor, têm frequentemente causado problemas relacionam–se com as margens do lado direito e com o número de linha associado com uma mensagem de erro. Muitos compiladores de COBOL foram escritos para o sistema de perfuração de cartões, e por isso a margem do lado direito é estabelecida na suposição de que o programa se apresenta num formato de «imagem–cartão». Isto significa que o comprimento de uma linha se restringe a um máximo de oitenta caracteres. Ao escrever um programa num terminal *on–line* (directamente), as mesmas restrições não se aplicam e podem escrever–se linhas mais longas. Infelizmente, se o compilador presume que existe uma margem do lado direito, isto pode fazer que as partes finais das linhas sejam perdidas. Nos piores compiladores pode ser listada a totalidade da linha, mas apenas uma parte é analisada pelo compilador.

Outro problema comum é que os alunos aparecem frequentemente com uma mensagem de erro associada a uma linha particular dos seus programas e dizem: «Mas não existe nada de errado nesta linha.» Geralmente, acontece que o erro foi causada pela linha ou linhas precedentes, o que faz que a linha em questão seja interpretada incorrectamente. Por exemplo:

```
201     COMPUTE  DESCONTO-DEVIDO
202              = 0,05 * (PRECO-VENDA * (QUANTIDADE-VENDA — 10)
203     PERFORM  IMPRIMIR-DETALHES
```

O compilador irá, provavelmente, assinalar a linha 203 com erro, talvez com uma mensagem sobre a falta de um operador aritmético ou, o que ainda ajuda menos, indicando que PERFORM foi utilizado incorrectamente. O defeito encontra-se na linha precedente, 202, onde a omissão do parêntese fez que o compilador continuasse a procurar os componentes de uma expressão aritmética.

Para completar esta secção, aqui estão umas linhas gerais que podem ajudar a evitar os erros de sintaxe.

a) Verificar a escrita de palavras reservadas, nomes de campos e nomes de parágrafos. Por exemplo, é muito fácil escrever:

```
05      CUSTOS-VAT-SUMARIO . . .
```

na DATA DIVISION e na PROCEDURE DIVISION:

```
COMPUTE CUSTO-VAT-SUMARIO = . . .
```

b) Verificar se os hífenes foram inseridos no local correcto e foram usados consistentemente. Por exemplo, o parágrafo:

```
PERCENTAGEM-TOTAL.
```

não deve ser referenciado como se segue:

```
PERFORM  PER-CENTAGEM-TOTAL
```

c) Garantir que as palavras reservadas não sejam utilizadas como nomes de dados ou nomes de parágrafos. Geralmente, a utilização de nomes com hífenes correctamente colocados evita este problema.

d) Tome atenção à colocação dos pontos finais nos sítios correctos e apenas aí. Isto é particularmente importante nas declarações IF. Por exemplo:

```
210     IF  REGISTO-VALIDO
211             PERFORM  CALCULAR-PRECO.
212             PERFORM  IMPRIMIR-CONTA
213     ELSE
```

O ponto final na instrução 211 termina a declaração IF e, assim, o ELSE da linha 213 será assinalado como erro.

A falta de pontos finais em certas posições críticas pode originar uma catadupa de erros, como se descreveu anteriormente.

e) O espaçamento é importante, particularmente entre os operadores aritméticos e de relação. Por exemplo:

```
IF COD-CONTA-CLIENTE NOT= «A»
```

irá provocar um erro porque não existe espaço entre NOT e «=».

No entanto, como já foi afirmado, nada substitui a experiência. As tentativas para compilar alguns programas ensinarão mais do que a leitura total de um livro de bons conselhos acerca de como evitar erros de sintaxe.

7.2 — VERIFICAÇÃO («DEBUGGING»)

A verificação é um processo cíclico que consiste nos seguintes passos:

a) Executar o programa compilado com dados de teste;
b) Inspeccionar os resultados obtidos e identificar os erros; se não existirem erros, verificar se são necessários mais dados de teste (ver secção 7.3);
c) Localizar as falhas do programa, ou dos dados, que provocam os erros;
d) Emendar o programa, ou os dados, tentar remover as falhas e voltar ao passo *a*).

Antes de passar às técnicas para a localização de erros, será útil fazer algumas observações gerais sobre a verificação. O primeiro aspecto é que a verificação não é fácil — geralmente requer um esforço mental considerável para a descoberta das falhas. Vale também a pena realçar que existe uma grande probabilidade de, ao remover uma falha, se introduzir uma nova falha. Por isso, o principiante não deve deixar-se desanimar se, por vezes, parecer que cada passo em frente é acompanhado dum aparente passo atrás.

Duas das principais causas das falhas nos programas são erros lógicos e erros nos dados de teste. Para reduzir o número de erros lógicos, recomenda-se que o desenho de programa seja verificado cuidadosamente antes da sua tradução para COBOL. Isto pode ser feito «correndo» manualmente o desenho com diferentes valores de dados. Se o programa de COBOL falha devido a um erro lógico, o desenho deve ser verificado para confirmar se é erro de desenho ou da fase da tradução. Quaisquer alterações significativas ao programa devem ser feitas ao nível do desenho, cuidadosamente verificadas, antes de alterar o programa de COBOL correspondente. Isto é uma disciplina difícil de aceitar, especialmente para programas pequenos, dado que existe um desejo compreensível de «pôr o programa a funcionar». No entanto, isto representa um hábito excelente a estabelecer desde o princípio.

Os dados utilizados para testar um programa devem também ser verificados cuidadosamente, porque um erro de dados é frequentemente a causa de um erro complicado nos resultados. Qualquer aluno que se aproxime do autor para lhe pedir um conselho e afirme que os seus dados estão perfeitos é imediatamente desafiado a produzir uma listagem dos dados, ou dos dados de facto utiliados (com cartões perfurados, por exemplo). Muitos problemas têm sido causados porque a letra «O» foi utilizada em vez de «0» ou porque os dados de *input* foram escritos com o desvio de uma coluna.

Todos os resultados produzidos por um programa devem ser verificados completamente. Tomar a atitude de que os resultados parecem «bastante razoáveis» conduz geralmente a que o programador despreze alguns pequenos erros, mas que são significativos para o programa. Outra tendência é localizar um erro óbvio e aparente nos resultados, corrigi-lo, enquanto outros erros menos óbvios não são detectados. Este erro mais pequeno pode ser útil para diagnosticar um problema óbvio, ou, mesmo que não se relacione com ele, pode também ser corrigido ao mesmo tempo.

Os melhores instrumentos para a verificação são a atenção e a atitude de espírito que não considera nada como certo. Muito frequentemente, os alunos são incapazes de detectar falhas porque se compenetraram de que «é impossível aquele *bit* estar errado». Outra falha comum é a combinação de «isto funcionou da última vez» e «bem, sim, eu fiz realmente uma pequena alteração ao programa».

Tendo feito estas observações gerais acerca da verificação, quais são as técnicas disponíveis para o programador? A técnica mais óbvia é a execução manual do programa — tente seguir o programa por partes, linha a linha, utilizando os dados legíveis pelo programa e anotando todos os resultados intermédios. Este processo pode começar quer a partir do início do programa, quer a partir de um ponto conhecido de êxito, por exemplo, o ponto no qual o último registo foi impresso com êxito.

Uma variante desta técnica é a execução da reversão a partir do ponto do inêxito ou da falha — partindo dos resultados errados, trabalhe no sentido inverso através das linhas de progra-

ma que os originaram. Tente predizer de novo o valor dos resultados intermédios e ver como é que estes possam ter surgido.

Um problema que pode surgir com a execução manual e, particularmente com a execução da reversão é estabelecer qual o ponto exacto da falha. Muitos sistemas de computador têm um utilitário de compensação (buffering) para o *output* que significa que o último registo impresso, ou escrito num ficheiro, pode ser o último, mas foi realmente produzido pelo programa. A experiência com um sistema particular e os conselhos de outros que tenham utilizado irá estabelecer se esta é ou não uma armadilha a ser evitada.

Se estas técnicas falharem, existe então um instrumento adicional muito poderoso, que é tentar convencer alguém de que um programa deve funcionar. Esta técnica é geralmente referida como passar por cima (walktrough) e pode também ser realizada por um grupo de programadores para testar um desenho de programa. Tudo o que é preciso é uma outra pessoa que actue como «advogado do diabo» e que pergunte «porquê?» de cada vez que alguém afirmar que alguma coisa deve acontecer. De acordo com a experiência do autor, esta tem provado ser uma técnica muito útil e, em alguns casos, alguns problemas têm sido resolvidos, apesar de a linguagem de programação não ser muito familiar.

Algumas falhas continuarão difíceis de tratar, mesmo depois de se ter trabalhado cuidadosamente ao longo do programa e de o ter discutido com outras pessoas. Nestes casos, a solução é tentar obter mais informação acerca do modo como o programa está a funcionar. O programador necessita de indagar se os valores esperados dos campos estão correctos e se o programa está a seguir a sequência correcta de parágrafos. A maior parte dos sistemas de COBOL incluem utilitários para depositar (dump) a maioria dos campos, enquanto o programa é executado, e para marcar o fluxo de controlo através de um programa. Uma discussão pormenorizada sobre estes utilitários cai fora do âmbito deste livro; seja como for, eles tendem a variar de sistema para sistema. No entanto, geralmente, estes utilitários tendem a produzir grandes quantidades de *output*, irrelevante na sua maior parte. Este utilitários devem sempre ser usados de maneira cuidadosa e planeada e não em desespero de causa.

Uma forma limitada de depósito e de marcação pode ser providenciada pelo programador, imprimindo os valores dos campos e os nomes dos parágrafos durante a execução de um programa. A maior desvantagem decorrente desta acção é que, ao introduzir declarações adicionais para imprimir estes valores, a estrutura do programa está a ser alterada.

Declarações «Display» (mostrar no dispositivo)

A forma mais simples de imprimir os valores dos campos e dos nomes de parágrafos é utilizar a declaração DISPLAY do COBOL. Isto permite uma mistura de literais não numéricos e de valores de campos a serem impressos no «dispositivo de defeito definido pelo implementador». O local em que irá aparecer o *output* varia de sistema para sistema, mas, geralmente, ele será incluído nalguma espécie de ficheiro orientador associado à execução do programa, separado do ficheiro de *output*.

Para obter o máximo benefício das declarações DISPLAY, a sua posição no programa e os valores a serem apresentados devem ser cuidadosamente projectados. Todo o *output* apresentado deve ser claramente identificado, de preferência através de nomes de parágrafos nos quais o DISPLAY seja inserido e através de nomes de quaisquer campos a serem apresentados. Por exemplo:

```
CALCULO-PRECO.
        DISPLAY  «CALCULO-PRECO»
        DISPLAY  «PRECO  VENDA»,  PRECO-VENDA,
                 «QUANTIDADE-VENDA»,  QUANTIDADE-VENDA
        COMPUTE  PRECO-BASICO  =  PRECO-VENDA  *  QUANTIDADE-VENDA
        DISPLAY  «PRECO-BASICO»,  PRECO-BASICO
        . . .
```

Sempre que CALCULO-PRECO é executado, irá aparecer uma série de mensagens do tipo:

```
CALCULO-PRECO
PRECO-VENDA 0765 QUANTIDADE-VENDA 00100
PRECO-BASICO 76500
```

Isto demonstra que CALCULO-PRECO foi gravado e apresenta os valores correntes de PRECO-VENDA e QUANTIDADE-VENDA. Após PRECO-BASICO ter sido calculado, o seu novo valor é apresentado.

Note que o DISPLAY apresenta *output* não alterado, de modo que, apesar de PRECO-BASICO ter sido descrito como:

```
05      PRECO-BASICO                PIC 999V99.
```

apenas os dígitos iniciais serão apresentados e o ponto decimal implícito será ignorado. Não existe nenhum espaçamento automático entre campos em DISPLAY, e por isso, quaisquer espaços requeridos devem ser incluídos nos literais não numéricos, como anteriormente foi mostrado. Por exemplo:

```
DISPLAY «PRECO-VENDA», PRECO-VENDA,
        «QUANTIDADE-VENDA», QUANTIDADE-VENDA
```

teriam resultado no output na forma

```
PRECO-VENDA0765QUANTIDADE-VENDA00100
```

As vírgulas entre os elementos na listagem são opcionais e são incluídas apenas por questões de legibilidade.

Como foi demonstrado anteriormente, DISPLAY pode ser utilizado para fornecer o equivalente a uma marca, imprimindo os nomes dos parágrafos à entrada dos mesmos, ou a um depósito apresentando os nomes e os conteúdos dos campos. Se o programa reagir de modo diferente após a inclusão de uma ou mais declarações DISPLAY, é óbvio que a lógica do programa foi alterada devido a essas mesmas declarações. Isto não constitui, necessariamente, um problema; a análise da posição e do efeito de cada declaração inserida deve ajudar a descobrir o erro original.

Um problema que afecta geralmente os programadores principiantes é o chamado «ciclo infinito» *(infinite loop)*. O programa utiliza a alocação dos recursos do computador e ou produz pouco ou nenhum *output,* ou produz a mesma linha de *output* repetidas vezes. Isto tem como causa o facto de o programa executar uma sequência de declarações repetidamente (ciclicamente), sem nunca encontrar uma declaração que altere a sequência.

Em COBOL, este problema é geralmente provocado pelo posicionamento incorrecto dos pontos finais. Por exemplo:

```
                .....
                PERFORM  LER-PROXIMO-CONTA-CLIENTE
                PERFORM  PROCESSAR-CONTA-CLIENTE  UNTIL  FIM-DE-CONTA-
CLIENTE
                .....
        PROCESSAR-CONTA-CLIENTE.
                PERFORM  VERIFICAR-CONTA-CLIENTE
                        IF  CONTA-CLIENTE-OK
                                PERFORM  PRECO-E-DESCONTO-CLIENTE
                PERFORM  IMPRIMIR-CLIENTE-DETALHE
                PERFORM  LER-PROXIMO-CONTA-CLIENTE.
                .....
```

O parágrafo VERIFICAR-CONTA-CLIENTE é uma rotina de validação que, entre outras coisas, coloca o valor de CONTA-CLIENTE-OK em verdadeiro para um registo válido e falso para o contrário. A primeira vez que o programa receba um registo inválido entrará em *loop* porque LER-PROXIMO-CONTA-CLIENTE é parte da estrutura IF de CONTA-CLIENTE-OK = verdadeiro. Repare na posição do primeiro ponto final a seguir ao IF.

Geralmente, os *loops* podem ser identificados trabalhando o programa cuidadosamente na verificação dos pontos finais. No entanto, se isto falhar, novamente o próximo passo será tentar obter mais informação acerca do comportamento do programa. O sistema de COBOL utilizado pode incluir uma opção sumário do fluxo de execução (ou de perfil) que irá indicar quantas vezes é que cada declaração do programa foi executada; isto indicará a posição do *loop*. De modo semelhante, um utilitário de marca dará o nome dos parágrafos que formam o *loop*. Se não existirem instrumentos de verificação disponíveis, o programador terá de utilizar a técnica da declaração DISPLAY para investigar quais são as partes do programa que estão a ser executadas repetidamente.

7.3 — TESTES

A execução de testes é o processo destrutivo que consiste em demonstrar a presença de erros no programa e está, como é óbvio, intimamente ligada com a verificação. É impossível provar que um programa não contenha erros testando-o — um teste pode apenas detectar a presença de erros, nunca a sua ausência. Por isso, daí decorre que, contrariamente ao que é costume supor-se, um teste com «êxito» é aquele que permite descobrir um ou mais erros no programa.

Um programa é criação de um programador, e, por isso, ele é a pior pessoa para o tentar destruir. A melhor pessoa para testar o programa é alguém que saiba o que o programa deve executar, mas que não tenha participado na sua escrita. Um bom exercício consiste em trocar um programa que se supõe que funciona com outro resultante do esforço de outro colega e ver quem consegue descobrir mais erros. Na prática, esta abordagem dos testes independentes é frequentemente utilizada para testar programas comerciais.

Uma abordagem à questão dos testes, preferida por muitos alunos, é o «método dos dados aleatórios» — escreva os primeiros *n* registos que lhe venham à cabeça, sendo *n* um número escolhido para impressionar o leitor! Isto permite descobrir alguns erros que podem ser corrigidos, mas no todo apenas os programas mais simples falham tantos erros quantos os que encontram. O que se requer é um modo sistemático de testar programas que permita que cada registo dos dados tenha um propósito definido, maximizando a probabilidade de descobrir erros.

Nesta secção será feita uma tentativa para explanar algumas linhas mestras simples para o teste de programas e para mostrar como estas podem ser aplicadas. Existem duas abordagens básicas para testar um programa:

- caixa-preta, teste que se concentra nos *inputs* e *outputs* do programa ignorando a construção real do mesmo;
- caixa-de-vidro (ou caixa-branca), teste que exercita as instruções do programa.

Utilizadas em conjunto, estas duas abordagens devem maximizar as possibilidades de descoberta de erros.

Uma observação interessante, nascida da experiência prática, é que o esforço para construir sistematicamente dados de teste pode ajudar a descobrir falhas antes da execução do programa.

Testes de caixa-preta

Existem duas técnicas muito úteis, que podem ser aplicadas neste caso, apresentadas, geralmente, pelos nomes deveras pomposos de «parcelamento equivalente» e «análise do valor-fronteira». O leitor não deve deixar-se desencorajar por estes rótulos técnicos — ambas as ideias são facilmente aplicáveis.

O princípio básico do parcelamento equivalente é olhar um por um todos os campos de *input* e identificar todas as «classes» de erros e de dados válidos. Os dados de teste para o programa devem então incluir pelo menos um valor, desenhado com base em cada classe de equivalência. Alguns exemplos de dados de equivalência são apresentados abaixo.

Exemplo 7.1

O campo «tipo-conta-cliente» deve conter uma única letra, «W» ou «R». As classes de equivalência são:

Classe	Tipo-conta-cliente	
1	W	válido
2	R	válido
3	outros valores	inválido

Exemplo 7.2

O «preço-por-artigo» não deve ser menor do que 10 pénis. fazendo uma abordagem simplista, existem somente duas classes de equivalência:

Classe	Preço-por-artigo	
1	menor do que 10 p	inválido
2	não menor do que 10 p	válido

De modo mais realista, convenciona-se que todos os campos numéricos utilizados por um programa devem ter uma classe de equivalência extra de não «não numérico» (a menos que a especificação do programa exclua explicitamente esta possibilidade). Assim, neste caso, as classes seriam:

Classe	Preço-por-artigo	
1	não numérico	inválido
2	não menor do que 10 p	inválido
3	não menor do que 10 p	válido

Exemplo 7.3

A «Quantidade vendida» nunca deve ser inferior a 10 ou superior a 1000.

Classe	Quantidade-vendida	
1	não numérico	inválido
2	menor do que 10	inválido
3	entre 10 e 1000 (inclusive)	válido
4	maior do que 1000	inválido

Exemplo 7.4

O «código-retirada» deve só ser «D» para retiradas em dinheiro ou «C» para cheques. É óbvio que isto é semelhante ao exemplo 7.1, mas suponha agora que a especificação é feita neste sentido: a «quantia-retirada» não deve exceder 250 libras para uma retirada em dinheiro e os cheques devem apenas ser retirados para quantias entre 25 e 1000 libras (incl.). Para identificar as classes de equivalência, torna-se agora necessário considerar os dois campos inter-relacionados: «código-retirada» e «quantia-retirada»:

Classe	Código-retirada	Quantia-retirada	
1	D	não numérico	inválido
2	D	não superior a 250 £	válido
3	D	superior a 250 £	inválido
4	C	não numérico	inválido
5	C	inferior a 25 £	inválido
6	C	entre 25 £ e 1000 £	válido
7	C	superior a 1000 £	inválido
8	nem C nem D	—	inválido

De modo restrito, a classe 6 pode ser dividida em duas classes:

6a	C	entre 25£ e 250£	válido
6b	C	superior a 250£ e até 1000£	válido

Apesar de a fronteira das 250 libras não ter nada a ver com o código–retirada = «C», a lógica do programa pode ficar incorrecta, de tal modo que esta fronteira se torne significativa para «C».

Tendo estabelecido as classes de equivalência para cada campo, ou grupo de campos relacionados, deve construir–se uma série de dados de teste para o registo de *input* que inclua valores de cada classe de equivalência identificados. Cada classe inválida deve ser incluída num registo próprio, para isolar o efeito das combinações de erros. As classes válidas podem ser combinadas com segurança dentro do mesmo registo.

A ideia que está por detrás da análise do valor–fronteira surgiu da observação de que muitos programas falham ao lidar com valores na fronteira entre classes de dados válidos e inválidos. Assim, para aumentar as possibilidades de descobrir um erro (um teste de êxito), escolha valores de dados nestas fronteiras. A aplicação deste método no exemplo 7.2 deve sugerir que o valor na classe 2 deve ser 9 p ou 9,5 p, se se lidar com 0,5 p, e o valor–fronteira mais baixo apropriado para a classe 3 deve ser exactamente 10 p.

A aplicação do mesmo método ao exemplo 7.3 sugere os seguintes valores:

Classe	Quantidade–vendida	
2	9	valor inválido mais alto inferior a 10
3	10	fronteira inferior para valores válidos
3	1000	fronteira superior para valores válidos
4	1001	valor inválido mais baixo superior a 1000

A análise do valor–fronteira pode ser também aplicada a registos de *output*. Para cada campo de um registo de *output*, o *input* apropriado deve ser incluído para cobrir o valor máximo de valores de *output*.

Olhando de novo para o exemplo 7.2, não há, aparentemente, necessidade de testar o valor mais alto de preço–por–item. Mas, se este valor vai ser impresso, o valor máximo deve ser incluído para garantir que ele está correctamente representado no campo de *output*.

Frequentemente se perdem caracteres de *output* devido ao uso incorrecto da edição. Por exemplo:

```
05      PRECO-BASE-CARRO          PIC 9(5)V99.
.....
05      PRECO-BASE-COBRAR         PIC ££,££9.99.
.....
MOVE PRECO-BASE-CARRO TO PRECO-BASE-COBRAR
```

Qualquer valor de PRECO–BASE–CARRO superior a 99999,99 libras será truncado porque o campo PRECO–BASE–COBRAR é muito pequeno. Isto pode beneficiar o vendedor de carros caros, mas não é uma boa prática de programação.

É também aconselhável testar o comprimento máximo dos campos usados para conter nomes e moradas, etc., mesmo que estes não sejam manipulados pelo programa. Isto deve identificar alguns problemas que possam surgir no *layout* do registo de *output*.

É particularmente importante testar os valores máximos de campos construídos. Por exemplo:

```
COMPUTE PRECO-COBRADO = PRECO-VENDA * QUANTIDADE-VENDA
```

onde o PRECO–COBRADO é um campo temporário ou um campo de *output* construído pela multiplicação de dois campos a partir de um registo de *input*. A utilização de PRECO–COBRADO deve ser testada pelo uso dos valores máximos possíveis nos campos de *input* PRECO–VENDA e QUANTIDADE–VENDA.

Os totais são outra área em que, frequentemente, surgem problemas — é muito embaraçoso quando um programa que funcionou perfeitamente durante algum tempo começa de repente a produzir valores sem sentido, quando um acumulador passa um valor crítico. Por isso, é muito importante tentar estabelecer um conjunto de dados de teste para puxar valores que contenham totais, até aos seus valores máximos.

Apesar de, até agora, a ênfase ter sido para valores máximos, é preciso recordar–se de que os valores mínimos também podem causar problemas. Isto é particularmente verdade para campos numéricos onde possa ocorrer a divisão por zero e campos negativos que possam causar grandes problemas de truncagem.

A aplicação das ideias de análise de parcelamento equivalente e valor–fronteira deveria permitir ao programador construir um conjunto de dados de testes para testar sistematicamente o valor dos campos individuais de um programa, No entanto, isto não garante que não surja um problema na prática — um erro num campo de dados pode originar um erro noutro campo de dados. Um programa pode aparentemente lidar com êxito com erros em campos de dados independentemente, mas falhar quando ocorrer uma combinação obscura de erro. A discussão de um método sistemático para lidar com combinações de erros (chamado gráfico de «causa–efeito») está para além dos objectivos deste livro, pelo que algumas ideias simples terão de bastar.

A abordagem mais simples consiste em testar cada combinação de classe de erro; isto é possível em programas com um número pequeno de classes de erro, mas torna–se impraticável se existirem mais do que quatro classes de erro independentes. Para programas com um número maior de classes de erro, torna–se necessário abandonar os testes estritos de caixa–preta e tentar descortinar as causas de erro mais prováveis a partir da estrutura do programa. Por exemplo, se ocorrem dois campos dentro da mesma estrutura IF, pode existir uma interacção entre os seus valores e, por isso, devem incluir–se combinações de erro nos dados de teste. Este tipo de análise leva–nos dos testes do tipo caixa–preta até aos testes caixa–de–vidro.

Em adição aos testes ao nível dos campos, um programa deve também ser testado ao nível do registo. Até agora, os programas estudados neste livro têm tido apenas um ficheiro de *input*, e os casos a testar são:

a) Inexistência de dados de *input*, o «conjunto–vazio» de dados de teste;
b) Um só registo no ficheiro de *input*, o «conjunto–singular» de dados de teste;
c) Vários registos no ficheiro.

Os casos especiais de zero registos ou de um registo podem ser pouco comuns na prática, mas um programa deve funcionar em todas as condições. Os programas com ficheiros de *input* múltiplos (discutidos no capítulo 9) irão requerer que as combinações destes casos sejam testadas.

Testes de caixa-de-vidro

O objectivo deste tipo de testes é construir um conjunto de dados de teste que irá exercitar todas as declarações na Procedure Division de um programa. O modo menos rígido de exercitar o programa é garantir que todas as declarações da Procedure Division foram executados pelo menos uma vez. Trabalhando o programa linha a linha, deve ser possível construir sistematicamente um ficheiro de registos de *input* que contenha valores que irão garantir que cada declaração é executada pelo menos uma vez. Alguns sistemas podem fornecer um sumário do fluxo de execução (ou perfil) opcional, que mostra quantas vezes foi executado cada condição do programa. Sempre que disponível, isto pode ser utilizado para garantir que cada instrução foi executada pelo menos uma vez.

A mera verificação de que cada instrução foi executada não é suficiente, dado que ela pode falhar. A fase seguinte consiste em adicionar registos aos dados de teste, para garantir que cada condição foi verdadeira ou falsa pelo menos uma vez. Isto não é o mesmo que garantir que cada instrução foi executada pelo menos uma vez. Considere o seguinte exemplo simples:

```
IF SALARIO-TRIBUTAVEL > O
        PERFORM  CALCULAR-TAXA.
PERFORM  IMPRIMIR-DETALHES
```

Para garantir que cada instrução foi executada pelo menos uma vez, requer-se apenas que SALARIO-TRIBUTAVEL tenha um valor superior a zero. Este teste não detectará um erro que surja devido ao facto de um valor de SALARIO-TRIBUTAVEL inferior a zero significar que o programa não consegue dar um valor ao campo IMPOSTO-PAGO, dentro de CALCULO-TAXA, e a um empregado não sujeito a imposto ser cobrada uma quantia de imposto arbitrária.

Qualquer programa que contenha condições compostas deveria ter um nível posterior de testes a elas aplicado. Para todas as condições compostas, todas as condições simples que a compõem deveriam ter o valor verdadeiro ou falso pelo menos uma vez. Por exemplo:

```
PERFORM  PROCESSAR-BATCH
        UNTIL FIM-DE-FICH-TRANS OR TIPO-TRANS = 1
```

Para assegurar que cada condição no programa é verdadeira ou falsa pelo menos uma vez, necessitam-se somente de dois conjuntos de dados de teste, por exemplo:

```
FIM-DE-FICH-TRANS  =  verdadeiro     TIPO-TRANS = 1condição = verdadeira
FIM-DE-FICH-TRANS  =  falso          TIPO-TRANS = 2condição = falsa
```

A condição mais rígida de que todas as condições simples que formam uma condição composta devem ser verdadeiro e falso, pelo menos uma vez, requer quatro conjuntos de dados de testes, por exemplo:

```
FIM-DE-FICH-TRANS  =  verdadeiro     TIPO-TRANS = 1condição = verdadeira
FIM-DE-FICH-TRANS  =  verdadeiro     TIPO-TRANS = 2condição = verdadeira
FIM-DE-FICH-TRANS  =  falso          TIPO=TRANS = 1condição = verdadeira
FIM-DE-FICH-TRANS  =  falso          TIPO-TRANS = 2condição = falsa
```

Este exemplo também ilustra outra ponto importante acerca de testes — nem sempre é possível combinar todos os casos de teste numa só execução do programa. No exemplo acima, serão necessárias pelo menos duas execuções do programa, porque a condição FIM-DE-FICH-TRANS só pode ser verdadeira uma vez para um ficheiro simples.

A consideração do exemplo anterior leva-nos a uma outra observação geral de que todos os ciclos de um programa devem ser testados cuidadosamente. Sugere-se que todos os ciclos until sejam testados para zero, uma e várias repetições. Os erros surgem frequentemente nos casos especiais de zero e uma repetição de um ciclo, e, assim, vale a pena testar estes casos como parte da estratégia de teste.

7.4 — ESTUDO DE CASO

Para demonstrar a forma de aplicação de testes discutida na secção 7.3 vai ser utilizado o estudo de caso da secção 6.7. A descrição do problema era como a seguir se apresenta.

O sistema de cobranças de chamadas telefónicas da Rockall Telecom Corporation recebe registos de *input* do seguinte formato:

Código da área	3 caracteres
Número de telefone	6 dígitos
Nome e morada do cliente	40 caracteres
Custo do aluguer	4 dígitos (pénis)
Impulsos gastos	4 dígitos
Impulsos com utilização de operador	3 dígitos

O código da área consiste em ou duas letras seguidas dum dígito ou uma letra seguida de dois dígitos.

Pretende-se que o programa verifique a validade destes dados e que processe os registos válidos calculando o valor que se deve cobrar a cada cliente. Devem ser executadas as seguintes validações nos registos:

a) O código da área está no formato correcto;
b) O custo do aluguer, os impulsos gastos e os impulsos com utilização do operador são todos números válidos;
c) O custo do aluguer não é menor que 8,50 libras.

Se algum destes testes falhar, o detalhe do registo deve ser imprimido com uma ou mais mensagens de erro adequadas.

Para cada um dos registos, calcular a quantia a cobrar pela adopção dos seguintes elementos:

a) O custo do aluguer;
b) Custo dos impulsos gastos; 2 pénis por impulso e 3 pénis por cada impulso com utilização do operador;
c) IVA 12,5%.

O *output* do programa deverá consistir num mapa tabular dos detalhes de cada registo, ao qual se adicionará, para além da quantia a cobrar, uma mensagem relevante de erro.

Esta descrição pode ser utilizada para construir um conjunto de dados para testes de caixa-preta. Para facilidades de referência, os campos que constituem os registos de *input* e *output* serão remetidos para os utilizados, como em baixo se reproduz, na secção 6.7.

registo–tel		
	tel–area	código da área
	tel–area-1	
	tel–area-2	
	tel–area-3	
	tel–numero	número de telefone
	tel–cliente	nome e morada do cliente
	tel–aluguer	custo do aluguer (pénis)
	tel–impulsos	impulsos gastos
	tel–imp–oper	impulsos com utilização de operador
linha–cobrança		
	cob–area	código da área
	cob–numero	número de telefone
	cob–clente	nome e morada do cliente
	cob–aluguer	custo do aluguer (edição, pénis, libras)
	cob–impulsos	impulsos gastos (edição)
	cob–imp–oper	impulsos com utiliz. de oper. (edição)
	conta–cobrança	total a cobrar adicionando o IVA

Provavelmente, a melhor abordagem é pegar nos campos individualmente e aplicar as ideias do parcelamento equivalente e da análise do valor–fronteira.

O primeiro campo é «tel–area», que tem quatro classes de equivalência:

Classe	Tel–area	
1	tel–area-1 não alfabético	inválido
2	tel–area-2 nem alfabético nem numérico	inválido
3	tel–area-3 não numérico	inválido
4	tel–area-4 alfabético e tel–area-2 alfabético ou numérico e tel–area-3 numérico	inválido

Aparentemente, não existem problemas com as fronteiras, mas pode ser necessário combinar combinações de classe de erro.

O segundo campo, «tel–numero», não contém restrições e, por isso, o único teste necessário é preencher os campos com dados para verificar que o campo de *output* correspondente, «cob–numero», é suficientemente grande.

O terceiro campo, «tel–cliente», também não contém restrições e o único teste válido é, assim, garantir que um registo contendo nome e morada que preenche todas as posições é construído para testar «cob–cliente».

O quarto campo, «tel–aluguer», é numérico e tem definido um valor mínimo possível; por isso, as classe de equivalência são:

Classe	Tel–aluguer	
1	não numérico	inválido
2	menor do que 8.50 £	inválido
3	não menor do que 8.50 £	válido

A aplicação da análise do valor–fronteira iria sugerir a utilização do valor 8.49 £ da classe 2 (limite superior da classe de erro) e 8.50 £ da classe 3 (limite inferior de classe válida). Para testar o comprimento do campo de *output* «cob–aluguer», deve também incluir–se um valor máximo equivalente a 99,99 £.

Os dois últimos campos, «tel–impulsos» e «tel–imp–oper», são semelhantes pelo facto de cada uma ter apenas duas classes de equivalência:

Classe Tel–impulsos

1	não numérico	inválido
2	numérico	válido

Classe Tel–imp–oper

1	não numérico	inválido
2	numérico	válido

Para testar a eficácia dos campos de *output* «cob–impulsos» e «cob–imp–oper», devem utilizar–se valores máximos de 9999 e 999, respectivamente.

Olhando para o registo de *output,* existe um campo que não foi mencionado na discussão acima. O campo «conta–cobrança» é construído a partir de «tel–aluguer», «tel–impulsos» e «tel–imp–oper». Por isso, para testar o limite superior de «conta–cobrança», todos estes campos devem ser testados simultaneamente com os seus valores máximos.

É agora possível construir um ficheiro de dados de teste que inclua todas estas classes de erro e fronteiras, e sugere–se o seguinte exemplo. O carácter «-» é utilizado para indicar que o valor nesta posição pode ser qualquer valor válido para o campo. O campo nome e morada do cliente é utilizado para indicar quais os campos que estão a ser testados.

```
9BC ——— PRIMEIRO CARACTER DO CODIGO AREA INVALIDO ——————————
A*3 ——— SEGUNDO CARACTER DO CODIGO AREA INVALIDO ——————————
ABC ——— TERCEIRO CARACTER DO CODIGO AREA INVALIDO ——————————
————— ALUGUER NAO NUMERICO                          ))!) ——————
————— ALUGUER ABAIXO DO MINIMO                      0849 ————
————— IMPULSOS GASTOS NAO NUMERICO                  —— ABCD —
————— OPERADOR UNIDADES NAO NUMERICO                ————————— )*)
ZZ9999999VALORES MAXIMOS EM TODOS OS CAMPOS ——— 99999999999
————— MINIMO ALUGUER PERMITIDO                      0850————
```

No caso dos dois últimos registos, os resultados esperados para «conta–cobrança» devem ser calculados <u>antes</u> da execução do programa e verificados em confronto com os resultados obtidos. Regra geral, em todos os casos em que se espera um resultado determinado, o seu valor deve ser calculado antes da execução do programa.

O ficheiro acima é um conjunto mínimo de dados de teste para testar o programa, aplicando as técnicas de caixa–preta. Podem adicionar–se a este conjunto mais registos, por exemplo, um no qual todos os campos a verificar sejam inválidos e possíveis combinações de erro. Outro bom teste consiste em colocar «tel–impulsos» e «tel–imp–oper» com valor zero para confirmar que esta situação não causa problemas.

A aplicação dos princípios dos testes de caixa–de–vidro a este exemplo não produzirá um conjunto de dados de teste com diferenças significativas. Trabalhando o programa, reparamos que a única instrução que não foi testada convenientemente pelos dados de teste acima referidos é:

```
IF TEL–AREA-2 NOT ALFABETIC
        AND TEL–AREA-2 NOT NUMERIC
```

Olhando para esta condição, existem três estados possíveis:

TEL–AREA-2 <u>numérico</u>	<u>verdadeiro</u>	E <u>falso</u>	= falso
TEL–AREA-2 <u>alfabético</u>	<u>falso</u>	E <u>verdadeiro</u>	= falso
TEL–AREA-2 não <u>numérico</u> nem <u>alfabético</u>	<u>verdadeiro</u>	E <u>verd</u>.	= <u>verd</u>.

O quarto estado, TEL–AREA-2 simultaneamente numérico e alfabético, é, evidentemente impossível. Os dados de teste devem incluir valores de todas estas classes para TEL–AREA-2.

Deixa–se como exercício para o leitor a análise do programa apresentado na secção 6.7 e a construção de um conjunto de dados de teste utilizando os princípios dos testes de caixa–de–vidro.

Exercícios 7

7.1 Utilize os princípios dos testes de caixa–preta para estabelecer valores de testes adequados aos seguintes casos:

a) A taxa de desconto deve ser 10, 15 ou 20. Considere um campo com dois dígitos.

b) O valor mínimo de média–hora é 2,10 £ e o máximo é 7,25 £. Considere três dígitos — pénis.

c) Se o código–tarifa for «N» (negócio), as primeiras duzentas unidades consumidas são cobradas a 15 p por unidade e quaisquer unidades adicionais a 5 p por unidade. Para o código–tarifa «D» (doméstico), as primeiras cem unidades são cobradas a 10 p por unidade e quaisquer unidades adicionais a 8 p por unidade. Suponha que código–tarifa consiste numa única letra e que unidades–gastas é um campo de quatro dígitos.

7.2. Aplique os princípios de teste de caixa–de–vidro para estabelecer dados de teste para verificar o seguinte extracto de programa:

```
IF TIPO–CLIENTE = «D»
        PERFORM  DESCONTO–DEALER
ELSE
        IF TIPO–CLIENTE = «T» OR TERMOS–ESPECIAIS
                PERFORM  DESCONTO–TRANSAC
ELSE
        IF QUANT–CLIENT > 50
        PERFORM  QUANT–DESCONTO.
PERFORM  PRODUZIR–FACTURA.
```

7.3 Demonstre como a aplicação dos testes de caixa–de–vidro iria mostrar que o seguinte extracto de COBOL:

```
QUANTIDADE–DESCONTO.
        IF  QUANTIDADE–VENDAS  <  VOLUME–NIVEL–DESCONTO
                IF  CLIENTE–TRANSACCAO
                        PERFORM  DESCONTO-5
        ELSE
                PERFORM  DESCONTO-10.
```

não é uma tradução correcta do fragmento de desenho:

```
quantidade–desconto
        if  quantidade–vendas  <  volume–nivel–desconto
                if  cliente–transacção  then
                        desconto-5
                endif
```

```
else
        desconto-10
endif
```

Para revisão e para confirmar que a questão foi totalmente entendida: faça uma tradução correcta do desenho de programa para COBOL e um fragmento de desenho correspondente ao extracto de COBOL apresentado.

7.4. Analise o estudo de caso discutido na secção 5.6 e estabeleça um conjunto de dados de teste para este problema o mais completo possível.

7.5. Aplique as técnicas discutidas neste capítulo para testar sistematicamente um dos programas sugeridos nos exercícios de programação 6.

8. ESTRUTURA DE PROGRAMA

O principal objectivo deste capítulo é levar o leitor até à estrutura de um programa de COBOL. Até agora, os programas analisados, e os que se espera tenham sido tentados pelo leitor, têm-se baseado no esqueleto de programa *standard* apresentado no apêndice 1. Este capítulo irá explicar como este esqueleto é, de facto, construído, preparando o caminho para a análise de estruturas de programa mais complexas.

As quatro divisões que constituem um programa de COBOL foram introduzidas de um modo breve no capítulo 3, mas cada uma das secções irá ser explicada mais pormenorizadamente neste capítulo. Uma das características mais importantes de um programa de COBOL que se ocultou ao leitor até agora é a descrição do tratamento de ficheiros. A razão que está por detrás da estrutura do *input* e do *output* utilizada no esqueleto de programa é apresentada na secção 8.4.

Tendo explicado as características básicas do tratamento de ficheiros em COBOL, podem introduzir-se algumas técnicas mais avançadas. Na secção 8.5 aprecia-se a criação de linhas de *output* melhor formatadas para impressora e na secção seguinte faz-se a introdução do conceito de utilização de ficheiros múltiplos de *input*. Finalmente, o estudo de caso ilustra a utilização de algumas das técnicas expostas neste capítulo.

8.1 — «IDENTIFICATION DIVISION»

Os desenhadores originais do COBOL criaram-na para fornecer unicamente a identificação e outras informações documentais acerca do programa. A única entrada obrigatória nesta divisão é a identificação do programa, que é utilizada pela maioria dos sistemas para fornecer a ligação entre programa *source* (fonte em COBOL) e programa objecto (numa linguagem de nível inferior). O formato geral do identificador do programa é:

```
PROGRAM-ID. nome-programa.
```

O nome-programa pode teoricamente ser qualquer um construído de acordo com as regras sobre nomes definidos pelo utilizador (ver secção 3.2). Na prática, ele é normalmente restringido no seu comprimento, e possivelmente no formato, pelo implementador ou por um sistema particular de COBOL. Esta restrição é, normalmente, imposta porque o nome do programa é utilizado como nome ou parte do nome para um ficheiro contendo o programa objecto e cada sistema operativo impõe as suas próprias regras peculiares para nomes-ficheiro.

Existe uma série de outras possíveis entradas que podem aparecer na IDENTIFICATION DIVISION, mas nenhuma delas é obrigatória — simplesmente fornecem documentação adicional para o programa. São possíveis as seguintes entradas:

```
AUTHOR. entrada-comentario.
INSTALATION. entrada-comentario.
DATE-WRITEN. entrada-comentario.
```

```
DATE-COMPILED. entrada-comentario.
SECURITY. entrada-comentario.
```

Uma «entrada-comentario» é uma sequência qualquer de caracteres que podem ocupar um número qualquer de linhas — o fim de uma entrada será reconhecido pela palavra reservada que inicia a próxima entrada.

A data colocada em DATE-COMPILED será trocada pela data corrente quando for produzida uma listagem de compilação do programa. As outras entradas-comentário são simplesmente testadas na sequência em que aparecem no programa-fonte.

Abaixo fornece-se um exemplo completo de uma IDENTIFICATION DIVISION utilizando todas a opções dadas acima:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. EXEMPLO1.
AUTHOR. NUNO AZEVEDO
INSTALLATION. CIGRECI, LDA
DATE-WRITEN. JANEIRO 1985
DATE-COMPILED. HOJE
SECURITY. NENHUMA
```

Quando o programa é listado após uma tentativa de compilação, HOJE irá ser substituído pela data corrente do sistema.

8.2 — «ENVIRONMENT [MEIO] DIVISION»

Esta divisão deve conter todos os pormenores acerca do meio envolvente em que o programa irá correr. Os desenhadores originais do COBOL definiram que todas as situações peculiares inerentes ao tipo particular de computador usado estivessem contidas nesta divisão. Assim, o transporte do programa de um tipo de computador para outro envolveria apenas a modificação da ENVIRONMENT DIVISION. Esta divisão tem duas secções: CONFIGURATION, para especificar o *hardware* (máquina física) do sistema utilizado, e INPUT-OUTPUT, que, tal como o próprio nome sugere, trata das ligações entrada e saída em ficheiros externos.

«Configuration Section»

O mínimo requerido para a CONFIGURATION SECTION é que SOURCE-COMPUTER e OBJECT-COMPUTER sejam especificados. O parágrafo computador-fonte fornece o nome da máquina na qual o programa irá ser compilado, enquanto o parágrafo computador-objecto fornece o nome da máquina na qual o programa-objecto resultante irá ser executado. Os computadores fonte e objecto são, normalmente, a mesma máquina, mas isto permite a um programa a possibilidade de ser compilado numa máquina e executado noutra, o que é, às vezes, chamado «compilação cruzada». Esta facilidade seria muito útil se a máquina definida no parágrafo computador-objecto não fosse suficientemente grande para comportar um compilador COBOL, provavelmente devido a uma insuficiência de capacidade de memória, mas seria utilizada para correr programas já compilados.

O que é actualmente definido para os parágrafos computador-fonte e computador-objecto é «definição do implementador». Isto significa que qualquer pessoa que produza um compilador COBOL pode especificar qual a informação que iria fazer parte destes parágrafos — não é fornecido nenhum formato *standard*. Para executar um programa de COBOL é, assim, necessário descobrir quais são as definições correctas para estes parágrafos, de acordo com o computador que esteja a ser utilizado.

Em adição aos parágrafos computador–fonte e computador–objecto, existe ainda uma possibilidade para se definirem SPECIAL–NAMES na CONFIGURATION SECTION. Esta facilidade não é normalmente utilizada pelo programador iniciado, mas é mencionada de forma breve, pois pode vir a ser encontrada durante a leitura de outros livros ou durante o estudo de programas mais avançados. A utilização mais comum do parágrafo SPECIAL–NAMES é para associar o nome de uma característica de *hardware* particular da máquina utilizada, com um «nome–mnemónica» a ser utilizado no programa.

Por exemplo, suponha que um sistema de COBOL particular utiliza o nome CONTROL-1 para a consola do computador. Então, o programador pode escolher um nome para o associar a este dispositivo, a ser executado ao longo do programa, pela utilização do parágrafo SPECIAL–NAMES. Por exemplo:

```
SPECIAL–NAMES.
        CONTROL-1  IS  OPERATORS–CONSOLE.
```

Para o envio de uma mensagem para o operador, é possível utilizar uma variante da declaração DISPLAY introduzida na secção 7.3. Por exemplo:

```
DISPLAY «INICIO DA FASE 2» UPON OPERATORS–CONSOLE
```

A mensagem «INICIO DA FASE 2» irá ser escrita na consola do operador quando esta instrução estiver a ser executada.

Agora, se o programa é transferido para outro sistema onde a consola do operador é chamada TYPEWRITER, uma simples alteração na ENVIRONMENT DIVISION permitirá ao programa continuar o seu trabalho. O parágrafo SPECIAL–NAMES necessitará de ser alterado para:

```
SPECIAL–NAMES.
        TYPEWRITER  IS  OPERATORS–CONSOLE.
```

A alteração ao programa é confinada à ENVIRONMENT DIVISION e quaisquer instruções de DISPLAY que se refiram a OPERATORS–CONSOLE mantêm–se inalteradas.

«'Input'-'output' Section»

A mínima definição requerida para a INPUT–OUTPUT SECTION é um parágrafo FILE–CONTROL contendo uma declaração SELECT para cada ficheiro de *input* ou de *output* utilizado pelo programa. As instruções de selecção preenchem um papel semelhante ao das entradas no parágrafo SPECIAL–NAMES; associam aos ficheiros os nomes utilizados no programa. Assim, se no programa existir um ficheiro chamado FICH–TRANS, ele pode ser associado a um ficheiro chamado TRANS-2505 através da instrução SELECT:

```
SELECT FICH–TRANS ASSIGN TO TRANS-2505.
```

FICH–TRANS é um nome de ficheiro interno (ou lógico) e TRANS-2505 é um nome de ficheiro externo (ou físico). As regras para a construção de nomes de ficheiros externos variam de implementação para implementação, mas as regras para construir nomes de ficheiro internos são *standard*. De novo, os programas são mais «transportáveis», visto que só os nomes de ficheiros externos têm de ser alterados aquando da transferência de um programa de COBOL de um sistema para outro.

Um exemplo de uma ENVIRONMENT DIVISON completa para um sistema de COBOL imaginário é dado abaixo.

```
ENVIRONMENT  DIVISON.
CONFIGURATION  SECTION.
```

```
SOURCE-COMPUTER. MM1.
OBJECT-COMPUTER. MM1.
SPECIAL-NAMES.
        MM-CONTROL IS OPERATORS-CONSOLE.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
        SELECT FICH-TRANS ASSIGN TO MM-INPUT.
        SELECT FICHOUT ASSIGN TO MM OUTPUT.
```

8.3 — «DATA DIVISION»

A DATA DIVISION encontra-se dividida em duas partes principais; a FILE SECTION, que contém descrições de todos os ficheiros de *input/output* utilizados pelo programa, e a WORKING-STORAGE SECTION, que descreve todos os campos de dados temporários requeridos pelo programa. Em programas mais avançados podem aparecer secções adicionais para ligações de subprogramas e ainda para outros aspectos, mas, de acordo com os propósitos deste livro, apenas duas secções serão consideradas em pormenor.

«File Section»

Todos os ficheiros descritos na FILE SECTION consistem numa descrição do ficheiro seguida de uma ou mais descrições de registos. Uma descrição de um ficheiro inicia-se com os caracteres FD seguidos por um nome interno de ficheiro, a cláusula LABEL RECORDS e, possivelmente, outras cláusulas de definição do ficheiro, sendo a descrição finalizada com um ponto final. O nome interno de ficheiro é o nome escolhido que irá ser utilizado ao longo do programa e deverá ter sido definido na declaração SELECT, do parágrafo FILE-CONTROL da INPUT-OUTPUT SECTION da ENVIRONMENT DIVISION.

A cláusula LABEL RECORDS fornece informação acerca da estrutura do ficheiro. Os ficheiros que existam em banda ou disco magnético têm, normalmente, um registo de *header* e *trailer* definidos para marcar, respectivamente, o início e fim do ficheiro. O formato destes registos especiais varia de sistema para sistema. Se o ficheiro for descrito como:

```
LABEL RECORDS STANDARD
```

então o ficheiro contém as marcas *standards* para *header* e *trailer* para o sistema que está a ser utilizado. Uma alternativa é:

```
LABEL RECORDS OMITTED
```

que indica que o ficheiro não tem marcas físicas para início e fim de ficheiro, podendo isto significar que o *input* é feito directamente a partir de cartões perfurados ou via teclado ou o *output* executado directamente para uma impressora.

Qual destas alternativas deve ser usada para um tipo particular de ficheiro varia de implementação para implementação. Assim, para um curso de programação, deve ser consultado o manual do sistema que vai ser utilizado, ou então os pormenores fornecidos pelo monitor do curso.

Outras cláusulas opcionais que podem aparecer numa descrição de ficheiro dizem respeito a dois outros aspectos da estrutura física do ficheiro. A cláusula BLOCK CONTAINS especifica o número de caracteres ou registos armazenados num bloco para um ficheiro em disco ou em banda. A cláusula RECORD CONTAINS permite a um ficheiro conter registos de comprimento

variável. Não se pretende que o programador iniciado vá utilizar nenhuma destas cláusulas, mas, em todo o caso, elas são introduzidas normalmente com interesse documental.

Finalmente, existe a cláusula DATA RECORDS, que permite ao programador listar os nomes de diferentes registos pertencentes a um ficheiro com diferentes tipos de registos. No entanto, como as descrições dos registos devem seguir-se imediatamente após a descrição do ficheiro, esta cláusula é, também, pouco importante, a não ser para documentação.

Abaixo é fornecido um exemplo de uma descrição de ficheiro completa, utilizando todas estas opções.

```
FD      FICHEIRO-TRANS
        BLOCK CONTAINS 1000 TO 2000 CHARACTERS
        RECORD CONTAINS 300 TO 1500 CHARACTERS
        LABEL RECORDS STANDARD
        DATA RECORDS ARE TRANS-BATCH-HEADER TRANS-DETAIL.
```

A esta descrição de ficheiro deveriam seguir-se descrições de registo para TRANS-BATCH-HEADER e TRANS-DETAIL.

Note que uma descrição de ficheiro consiste em «FD» seguido do nome do ficheiro e de quaisquer cláusulas descritivas apropriadas e termina com um ponto final. Após detectar este ponto final, o compilador de COBOL esperará encontrar a descrição dos registos. Um erro muito comum é colocar o ponto final após o nome do ficheiro — isto irá originar um ou vários erros de sintaxe.

Resumindo, a mínima descrição de registo possível para servir os propósitos deste livro consiste em «FD» seguido de um nome interno de ficheiro, da cláusula LABEL RECORDS e de um ponto final. Por exemplo:

```
FD      FICH-CLIENTE
                LABEL RECORDS OMITTED.
```

A descrição do ficheiro deve ser seguida de uma ou mais descrições de registos. Todas as descrições de registos até à descrição seguinte de ficheiro ou ao fim da FILE SECTION serão associadas como pertencendo ao ficheiro corrente. Quando um ficheiro tem associadas mais do que uma definição de registo, surgem problemas — eles serão analisados na secção 8.6, abaixo. A descrição de registo é uma estrutura de dados com número de nível 01. A descrição de registo mais simples é um único campo com número de nível 01 e uma *picture*. Por exemplo:

```
FD      FICHOUT
        LABEL RECORDS OMITTED.
01      LINHA-OUT                       PIC X(120).
```

Isto vai criar uma área de 120 caracteres associada ao ficheiro FICHOUT, que pode ser acedida pelo nome de registo LINHA-OUT.

Para ficheiros de *input*, a descrição de registo é normalmente estruturada de forma que seja possível aceder a campos que existam dentro desse registo. A estrutura e o tipo de campos dentro dum registo foram apresentados nos capítulos 3 a 5, restando, assim, muito pouco para adicionar.

Uma descrição importante é que a cláusula VALUE não deve ser utilizada para dar valores iniciais a nenhum campo de dados da FILE SECTION. Isto não invalida o uso de nomes-condição onde VALUE é utilizada de forma diferente.

«Working–storage Section»

Esta secção do programa contém descrições para todos os campos necessários para armazenar valores durante a execução do programa. Estes podem ser mais ou menos categorizados nas classes fornecidas abaixo.

a) Variáveis de estado: descrevendo o estado corrente do programa. O único exemplo utilizado até agora é o da marca de FIM–DE-<nome–de–ficheiro>, mas outros exemplos irão aparecer mais tarde neste livro.

b) Registos de *output:* estruturas de dados utilizadas para construir um ou vários registos de *output* para eventual transmissão para o ficheiro de *output.*

c) Campos de dados temporários: campos utilizados para armazenar os valores que se utilizam para reter valores intermédios de cálculos, acumulações, etc.

d) Cabeçalhos e outras constantes: áreas de memória que são atribuídas a valores no início do programa e que irão manter esses valores ao longo da execução do programa.

As regras para construção destas estruturas de dados foram já discutidas nos capítulos 3 e 5, mas, neste ponto, vale a pena fazer algumas observações gerais acerca da WORKING–STORAGE SECTION.

Segundo a experiência do autor, um dos blocos mais inconsistentes diz respeito ao valor inicial dos campos de dados. Quando um programa inicia a sua execução, os valores de todos os campos são indefinidos, a menos que se tenha fornecido um valor inicial através da utilização da cláusula VALUE na DATA DIVISION. Um campo que não tenha valor inicial pode ser carregado com um valor, utilizando–se as instruções MOVE, COMPUTE ou READ na PROCEDURE DIVISION.

Vale a pena repetir que convém que se tenha cuidado com o uso da cláusula VALUE porque ela normalmente causa problemas. Por exemplo, suponha que o programa contém a seguinte descrição de um campo:

```
04      TOTAL–VENDAS          PIC 9(6) VALUE 0.
```

Isto significa que quando o programa iniciar a sua execução o valor de TOTAL–VENDAS será colocado a zeros. Se o valor for alterado pelo programa, este não será automaticamente recolocado em zero, o que só irá acontecer se especificamente existir uma instrução no programa para tal, por exemplo através de uma instrução de MOVE. Um ponto importante a recordar é que a clausula VALUE só atribui a um campo um valor inicial apesar de existirem outras linguagens de programação que defendam o conceito de constante fixa.

Olhar para certos campos como constantes que terão um valor fixo durante o programa é uma boa prática de programação. Esta ideia foi já utilizada na construção de cabeçalhos, mas pode ser também utilizada noutras circunstâncias, especialmente em programas maiores. Os programas que tratem facturas terão de considerar o IVA e um programa muito mal escrito terá a taxa corrente de 15% que aparece em vários cálculos de facturação. Por exemplo:

```
COMPUTE IVA–SUBTOTAL = 0.15 * TOTAL–VENDAS
```

Se o Governo mudar a taxa do IVA para 12,5% na semana seguinte, este programa será difícil de alterar porque cada referência separada à taxa do IVA terá também de ser alterada.

No entanto, se se declarar um campo chamado TAXA–IVA, dado um valor de 15, e se não se alterar no programa, ele pode ser utilizado como uma constante. Por exemplo:

```
05      TAXA–IVA                    PIC 99V9      VALUE 15.
.....
COMPUTE IVA–SUBTOTAL = 0.01 * TAXA–IVA * TOTAL–VENDAS
```

Uma alteração na taxa do IVA irá agora requerer uma simples alteração no programa, existindo assim uma possibilidade muito menor de erro ao executar esta emenda. Mesmo que a taxa do IVA seja alterada para 13,75%, a alteração continua a limitar-se a um só campo.

Outro ponto que determinará um bom estilo de programação é manter juntos e na mesma ordem em todos os programas escritos os campos de dados similares. Isto poupa tempo ao executar-se uma pesquisa através da WORKING-STORAGE SECTION. Por exemplo, poder-se-ia adoptar o seguinte formato:

```
WORKING-STORAGE  SECTION.
01      VECTOR-ESTADO.
        05      FIM-FICHEIRO-ENCOMENDA     PIC X.
                88      FIM-DE-FICHEIRO-ENCOMENDA          VALUE «E».
*       Podem juntar-se ao VECTOR-ESTADO outras variáveis, tais
*       como indicadores de erro e boolianos
01      CAMPOS-TEMPORARIOS.
        05      ALCATIFA-TOTAL          PIC 9(6)V99.
        05      ENTREGA-TOTAL           PIC 999.
*       Podem juntar-se a CAMPOS-TEMPORARIOS as descricões de todos
*       os campos temporários.
01      DADOS-CONSTANTES.
        05      TAXA-IVA                PIC 99V9        VALUE 15.
*       Podem colocar-se em DADOS-CONSTANTES quaisquer definições
*       de constantes utilizadas no programa
*       Podem então ser definidos todos os registos relacionados
*       com o ficheiro de output, começando por exemplo
*       pelos cabeçalhos
01      CABECALHOS-PAGINA.
        05      FILLER                  PIC X(36)       VALUE SPACES.
        05      FILLER                  PIC X(25)       VALUE
                «NOME E MORADA DO CLIENTE».
        ....
01      CABECALHOS-DET.
        05      FILLER                  PIC X(5)        VALUE «TOTAL».
        ....
01      LINHA-DET.
        05      FILLER                  PIC X.
        05      NUMERO-DETALHE          PIC 9(4).
        .....
```

Não interessa qual a ordem em que aparecem estas estruturas de dados, mas, se se utilizar um estilo consistente, a programação torna-se mais fácil.

Isto pode parecer desnecessário para programas pequenos, mas para começar é melhor ganhar logo bons hábitos, em vez de «desaprender» maus hábitos mais tarde.

O COBOL *standard* inclui um número de nível especial, 77, para campos «independentes» que aparecem no príncipio da WORKING-STORAGE SECTION. Não se recomenda o uso do número de nível 77 porque é muito melhor juntar os vários tipos de campos tornando desnecessário um número de nível especial — o nível 01 com uma *picture* tem o mesmo efeito. No entanto, outros livros e programas podem incluir esta configuração.

8.4 — «PROCEDURE DIVISION»

A PROCEDURE DIVISION é a parte executável, ou algorítmica, de um programa de COBOL. A execução de um programa começa com a primeira declaração do primeiro parágrafo da PROCEDURE DIVISION e continua sequencialmente até que apareça a declaração especial STOP RUN. Isto é tratado com mais pormenor em «Parar um programa», adiante.

A utilização de PERFORM e PERFORM ... UNTIL irá alterar o «fluxo de controlo» de um programa, obedecendo às instruções de um parágrafo noutro ponto do programa. No entanto, eventualmente, a execução continua sequencialmente, a menos que ocorra uma situação de erro ou que se execute um STOP RUN.

Considere o primeiro parágrafo de um esqueleto de programa *standard:*

```
PROGRAMA-PRINCIPAL.
        PERFORM  ESTADO-INIC
        PERFORM  <processamento-principal>
        PERFORM  FECHO.
```

Não obstante o número de declarações PERFORM contidas nos parágrafos referidos, estas três declarações devem sempre ser executadas na sequência apresentada.

Afirmou-se no capítulo 1 que todos os programas têm três partes principais: início, processamento e fim, e esta estrutura é referida na estrutura de controlo de nível mais alto apresentada acima. Para compreender a construção dos parágrafos ESTADO-INIC e FECHO, juntamente com a implementação de ler-próximo-<nome-de-ficheiro-input> e escrever-<nome-registo>, é necessário observar com alguma minúcia os verbos mantidos afastados do leitor até agora. Particularmente, os verbos que dizem respeito ao tratamento de ficheiros precisarão de ser cuidadosamente investigados. Isto deverá preparar o caminho para o desenvolvimento de programas com requisitos mais complexos para o processamento de dados.

Parar um programa

O formato da declaração STOP utilizada no esqueleto de programa *standard* é:

```
STOP  RUN
```

A execução desta declaração indica que o programa acabou e que o sistema operativo utilizado para controlar o programa deve apagá-lo e continuar com outra tarefa. É possível que um programa contenha várias declarações SOTP RUN, mas, geralmente, isto denuncia um desenho pobre. Recomenda-se vivamente que um programa tenha só um ponto de saída e, por isso, apenas uma declaração STOP RUN.

Uma forma alternativa para a declaração STOP é:

```
STOP  «<mensagem>»
```

A execução desta declaração irá parar o programa temporariamente, mandando a <mensagem> para a consola do operador. O programa pode ser recomeçado pelo operador do computador, geralmente depois de ter empreendido uma acção adequada. Por exemplo, um programa de salários pode requerer uma biblioteca especial para que os talões de salários sejam descarregados na impressora. Assim, o programa pode incluir a declaração:

```
STOP «CARREGAR BIBLIOTECA SALARIOS»
```

Quando o operador tiver completado o carregamento da biblioteca, o programa pode ser reiniciado.

Abertura e fecho de ficheiros

Qualquer ficheiro que seja utilizado por um programa de COBOL deve ser aberto antes que os seus conteúdos possam ser lidos ou escritos. A área de registos associada ao ficheiro é também inacessível ao programa até que o ficheiro tenha sido aberto. Qualquer tentativa para utilizar o ficheiro ou a sua área de registo associada, antes que o ficheiro tenha sido aberto, irá provocar um erro de execução e um fracasso do programa.

Ao abrir um ficheiro, é necessário especificar se ele vai ser um ficheiro de *input* ou *output*. Uma simples declaração OPEN pode ser utilizada para abrir um número de ficheiros qualquer, quer sejam de *input,* quer de *output*. A forma geral desta declaração é:

OPEN INPUT <lista–input> OUTPUT <lista–output>

onde <lista–input> contém os nomes de um ou mais ficheiros a ser abertos para *input* e <lista–output> contém os nomes de um ou mais ficheiros a ser abertos para *output,* por exemplo:

a) OPEN INPUT FICH–CLIENTE OUTPUT FICHOUT
b) OPEN INPUT FICH–TRANS FICH–MESTRE–ANTIGO
OUTPUT FICH–MESTRE–NOVO FICH–ERROS

Quando a instrução OPEN é executada, é feita uma tentativa para abrir cada ficheiro de cada vez. No caso de um ficheiro de *input,* se o mesmo não existir ou não se encontrar disponível para o utilizador, irá ocorrer um erro de execução. Se já existir o ficheiro de *output,* então a acção a tomar irá depender do sistema que está a ser utilizado. Normalmente, é criado um ficheiro novo vazio para cada ficheiro de *output* aberto. Depois de se completar com êxito uma instrução OPEN, todos os ficheiros e as suas áreas de registo associadas ficarão acessíveis ao programa.

A declaração OPEN permite opções para além do ficheiro de *input* e *output,* mas não são relevantes para o desenvolvimento de programas simples.

Quando o programa tiver acabado de utilizar o ficheiro, este deve ser fechado, o que o vai libertar do programa. Numa simples declaração CLOSE pode ser fechado um número qualquer de ficheiros, não sendo necessário especificar se os ficheiros foram utilizados para *input* ou *output.* Por exemplo:

a) CLOSE FICH-CLIENTE FICHOUT
b) CLOSE FICH-TRANS FICH-MESTRE-ANTIGO
FICH-MESTRE-NOVO FICH-ERROS

Após a execução de uma declaração CLOSE, os ficheiros e as suas áreas de registo associadas deixarão de estar acessíveis ao programa e qualquer tentativa para a sua utilização resultará num erro de execução.

Olhando para o esqueleto de programa *standard* (ver apêndice 1), podemos observar que uma declaração OPEN é incluída em ESTADO-INIC, para abrir os ficheiros requeridos pelo programa, e uma declaração CLOSE está incluída em FECHO para libertar estes ficheiros.

Leitura de registos de um ficheiro

Repetindo um ponto prévio — antes que qualquer coisa possa ser lida de um ficheiro, é preciso abri-lo para *input.* Se o ficheiro não foi correctamente aberto, uma tentativa para a sua leitura originará um erro de execução.

O mecanismo de leitura do COBOL tem como base os registos e utiliza uma aproximação de «leitura-falhada» para a condição de fim de ficheiro. O facto de a leitura ter como base os registos significa que não é possível ler caracteres ou números individuais do ficheiro, mas apenas registos completos tal como o especificado na FILE-SECTION da DATA DIVISION.

A abordagem leitura-falhada significa que, quando uma declaração READ é executada, ela ou copia um registo para a área de *input* associada, ou executa uma sequência de instruções específicas para lidar com a condição fim de ficheiro. Depois da execução desta sequência de instruções, diz que o ficheiro está na condição «at–end» e que a área de *input* do ficheiro está inacessível para o programa. Se um ficheiro se encontra na posição «at–end», qualquer futura tentativa de leitura sobre ele irá causar um erro de execução.

A forma mais largamente utilizada da funcão READ é:

```
READ <nome-ficheiro-input>
        AT END <acção-em-fim-ficheiro>.
```

onde <acção-em-fim-ficheiro> é uma sequência de «declarações imperativas» que terminam com um ponto final. As declarações imperativas são um segmento das declarações disponíveis no COBOL e são definidas nos manuais apropriados.

O modo recomendado para lidar com a condição de fim de ficheiro é apresentado no esqueleto de programa apresentado no apêndice 1. O método é resumido abaixo.

a) Defina uma condição de END-OF-<nome-ficheiro-input> na WORKING-STORAGE SECTION da DATA DIVISION, como se segue:

```
05      END-<nome-ficheiro-input>                PIC X.
        88      END-OF-<nome-ficheiro-input>     VALUE «F».
```

A condição END-OF-<nome-ficheiro-input> será verdadeira se e só se o valor «F» for movido para o campo de dados associado END-OF-<nome-ficheiro-input>.

b) Inicialize a condição END-OF-<nome-ficheiro-input> com o valor falso no início do programa; no esqueleto do programa isto é feito em ESTADO-INIC. O valor inicial pode ser qualquer coisa diferente de «F»; no esqueleto do programa foi escolhido o espaço para valor inicial e, assim, a inicialização é:

```
MOVE SPACE TO END-<nome-ficheiro-input>
```

c) Sempre que se executa uma declaração READ, certifique-se de que a condição END-OF-<nome-ficheiro-input> é tomada verdadeira pelo membro AT END do READ. Assim, o esqueleto do programa contém:

```
LER-PROXIMO-<nome-ficheiro-input>.
        READ <nome-ficheiro-input>
                AT END MOVE «F» TO END-<nome-ficheiro-input>.
```

d) Certifique-se de que cada declaração READ somente é executada se END-OF-<nome-ficheiro-input> for falso.

O desenho de programa utilizado usualmente é:

```
ler-proximo-<nome-ficheiro-input>
until fim-de-<nome-ficheiro-input> do
. . . .
        ler-proximo-<nome-ficheiro-input>
enduntil
```

é construído de forma que seja sempre seguro ler o ficheiro. Se não existirem registos no ficheiro de *input*, ler-proximo–<nome-ficheiro-input> antes do until estabelecerá fim-de–<nome-ficheiro-input> como verdadeiro e o ciclo não será executado. De outro modo, o ciclo continua até que

ler-proximo-<nome-ficheiro-input> no fim do ciclo mude fim-de-<nome-ficheiro-input> para verdadeiro, terminando o ciclo.

Existe uma forma alternativa para a declaração READ:

```
READ <nome-ficheiro-input> INTO <armazenamento-registo>
     AT END <acção-em-fim-ficheiro>.
```

Se o registo for lido com êxito de um ficheiro, ele será copiado da área de *input* para a área definida em <armazenamento-registo>, que deve ser na WORKING-STORAGE SECTION. Um exemplo da utilização desta forma alternativa é apresentada na secção 9.4.

Escrita num ficheiro

Antes que quaisquer registos possam ser escritos num ficheiro ele deve ser aberto para *output*, donde qualquer tentativa para escrever num ficheiro por abrir irá originar um erro de execução.

A forma mais simples da declaração WRITE é:

```
WRITE <nome-registo-output>
```

onde <nome-registo-output> deve ser o nome de um registo associado a um ficheiro. Isto significa que a FILE SECTION da DATA DIVISION deve incluir um descrição da seguinte forma:

```
FD    <nome-ficheiro-output>
      LABEL RECORDS OMITTED.
01    <nome-registo-output>
      ....
```

O efeito desta declaração WRITE é transferir os dados armazenados na área de *output*, referida por <nome-registo–output>, para o ficheiro ao qual está associado <nome-ficheiro-output>.

Repare que uma declaração WRITE utiliza o nome de um registo, enquanto ler se refere ao nome de um ficheiro.

Se existirem vários registos descritos para um dado ficheiro, é então necessário uma declaração WRITE diferente para cada um deles. No entanto, estes registos não são independentes — existe apenas uma área de *output* definida para o ficheiro que é partilhada por todos os registos. É como se se descrevesse um registo e depois se considerassem todos os registos subsequentes como uma redefinição do registo original.

O estilo recomendado para a escrita de registos, utilizado no esqueleto de programas, é definir um único registo associado ao ficheiro de *output*, utilizando uma forma alternativa da instrução WRITE:

```
WRITE <nome-registo-output> FROM <nome-registo>
```

onde <nome-registo-output> é o nome de um registo associado a um ficheiro e <nome-registo> é uma estrutura de dados descrita na WORKING-STORAGE SECTION do programa.

O programa requer ainda uma declaração WRITE para cada registo diferente descrito, mas os registos são agora independentes e podem ser, se necessário, construídos em paralelo. Esta abordagem evita também outro problema, frequentemente encontrado em programas de principiantes, em que o registo de *output* para o ficheiro contém uma mistura de dois ou mais registos construídos no programa.

Os diferentes registos descritos na WORKING-STORAGE SECTION não têm o mesmo comprimento porque o efeito de WRITE ... FROM é o mesmo que:

```
MOVE <nome-registo> TO <nome-registo-output>
WRITE <nome-registo-output>
```

A declaração MOVE irá seguir as regras de um *move* de grupo, explicada na secção 4.2. Isto significa que, se WRITE ... FROM for utilizado correctamente, a área de *output* será preenchida com espaços para registos de tamanho inferior ao comprimento especificado na descrição de <nome-registo–output>.

A operação especial ESCREVER-LINHA-EM-BRANCO utiliza este preenchimento a espaços. O registo LINHA-EM-BRANCO é definido como se segue:

```
01      LINHA-EM-BRANCO      PIC X          VALUE SPACE.
```

e a operação associada é definida como:

```
ESCREVER-LINHA-EM-BRANCO.
        WRITE LINHA-OUT FROM LINHA-EM-BRANCO.
```

O efeito desta acção é movimentar um espaço em branco para LINHA-OUT e preencher então as restantes posições de LINHA-OUT com brancos, donde resulta uma linha em branco com o comprimento apropriado para *output*.

Quando um programa começa a execução, os valores de todos os campos são indefinidos, a menos que se lhe tenham atribuído valores através da cláusula VALUE. Assim, uma estrutura de registos que consista apenas em dados variáveis e em campos FILLER sem valores pode conter quaisquer caracteres quando o programa começa a execução. Para garantir que o registo começa com uma «vida nova», o procedimento de inicialização, ESTADO-INIC, deve incluir:

```
MOVE SPACES TO <nome-registo>
```

Isto irá garantir que todos os campos FILLER em <nome-registo> serão correctamente inicializados com espaços. Se existir mais do que um registo contendo apenas dados variáveis e campos FILLER, cada um deles deverá ser inicializado em ESTADO-INIC. Isto pode ser feito através da utilização de uma forma especial da declaração MOVE:

```
MOVE SPACES TO <nome-registo-1> <nome-registo-2>
        . . . . <nome-registo-n>
```

Tendo garantido que todos os registos são inicializados correctamente, é também necessário garantir que cada novo registo de *output* a ser construído comece por ser inicializado correctamente. A maneira mais simples de o fazer é reinicializar a área definida por <nome-registo> cada vez que um registo for escrito. Assim, a definição recomendada da declaração de escrita é:

```
ESCREVER-<nome-registo>.
        WRITE LINHA-OUT FROM <nome-registo>
        MOVE SPACES TO <nome-registo>.
```

Isto irá garantir que cada registo é construído independentemente do registo de *output* prévio.

Estruturas de registos, tais como cabeçalhos, contendo cláusulas VALUE para inicializar todo ou parte do registo não devem ser reinicializadas, caso contrário todos os valores desaparecerão. Assim eles não são referidos em ESTADO-INIC e operação de escrita destes tipos de registos será da forma:

```
ESCREVER -<registo constante>.
    WRITE LINHA-OUT FROM <registo-constante>.
```

8.5 — LINHAS E PÁGINAS

A discussão na secção 8.4 acerca da forma como escrever num ficheiro aplica-se a qualquer ficheiro de *output* de série. Geralmente, a execução de uma declaração WRITE faz que o registo seja escrito no ficheiro de *output* imediatamente após o último registo escrito. Assim, para fazer o *output* de linhas em branco para a linha da impressora, o método tem consistido em escrever registos completamente em branco no ficheiro de *output*.

A declaração Write permite uma série de opções específicas para o controle de espaceamento vertical numa linha de impressora. A utilização destas opções torna mais eficiente o processo de *output* e fornece também paginação que seria díficil escrevendo uma linha de cada vez.

Existem duas formas possiveis de controlo vertical com a declaração WRITE: o papel pode ser movido depois ou antes de a linha ser impressa.

A forma geral da primeira destas declarações WRITE é:

```
WRITE <nome-registo-output> FROM <nome-registo>
      AFTER ADVANCING <numero> LINES
```

onde <numero> pode ser ou um integral constante ou o nome de um campo contendo o valor integral. O número de linhas especificado deve ser positivo ou zero. Devem usar-se as zero linhas somente em circunstâncias especiais e recomenda-se vivamente que o principiante evite esta opção.

O efeito desta declaração é avançar o papel o número de linhas especificado e imprimir depois os componentes de <nome-registo>. A declaração:

```
WRITE <nome-registo-output> FROM <nome-registo>
      AFTER ADVANCING 1 LINES
```

irá ter o mesmo feito que uma declaração WRITE sem opção de controlo de linha — o registo será imprimido imediatamente abaixo do último. Uma única linha em branco entre o registo corrente e o registo precedente pode ser conseguida através de:

```
WRITE <nome-registo-output> FROM <nome-registo>
      AFTER ADVANCING 2 LINES
```

As palavras ADVANCING e LINES são opcionais e podem ser omitidas, caso se deseje.

A alternativa é imprimir um registo antes de se movimentar o papel; e, assim, eis a forma geral:

```
WRITE <nome-registo-output> FROM <nome-registo>
      BEFORE ADVANCING <numero> LINES
```

Exceptuando o momento em que o papel é movimentado, as regras são semelhantes à opção AFTER. Se <numero> tem o valor 1, esta situação é, de novo, equivalente àquela de linha corrente imediatamente a seguir à última.

Para se garantir que uma linha em branco se segue a registo corrente a ser imprimido, a declaração WRITE deve ser da forma:

```
WRITE <nome-registo-output> FROM <nome-registo>
      BEFORE ADVANCING 2 LINES
```

Para se começar a imprimir no início de uma nova página, pode ser utilizada a palavra reservada PAGE em vez de se especificar o número de linhas. Existem duas opções:

a) WRITE <nome-registo-output> FROM <nome-registo>
AFTER ADVANCING PAGE
b) WRITE <nome-registo-output> FROM <nome-registo>
BEFORE ADVANCING PAGE

Com o formato *a*), os conteúdos de <nome-registo> são imprimidos no início de uma nova página, enquanto o formato *b*) imprime o conteúdo de <nome-registo> e depois muda de página.

É mais seguro escolher uma das opções antes ou depois e utilizá-la consistentemente ao longo do programa. Isto irá evitar alguns dos desastres que frequentemente ocorrem quando se misturam as duas. Algumas implementações de COBOL obrigam o utilizador a escolher a especificação de que só será utilizada uma das opções para um dado ficheiro de *output*. Neste livro convencionou-se utilizar o fomato AFTER ADVANCING, quando necessário.

Suponha que se pretende um programa que começa o *output* no início de uma nova página com a escrita de um cabeçalho principal descrito em CABECALHO-PRINCIPAL, seguido de duas linhas em branco e duas linhas de cabeçalho chamados SUB-CABECALHO-1 e SUB-CABEÇALHO-2. A primeira linha de detalhe do mapa deve ser precedida de uma linha em branco. A estrutura seguinte deve conseguir o que se pretende.

```
IMPRIMIR-CABECALHOS.
        PERFORM  ESCREVER-CABECALHO-PRINCIPAL
        PERFORM  ESCREVER-SUB-CABECALHOS
        PERFORM  ESCREVER-LINHA-EM-BRANCO.

ESCREVER-CABECALHO-PRINCIPAL.
        WRITE  LINHA-OUT  FORM  CABECALHO-PRINCIPAL
                AFTER  ADVANCING  PAGE.

ESCREVER-SUB-CABECALHO.
        WRITE  LINHA-OUT  FROM  SUB-CABECALHO-1
                AFTER ADVANCING 3 LINES
        WRITE  LINHA-OUT  FROM  SUB-CABECALHO-2
                AFTER ADVANCING 1 LINES.

ESCREVER-LINHA-EM-BRANCO.
        WRITE  LINHA-OUT  FROM  LINHA-EM-BRANCO
                AFTER ADVANCING 1 LINES.
```

Apesar de a operação AFTER ADVANCING 1 LINES não ser estritamente necessária, é aconselhável utilizar AFTER ADVANCING em todas as declarações WRITE para um ficheiro se ela for utilizada em qualquer delas.

Existe a tentação de substituir a declaração WRITE para SUB-CABECALHO-2 por um WRITE ... BEFORE, mas isto irá fazer que os dois subcabeçalhos sejam escritos na mesma linha.

Tendo apreciado o salto para uma nova página, a próxima fase é considerar o modo de repetir um mapa muito extenso por várias páginas, cada uma com cabeçalho próprio. Suponha que um programa produz um mapa com um grande número de linhas de detalhe, sendo cada uma resultado de uma operação *standard* de escrita como se segue:

```
ESCREVER-LINHA-DETALHE.
        WRITE  LINHA-OUT  FROM  LINHA-DETALHE
                AFTER ADVANCING 1 LINES
        MOVE  SPACES  TO  LINHA-DETALHE.
```

O exemplo precedente demonstra como fazer o *output* de cabeçalhos no topo de uma página, no ínicio do *output*. Suponha agora que o número de linhas por página se restringe a 50 e que

se deve começar uma nova página após se escrever 50 linhas, com um conjunto completo de cabeçalhos.

A primeira acção a tomar é definir um contador para se contar o número de linhas já impressas. Esse contador poderia ser chamado LINHAS-IMP e a sua descrição adicionada ao VECTOR-ESTADO, já que ele traduz o estado do programa. Por exemplo:

```
01      VECTOR-ESTADO.
        .....
        05      LINHAS-IMP      PIC 99.
```

Dado que o número máximo de linhas a ser impresso é 50, dois dígitos serão suficientes para o contador. Sempre que se inicia uma nova página, LINHAS-IMP necessitará de ser colocado a zeros. Assim, IMPRIMIR-CABECALHOS poderia ser modificado como se segue:

```
IMPRIMIR-CABECALHOS.
PERFORM  ESCREVER-CABECALHO-PRINCIPAL
PERFORM  ESCREVER-SUB-CABECALHOS
PERFORM  ESCREVER-LINHA-EM-BRANCO
MOVE 6 TO LINHAS-IMP.
```

O contador LINHAS-IMP pode ser incrementado por cada operação escrever separada, mas, em suma, ele pode ser adaptado ao número de linhas imprimidas nos cabeçalhos. Isto é possível porque, neste caso, os cabeçalhos são sempre impressos em grupo. O valor inicial de LINHAS-IMP deve ser 0 em ESTADO-INIC.

É agora necessário modificar ESCREVER-LINHA-DETALHE para imprimir, sempre que necessário, os cabeçalhos para uma nova página e para incrementar de uma unidade o contador LINHAS-IMP por cada linha impressa. A versão modificada da operação de escrita poderia ser:

```
ESCREVER-LINHA-DETALHE.
        IF LINHAS-IMP = 50
                PERFORM  IMPRIMIR-CABECALHOS-PAG.
        WRITE  LINHA-OUT  FROM  LINHA-DETALHE
                AFTER ADVANCING 1 LINES
        MOVE  SPACES  TO  LINHA-DETALHE
        COMPUTE  LINHAS-IMP = LINHAS-IMP + 1.
```

Esta operação pode, obviamente ser codificada de modo que o registo de *output* seja impresso e o contador incrementado antes de se fazer o teste para uma nova página. A codificação mostrada em cima é melhor porque imprime apenas um novo cabeçalho de página se isto for requerido, isto é, quando existe um registo de *output* para imprimir. O método alternativo pode resultar em cabeçalhos sem linhas de detalhe para imprimir na última página — isto poderá fazer o utilizador pensar na hipótese de faltar alguma coisa ou de ter ocorrido um erro.

Existe um módulo especial em COBOL chamado Report-Writer que, tal como o nome sugere, fornece facilidades para criação de mapas. Esta é uma característica muito complexa e a sua análise não é possível no âmbito deste livro. No *ANS'74 standard,* o módulo Report-Writer é opcional e pode não ser implementado para sistemas menoros.

8.6 — REGISTOS DE «INPUT» DE TIPOS DIFERENTES

É possível ter um ficheiro que contenha tipos diferentes de *output;* por exemplo, um ficheiro de encomendas pode conter um registo de cabeçalho de encomendas contendo detalhes de

cliente seguido por um número de registos de detalhe contendo, cada um, detalhes de uma encomenda para um tipo de artigo.

A leitura de um ficheiro contendo dois ou mais tipos de registo pode constituir um grande problema, dado que só é possível decidir do tipo do próximo registo de um ficheiro através da sua leitura. Isto significa que será difícil desenhar ou projectar qualquer programa que dependa de uma ordenação especial ou associação de tipos de registo.

A descrição de um ficheiro contendo dois ou mais tipos de registos foi mencionada de um modo breve em 8.3, acima. Como exemplo, considere o ficheiro de uma companhia de finanças contendo pares de registos, o primeiro contendo informação pessoal do cliente e o segundo contendo dados financeiros confidenciais. Isto pode ser criado como um ficheiro chamado FICHEIRO-CLIENTE que contém dois tipos de registos chamados CLIENTE-PESSOAL e CLIENTE-FINANCA. Então, a descrição do ficheiro seria da seguinte forma:

```
FD      FICHEIRO-CLIENTE
        LABEL RECORDS OMITTED.
01      CLIENTE-PESSOAL
        .....
O1      CLIENTE-FINANCA
        .....
```

Existe apenas uma área de *input* associada ao ficheiro, e assim, as descricões do registo coexistem como se CLIENTE–FINANCA redefinisse a área ocupada por CLIENTE-PESSOAL. Assim, quando um registo é lido do ficheiro, os seus conteúdos são colocados numa área de *input* e requer-se então mum meio de se distinguirem os dois tipos de registo.

O «truque» *standard* é alocar um campo, que identifica o sítio do registo, na mesma posição fixa de cada registo. Testando o valor desse campo, pode decidir-se qual o tipo de registo que foi lido. Mais simples ainda é alocar o primeiro campo para definir o tipo de registo. Assim, no exemplo do FICHEIRO-CLIENTE, os registos podem ser definidos como se segue:

```
01      CLIENTE-PESSOAL.
        05      TIPO-REC-CLIENTE        PIC 9.
        .....
01      CLIENTE-FINANCA.
        05      FILLER                  PIC 9.
        .....
```

O segundo e subsequente tipo de registo não necessita de um campo com nome na posição definida porque todos os registos partilham a mesma área de *input*. Uma referência a TIPO-REC-CLIENTE aponta para a mesma posição em todos os registos independentemente do número de registos definidos.

Em muitos casos, esta técnica da utilização de campos partilhados é alargada para incluir um segundo campo comum para identificar os conteúdos de registos associados. No exemplo acima pode ser muito embaraçante se um registo de finanças de um cliente foi anexado aos detalhes pessoais de um outro. Assim, cada registo pode também conter um número de cliente numa posição comum. Por exemplo:

```
01      CLIENTE-PESSOAL.
        05      TIPO-REC-CLIENTE        PIC 9.
05      CLIENTE-NUMERO                  PIC 9(6).
        .....
01      CLIENTE-FINANCA.
        05      FILLER                  PIC 9.
        05      FILLER                  PIC 9(6).
        .....
```

Tendo verificado qual o tipo de registo lido, o programa deve usar então as descrições de registos apropriadas para aceder aos campos não comuns dos registos. Por exemplo:

```
01      CLIENTE-PESSOAL.
        05      TIPO-REC-CLIENTE            PIC 9.
        05      CLIENTE-NUMERO              PIC 9(6).
        05      CLIENTE-P-NOME              PIC X(30).
        .....
01      CLIENTE-FINANCA.
        05      FILLER                      PIC 9.
        05      FILLER                      PIC 9(6).
        05      CLIENTE-F-SALARIO           PIC 9(5)V99.
        05      CLIENTE-F-EMPRESTIMO        PIC 9(4).
        .....
```

Suponha que CLIENTE-PESSOAL é identificado pelo tipo 1 e CLIENTE-FINANCA pelo tipo 2. Se TIPO-REC-CLIENTE for igual a 1, todas as referências subsequentes se devem relacionar com campos definidos em CLIENTE-PESSOAL, por exemplo CLIENTE-P-NOME. Como alterativa, se TIPO-REC-CLIENTE for igual a 2, as referências devem relacionar-se com campos que são comuns aos dois tipos de registo ou em CLIENTE-FINANCA, por exemplo CLIENTE-F-SALARIO e CLIENTE-F-EMPRESTIMO.

Repare que uma tentativa para tratar um registo de tipo 1 como um registo de tipo 2, ou vice-versa, resultará em desastre, porque os tipos de registo são diferentes.

Os registos diferentes pertencentes a um mesmo ficheiro podem não ser todos do mesmo comprimento. O tamanho da área de *input* para o ficheiro será escolhida para receber o maior registo definido para o ficheiro. De que modo os dados são na realidade representados no ficheiro, quando os comprimentos dos registos variam, irá estar sujeito à implementação de COBOL. Na maioria dos sistemas, o modo mais simples de representar os campos é «preencher» todos os registos formatando-os com o mesmo comprimento.

A utilização de tipos diferentes de registos de *input* num ficheiro de *input* irá afectar a estrutura do programa. Até agora, os exemplos e exercícios têm lidado com um tipo único de registo e têm-se baseado na estrutura de desenho:

```
ler-proximo-<nome-ficheiro-input>
until  fim-de-<nome-ficheiro-input>  do
        <processar-registo>
        ler-proximo-<nome-ficheiro-input>
enduntil
```

Continuando com o exemplo sublinhado em cima, suponha que se requer um programa para processar tipos de registos aos pares, misturando os dados de ambos para criar um *output* não especificado. Uma primeira aproximação para o desenho de programa pode ser:

```
ler-proximo-ficheiro-cliente
until  fim-de-ficheiro-cliente  do
        if tipo-rec-cliente = 1 then
                processar-pessoal
        else
                processar-finança
        endif
        ler-proximo-ficheiro-cliente
        enduntil
```

No refinamento «processar-pessoal» seria definido o processamento apropriado para um registo de tipo 1 <CLIENTE–PESSOAL>. Isto provavelmente incluiria o armazenamento de alguns

dados deste registo para serem combinados com dados do registo semelhante do tipo 2 <CLIENTE-FINANCA>. No refinamento «processar-finança» seria definido o processamento para um registo do tipo 2.

Existe um certo número de problemas com esta estrutura:

a) Se o campo 'tipo-rec-cliente' contiver um valor inválido, então o registo será tratado como sendo do tipo 2;

b) O programa não testa se um registo do tipo 2 foi ou não precedido por um do tipo 1;

c) O programa deve garantir que um registo do tipo 2 pertence ao tipo 1 precedente, caso este exista.

A solução para o problema *a*) é perfeitamente linear — incluir um teste para o tipo de registo 2 em vez de convencionar que um registo que não é de tipo 1 é obrigatoriamente de tipo 2. Assim, a estrutura de programa seria:

```
ler-proximo-ficheiro-cliente
until fim-de-ficheiro-cliente do
        if tipo-rec-cliente = 1 then
                processar-pessoal
        else
                if tipo-rec-cliente = 2 then
                        processar-finança
                else
                        tipo-rec-erro
                endif
        endif
        ler-proximo-ficheiro-cliente
enduntil
```

O refinamento «tipo-rec-erro» deve especificar a acção a ser tomada se houver ocorrência de tipo inválido no *input*. Isto poderia ocasionar a impressão do registo em questão e uma consequente mensagem de erro.

A solução para o problema *b*) é gravar o estado corrente do *input*. O que é essencial na informação é saber se um registo de tipo 1 que está a ser retido é de tipo 2 ou não. Assim, pode definir-se um campo booliano «tipo-1-retido», que é verdadeiro se um registo de tipo 1 foi encontrado e não foi ainda combinado com o de tipo 2 e falso no caso contrário. Este campo necessitará de ser inicializado para falso no início do programa e para verdadeiro sempre que um registo do tipo 1 é encontrado e redefinido para falso sempre que um registo de tipo 2 é processado com êxito. Existem dois casos especiais quando faltar um registo de tipo 2: se o registo de tipo 1 é lido e «tipo-1-retido» é já verdadeiro ou, no fim do programa, se «tipo-1-retido» é verdadeiro.

A incorporação deste campo booliano conduz-nos à seguinte estrutura de programa:

```
mover falso para tipo-1-retido
ler-proximo-ficheiro-cliente
until fim-de-ficheiro-cliente do
        if tipo-rec-cliente = 1 then
                processar-pessoal
        else
                if tipo-rec-cliente = 2 then
                        processar-finança
                else
                        tipo-rec-erro
                endif
        endif
```

```
        ler-proximo-ficheiro-cliente
enduntil
if tipo-1-retido then
        num-erro-tipo-2
endif

processar-pessoal
        if tipo-1-retido then
                num-erro-tipo-2
        endif
        mover verdadeiro para tipo-1-retido
        .....

processar-finança
        mover falso para tipo-1-retido
        .....
```

Isto deixa ainda o problema *c*) por resolver — como pode o programa garantir que apenas se processe o registo de tipo 2 que pertence ao registo de tipo 1 retido? A solução é a seguinte: sempre que um registo de tipo 1 é «retido», o valor do «cliente-numero» associado deve também ser armazenado. (Seja como for, isto pode ser feito como parte de «processar pessoal».) Sempre que um registo de tipo 2 é encontrado, o seu «cliente-numero» deve ser verificado para se ver se condiz com o cliente-número do registo de tipo 1 retido. Se os cliente-número condizem, o processamento pode continuar, de outro modo, existe um erro e o registo de tipo 2 deve ser rejeitado. Os refinamentos modificados para «processar-pessoal» e «processar-finança» são apresentados em baixo.

```
processar-pessoal
        if tipo-1-retido then
                num-erro-tipo-2
        endif
        mover verdadeiro para tipo-1-retido
        mover cliente-numero para cliente-num-retido
        .....

processar-finança
        if cliente-numero = cliente-num-retido then
        mover falso para tipo-1-retido
        lidar-tipo-2
        else
                num-erro-tipo-1
        endif
```

O refinamento «lidar-tipo-2» encontra-se no local onde será definido o processamento de um registo de tipo 2 combinado correctamente. O refinamento «num-erro-tipo-1» irá especificar a acção a ser tomada quando um registo de tipo 2 se segue a um registo de tipo 1, mas os cliente-número não condizem.

Repare que este desenho é «conclusivo» pelos registos de tipo 1 — tem de ser encontrado um registo de tipo 1 seguido pelo registo de tipo 2 correspondente. Não é feita nenhuma tentativa para reter um registo de tipo 2 na esperança de que apareça um registo de tipo 1 correspondente.

Este exemplo não terá mais nenhum desenvolvimento, mas o leitor poderia continuá-lo como se fosse um exercício. O exemplo foi incluído precisamente para dar ao leitor algumas perspectivas quanto às dificuldades de desenhos de programas que utilizam ficheiros de *input* com ti-

pos de registos diferentes. Demonstra também quanta atenção necessita de ser dispensada na solução de um problema aparentemente simples.

O exemplo acima demonstra também que a estrutura básica alternativa do programa poderia ser talvez melhor para este tipo de problema. O modelo geralmente usado para este tipo de programa é apresentado no estudo de caso da próxima secção.

8.7 — ESTUDO DE CASO

Este estudo de caso explora um problema relacionado com o processamento de um ficheiro que contém diferentes tipos de registos de *input*. A estrutura geral do registo de *input* é que os dados se agrupam em grupos de registos ou «porções», relacionando-se cada porção com um cliente. Este agrupamento de registos aparece muito frequentemente em aplicações de processamento de dados comerciais. A especificação do problema é apresentada em baixo.

O *input* de um programa que produz declarações mensais para clientes da XYZ Vendedores de Livros consiste em três tipos de registos:

Registo	Conta	
	Tipo de registo	1
	Número de conta	2 letras e 4 dígitos
	Nome do cliente	25 caracteres
	Balanço posterior	5 dígitos, pénis com sinal
Registo	Vendas	
	Tipo de registo	2
	Número de conta	como acima
	Data da transacção	8 caracteres (DD/MM/AA)
	Número da factura	1 letra e 4 dígitos
	Preço cobrado	4 dígitos, pénis
Registo	Pagamentos	
	Tipo de registo	3
	Número de conta	como acima
	Data do pagamento	igual ao registo de vendas
	Código de receita	1 letra e 3 dígitos
	Quantia recebida	5 dígitos, pénis

Uma «porção» de *input* para uma conta particular consiste num registo de conta seguido por uma mistura de registo de vendas e de pagamento. Se não existirem transacções do mês anterior, irá aparecer um registo de tipo 1 sem quaisquer registos associados de tipo 2 ou 3.

Faz-se o *input* dos registos em porções, com os registos de tipo 1 precedendo os registos de tipo 2 e 3, caso haja algum. Os registos de tipo 2 e 3 para uma conta são gravados na ordem por que aparecem, de modo que possam aparecer na declaração por ordem de data. Os dados foram já processados num simples programa de validação de dados que verificou se os campos têm todos valores válidos. O programa de validação de dados não garante que não haja registos de tipo 2 ou 3 sem registos de tipo 1 correspondentes.

Um exemplo de uma porção de *input* pode ser:

```
1 AA0012 JOE BLOGGS                    00695
3 AA0012 03/07/82 C016 00695
2 AA0012 08/07/82 G0003 1525
2 AA0012 17/07/82 T0010 2695
3 AA0012 25/07/82 C345 01525
```

Repare que os campos foram separados por espaços para que os dados ficassem mais legíveis. Os valores apareceriam, normalmente, contiguamente no registo de *input* final, a menos que tivessem sido definidas áreas em branco na especificação.

O programa pretendido deve produzir uma declaração para cada conta. Cada declaração deve aparecer numa página separada com cabeçalhos adequados seguidos por: o balanço do transporte anterior, detalhes das transacções e o balanco do transporte a fazer. O programa deve rejeitar quaisquer registos de tipo 2 ou 3 que não pertençam ao grupo corrente. Não é necessário verificar se os registos estão, de facto, na ordem por datas. O *output* correspondente ao grupo amostra de dados apresentado acima deve ser:

CONTA: AA0012 NOME: JOE BLOGGS

DATA	REF NUM.	CREDITO	DEBITO	BALANCO
		TRANSPORTE		£6.95
03/07/82	C016	£L6.95		£0.00
08/07/82	G0003		£15.25	£15.25
17/07/82	T0010		£26.95	£42.20
25/07/82	C345	£15.25		£26.95
		QUANTIA CR.		£26.95

Este problema foi deliberadamente simplificado ao excluir a actualização do balanço no registo de conta. O problema geral da actualização de ficheiros sequenciais será considerado no próximo capítulo.

Para começar a resolver este problema, é, provavelmente, melhor atribuir nomes a todos os campos de *input* para evitar qualquer confusão que possa surgir da utilização de campos comuns em registos diferentes. A seguir sugere-se um esquema de atribuição de nomes:

```
reg-conta
        conta-reg-tipo      tipo de registo, comum a todos os registos
        conta-numero        número de conta do cliente, comum
        conta-nome          nome do cliente
        conta-trans         balanço da conta a transportar

reg-vendas
        data-venda          data da transacção de venda
        num-factura         número da factura para a venda
        preço-factura       valor da factura de venda

reg-pagamento
        data-pag            data do pagamento
        cod-rec-paga        código do recibo do pagamento
        val-paga            valor do pagamento
```

Para a atribuição de nomes convencionou-se que campos de dados comuns têm como prefixo somente o nome-ficheiro e uma letra para indicar o tipo de registo. Na descrição do segundo e terceiro registo irão existir *fillers* para preenchimento de espaços, o que não irá afectar o desenho de programa.

Seria possível ter apenas dois tipos de registo, combinando os registos de vendas e de pagamentos, usando a redefinição dentro de um registo comum. No entanto, estes registos são definidos separadamente, para realçar as suas diferenças e para melhorar a clareza do programa.

Este problema já não se adapta ao padrão dos primeiros problemas estudados neste livro. Os registos já não são independentes uns dos outros, e, assim, requer-se uma estrutura diferente

para o programa. A principal unidade de dados que conduz o programa é uma série de registos relacionados com uma conta. O nível superior do desenho de programa poderia ser:

```
declarações-print
        encontrar-tipo-1
        until fim-de-fich-conta do
                processar-lote
        enduntil
```

O programa necessita de ignorar quaisquer registos precedendo o primeiro registo de tipo 1 (conta), dado que estes são, obviamente, inválidos. Encontrado o primeiro registo de tipo 1, o programa deve processar os dados lote a lote. Repare que não se imprimem cabeçalhos no início do programa — os cabeçalhos estão associados a cada lote de *input*.

O refinamento mais simples de «encontrar-tipo-1» é:

```
encontrar-tipo-1
        ler-proximo-fich-conta
        until tipo-reg-conta = 1 do
                erro-tipo-1-esperado
                ler-proximo-fich-conta
        enduntil
```

Se se pressupõe que o programa deverá funcionar com todos os tipos de dados de *input*, este refinamento não é suficientemente bom. O refinamento apresentado acima não irá funcionar se não existirem no ficheiro registos de tipo 1 no *input*. O ciclo until tipo-reg-conta = 1 nunca irá terminar e o programa continuará a executar para além do fim do ficheiro <ciclo infinito>. Para evitar esta situação, sugere-se a seguinte estrutura:

```
encontrar-tipo-1
        mover falso to fim-pesquisa
        until-fim-pesquisa do
                ler-proximo-fich-conta
                if fim-de-fich-conta then
                        mover verdadeiro to fim-pesquisa
                else
                        if tipo-reg-conta = 1 then
                                mover verdadeiro to fim-pesquisa
                        else
                                erro-tipo-1-esperado
                        endif
                endif
        enduntil
```

A saída do ciclo deve ocorrer quando é encontrado um registo de tipo 1 ou o fim do ficheiro. O ciclo:

```
until fim-de-fich-conta or tipo-reg-conta = 1 do
```

poderia ser considerado, mas a segunda condição não é testada se a primeira for verdadeira (ver secção 6.5). Assim, utiliza-se um campo booliano, «fim-pesquisa», o qual é inicalizado como falso e se torna verdadeiro quando ocorre uma das condições de terminação.

Para completar esta parte do programa, pode especificar–se o refinamento de «erro-tipo-1-esperado» neste ponto. Para evitar a mistura de mensagens sobre erros de *input* com a impressão de declarações, é aconselhável fazer o *output* destas mensagens de erro para um ficheiro diferente, ou num dispositivo diferente do das declarações. O uso de um ficheiro de *output* alter-

nativo é analisado no capítulo 9. Para os objectivos deste estudo de caso, a declaração *display* introduzida na secção 7.2 pode ser utilizada para identificar registos de erro. Assim, o refinamento poderia ser:

```
erro-tipo-1-esperado
        display «registo de tipo 1 esperado — encontrado».
                conta-reg-tipo, «  », conta-numero
```

A declaração *display* destina-se apenas a *output* «baixo volume» e, neste caso, visto que os dados foram parcialmente validados, o número de erros deve ser baixo.

O refinamento de «processar-lote» requer também um ciclo com duas condições de paragem. O fim de um lote de tipo 1 é normalmente marcado pelo aparecimento de um registo de tipo 1, mas o último lote termina com a condição fim-de-ficheiro. Mais uma vez deve ser utilizado um campo booliano para controlar o ciclo, e um possível refinamento é:

```
processar-lote
        imprimir-cabeçalhos
        guardar-detalhes-conta
        ler-proximo-fich-conta
        activar-lote-completo
        until lote-completo do
                processar-registo-det
                ler-proximo-fich-conta
                activar-lote-completo
        enduntil
        imprimir-qt-cred
```

No início de cada lote é necessário escrever os cabeçalhos respectivos. É também necessário armazenar os detalhes necessários do registo de tipo 1, pois a área de *input* será apagada com os conteúdos do registo seguinte lido do ficheiro. O processamento do lote corrente continua até que o campo booliano «lote-completo» se torne verdadeiro. No fim do lote, a linha final é imprimida por «imprimir-qt-cred».

O refinamento «activar-lote-completo» tem de fornecer um valor ao campo booliano lote-completo de acordo com o corrente estado do ficheiro de *input*. O refinamento poderia ser:

```
activar-lote-completo
        if fim-de-fich-conta then
                mover verdadeiro to lote-completo
        else
                if tipo-reg-conta = 1 then
                        mover verdadeiro to lote-completo
        else
                mover falso to lote-completo
        endif
```

Uma abordagem alternativa, que conduz a uma estrutura de programa mais nítida, é convencionar, simplesmente, que o lote está incompleto à entrada do ciclo until. O ciclo começa então com a tentativa de leitura do próximo registo e lida com a situação corrente colocando «lote-completo» com o valor verdadeiro ou processando um registo de detalhe. O refinamento de «processar-lote» é então:

```
processar-lote
        imprimir-cabeçalhos
        guardar-detalhes-conta
        mover falso to lote-completo
```

```
until lote-completo do
        ler-proximo-fich-conta
        tratar-registo-corrente
enduntil
imprimir-qt-cred
```

A primeira acção a ser tomada é definir o refinamento «imprimir-cabeçalhos», e poderia ser:

```
imprimir-cabeçalhos
        escrever-cabeçalho-pag (após salto de pagina)
        imprimir-cabeçalho-conta
        escrever-cabeçalhos-det (após saltar uma linha)
        imprimir-balanço-trans
```

Os comentários entre parênteses não são necessários para o desenho de programa, mas recordarão o que se requer ao traduzir-se o desenho para COBOL.

Continuando o desenvolvimento desta parte do programa, é necessário definir campos para guardar o número de conta e o nome do cliente no cabeçalho de conta. Por exemplo:

```
cabeçalho-conta
        num-conta-cab       número de conta
        nome-conta-cab      nome do cliente
```

O refinamento de «imprimir-cabeçalho-conta» poderia então ser:

```
imprimir-cabeçalho-conta
        mover conta-numero para num-conta-cab
        mover conta-nome para nome conta-cab
        escrever-cabeçalho-conta (após saltar duas linhas)
```

Do mesmo modo, «imprimir-balanço-trans» pode ser refinado após ter sido definido um registo de *output* adequado. Olhando bem, podemos ver que linhas para o transporte a fazer e para a quantia devida são semelhantes em conteúdo e utilização, e, assim poder-se-ia definir um registo de *output* comum:

```
linha-balanço
        balanço-texto     texto apropriado para a linha
        balanço-qt        balanço a ser imprimido
```

O refinamento então será:

```
imprimir-balanço-trans
        mover «transporte» para balanço-texto
        mover conta-trans para balanço-qt
        escrever-linha-balanço (após saltar uma linha)
```

Voltando agora a «processar-lote», o próximo refinamento requerido é «guardar-detalhes-conta», que necessita de copiar para campos temporários alguns dados do registo de tipo 1 que sejam necessários durante o processamento do lote. Existem dois campos nessas condições, «conta-numero» e «conta-trans», que necessitam de ser copiados para campos de dados correspondentes, por exemplo:

```
campos-temporarios
        num-conta-corrente      número de conta para o lote
        balanço-corrente        balanço para ser actualizado
```

```
guardar-detalhes-conta
        mover-numero para num-conta-corrente
        mover conta-trans para balanço-corrente
```

O refinamento que a seguir se requer é «tratar-registo–corrente», que é na realidade o coração do programa. Este refinamento necessita de confirmar se o lote está completo, e, no caso de não estar, dever processar um registo de detalhe para o lote. O próximo nível de refinamento poderia ser:

```
tratar-registo-corrente
        if fim-de-fich-conta then
                mover verdadeiro para lote-completo
        else
                if tipo-reg-conta = 1 then
                        mover verdadeiro to lote-completo
                else
                        processar-reg-det
                endif
        endif
```

Para processar um registo de detalhe, o programa precisa de verificar se o registo pertence à situação corrente e, no caso afirmativo, processar ou uma venda ou um pagamento. Assim, o refinamento pode ser:

```
processar-reg-det
        if conta-numero not = num-conta-corrente then
                erro-num-conta-invalido
        else
                if tipo-reg-conta = 2 then
                        processar-venda
                else
                        processar-recibo
                endif
        endif
```

O programa de validação de dados deve ter garantido que apenas registos de tipo 2 ou 3 pertencem a este programa, e, assim, não deve ser requerido um teste para o tipo 3. Os números de conta inválidos podem ser tratados através do uso de declarações *display* de modo semelhante ao de rotinas de erro prévias:

```
erro-num-conta-invalido
        display «numero de conta invalido»
        display «encontrado:», conta-numero, «no lote para:»,
                num-conta-corrente
```

Para refinar «processar-venda» é necessário decidir quais os nomes para os campos de dados de *output* na linha de detalhe impressa para cada transacção. Podem definir-se registos separados para vendas e para pagamentos, mas as duas linhas são suficientemente semelhantes para se utilizar um registo de *output* comum. Por exemplo:

```
linha-trans
        trans-data              data da transacção
        trans-num-factura  ==  trans-cod-reg
        trans-credito           valor do crédito, se existir
        trans-debito            valor do débito, se existir
        trans-balanço           balanço actualizado
```

onde:

trans-num-factura	número da factura para a venda
trans-cod-reg	código do recibo para pagamento

O formato dos números de factura e dos códigos de recibo diferem apenas ligeiramente, e, por isso, estes campos podem partilhar a mesma área utilizando a redefinação.

Os refinamentos de «processar-venda» e «processar-recibo» são agora bastante lineares:

```
processar-venda
        calcular balanço-corrente = balanço-corrente + preço-factura
        mover data-venda para trans-data
        mover num-factura para trans-num-factura
        mover preço-facutura para trans-debito
        mover balanço-corrente para trans-balanço
        escrever-linha-trans

processar-recibo
        calcular balanço-corrente = balanço-corrente — val-paga
        mover data-paga para trans-data
        mover cod-rec-paga para trans-cod-reg
        mover val-paga para trans-credito
        mover balanço-corrente para trans-balanço
        escrever-linha-trans
```

O refinamento final necessário para completar o desenho de programa é «imprimir-qt-cred», que produz a linha no fim da declaração. O registo de *output* correspondente foi definido aquando da consideração dos cabeçalhos, e, assim, o refinamento é:

```
imprimir-qt-cred
        mover «quantidade deduzida» para balanço-texto
        mover balanço-corrente para balanço-qt
        escrever-linha-balanço (após saltar uma linha)
```

O desenho de programas completo pode agora ser criado, incluindo as estruturas de registo. Os registos de *input* são:

reg-conta

conta-reg-tipo	tipo de registo, comum a todos os registos
conta-numero	número de conta do cliente, comum
conta-nome	nome do cliente
conta-trans	balanço da conta a transportar

reg-vendas

data-venda	data da transacção de venda
num-factura	número da factura para a venda
preço-factura	valor da factura de venda

reg-pagamento

data-paga	data do pagamento
cod-rec-paga	código do recibo do pagamento
val-paga	valor do pagamento

Os registos de *output* são:

```
cabeçalho-conta
        num-conta-cab           número de conta
        nome-conta-cab          nome do cliente

linha-balanço
        balanço-texto           texto apropriado para a linha
        balanço-qt              balanço a ser imprimido
linha-trans
        trans-data              data da transacção
        trans-num-factura == trans-cod-reg
        trans-credito           valor do crédito, se existir
        trans-debito            valor do débito, se existir
        trans-balanço           balanço actualizado

onde:
        trans-num-factura       número da factura para a venda
        trans-cod-reg           código do recibo para pagamento
```

Em adição, serão requeridas descrições para as linhas constantes:

«cabeçalhos-pag» e «cabeçalhos-det».

Os campos temporários necessários para o programa são:

```
campos-temporarios
        num-conta-corrente      número de conta para o lote
        balanço-corrente        balanço para ser actualizado
```

Em adição a estes, podem ser adicionados ao vector-estado dois campos boolianos «fim-pesquisa» e «lote-completo» se estes forem utilizados.
O desenho completo do programa é então:

```
declarações-print
        encontrar-tipo1
        until fim-de-fich-conta do
                processo-lote
        enduntil

encontrar-tipo-1
        mover falso para fim-pesquisa
        until fim-pesquisa do
                ler-proximo-fich-conta
                if fim-de-fich-conta then
                        mover verdadeiro para fim-pesquisa
                else
                        if tipo-reg-conta = 1 then
                                mover verdadeiro para fim-pesquisa
                        else
                                erro-tipo-1-esperado
                        endif
                endif
        enduntil
```

```
erro-tipo-1-esperado
        display «registo de tipo 1 esperado — encontrado»,
                conta-reg-tipo, « », conta-número

processar-lote
        imprimir-cabeçalhos
        guardar-detalhes-conta
        mover falso para lote-completo
        until lote-completo do
                ler-proximo-fich-conta
                tratar-registo-corrente
        enduntil
        imprimir-qt-cred

imprimir-cabeçalhos
        escrever-cabeçalho-pag (após salto de página)
        imprimir-cabeçalho-conta
        escrever-cabeçalhos-det (após saltar uma linha)
        imprimir-balanço-trans

imprimir-cabeçalho-conta
        mover conta-numero para num-conta-cab
        mover conta-nome para nome-conta-cab
        escrever-cabeçalho-conta (após-saltar duas linhas)

imprimir-balanço-trans
        mover «transporte» para balanço-texto
        mover conta-trans para balanço-qt
        escrever-linha-balança (apos saltar uma linha)

guardar-detalhes-conta
        mover conta-numero para num-conta-corrente
        mover conta-trans para balanço-corrente

tratar-registo-corrente
        if fim-de-fich-conta then
                mover verdadeiro para lote-completo
        else
                if tipo-reg-conta = 1 then
                        mover verdadeiro para lote-completo
                else
                        processar-reg-det
                endif
        endif

processar-reg-det
        if conta-numero not = num-conta-corrente then
                erro-num-conta-invalido
        else
                if tipo-reg-conta = 2 then
                        processar-venda
                else
                        processar-recibo
                endif
        endif
```

```
erro-num-conta-invalido
        display «numero de conta invalido»
        display «encontrado:», conta-numero, «no lote para:»,
                num-conta-corrente

processar-venda
        calcular balanço-corrente = balanço-corrente + preço-factura
        mover data-venda para trans-data
        mover num-factura para trans-num-factura
        mover preço-factura para trans-debito
        mover balanço-corrente para trans-balanço
        escrever-linha-trans

processar-recibo
        calcular balanço-corrente = balanço-corrente — val-paga
        mover data-paga para trans-data
        mover cod-rec-paga para trans-cod-reg
        mover val-paga para trans-credito
        mover balanço-corrente para trans-balanço
        escrever-linha-trans

imprimir-qt-cred
        mover «quantia deduzida» para balanço-texto
        mover balanço-corrente para balanço-qt
        escrever-linha-balanço (após saltar uma linha)
```

O programa completo de COBOL correspondente a este desenho é apresentado a seguir.

```
IDENTIFICATION  DIVISION.
PROGRAM-ID.  CNTMVT.

ENVIRONMENT  DIVISION.
CONFIGURATIN  SECTION.
SOURCE-COMPUTER.  IBM-S34.
OBJECT-COMPUTER.  IBM-S34
*       <nome-computador> deve ser substituído pelo nome
*       do computador usado para correr os programas.

SPECIAL-NAMES.
        REQUESTOR  IS  WORKSTN.
INPUT-OUTPUT  SECTION.
FILE-CONTROL.
        SELECT  FICHCNT  ASSIGN  TO  DISK-FICHCNT.
        SELECT  FICHOUT  ASSIGN  TO  PRINTER-FICHOUT.
*       <sistema-imput> e <sistema-output> devem estar conformes
*       com as regras do sistema utilizado.

DATA  DIVISION.
FILE  SECTION.
FD      FICHCNT
        LABEL  RECORDS  ARE  STANDARD.
01      REG-CONTA
        05      CONTA-REG-TIPO                  PIC 9.
        05      CONTA-NUMERO
```

```
                10      NUM-CONTA-1     PIC AA.
                10      NUM-CONTA-2     PIC 9999.
        05      CONTA-NOME              PIC X(25).
        05      CONTA-TRANS             PIC S999V99.

01  REG-VENDAS.
        05      FILLER                  PIC 9.
        05      FILLER                  PIC X(6).
        05      DATA-VENDA              PIC X(8).
        05      NUM-FACTURA.
                10      NUM-FACT-1      PIC A.
                10      NUM-FACT-2      PIC 9999.
        05      PRECO-FACTURA           PIC 99V99.

01  REG-PAGAMENTO.
        05      FILLER                  PIC 9.
        05      FILLER                  PIC X(6).
        05      DATA-PAGA               PIC X(8).
        05      COD-REG-PAG.
                10      COD-REG-1       PIC A.
                10      COD-REG-2       PIC 999.
        05      VAL-PAGA                PIC 999V99.
FD  FICHOUT
                LABEL RECORDS OMITTED.
01  LINHA-OUT                          PIC X(120).

WORKING-STORAGE SECTION.
01  VECTOR-ESTADO.
        05      FIM-FICH-CONTA          PIC X.
                88      FIM-DE-FICHEIRO-CONTA       VALUE «E».
        05      ESTADO-PESQUISA         PIC X.
                88      FIM-PESQUISA                VALUE «V».
        05      ESTADO-LOTE             PIC X.
                88 LOTE-COMPLETO                    VALUE «V».
01  CAMPOS-TEMPORARIOS.
        05      NUM-CONTA-CORRENTE      PIC X(6).
        05      BALANCO-CORRENTE        PIC S999V99.

01  CABECALHOS-PAG.
        05      FILLER                  PIC X(19)   VALUE SPACES.
        05      FILLER                  PIC X(15)   VALUE«XYZ BOOKSELLERS»

01  CABECALHO-CONTA.
        05      FILLER                  PIC X(9)    VALUE «CONTA:».
        05      NUM-CONTA-CAB           PIC X(6).
        05      FILLER                  PIC X(7)    VALUE SPACES.
        05      FILLER                  PIC X(6)    VALUE «NOME:».
        05      NOME-CONTA-CAB          PIC X(25).

01  CABECALHOS-DET.
        05      FILLER                  PIC X(8)    VALUE «DATA».
        05      FILLER                  PIC X(3)    VALUE SPACES.
        05      FILLER                  PIC X(13)   VALUE «REFERENCIA».
        05      FILLER                  PIC X(3)    VALUE SPACES.
        05      FILLER                  PIC X(7)    VALUE «CREDITO».
```

```
           05      FILLER                  PIC X(3)        VALUE SPACES.
           05      FILLER                  PIC X(6)        VALUE «DEBITO».
           05      FILLER                  PIC X(3)        VALUE SPACES.
           05      FILLER                  PIC X(7)        VALUE «BALANCO».

01         LINHA-TRANS.
           05      TRANS-DATA              PIC X(8).
           05      FILLER                  PIC X(7).
           05      TRANS-NUM-FACTURA       PIC X(5).
           05      TRANS-COD-REG REDEFINES TRANS-NUM-FACTURA.
                   10   TRANS-REG-COD      PIC X(4).
                   10   FILLER             PIC X.
           05      FILLER                  PIC X(7).
           05      TRANS-CREDITO           PIC £££9.99
           05      FILLER                  PIC XXX.
           05      TRANS-DEBITO            PIC ££9.99
           05      FILLER                  PIC XXX.
           05      TRANS-BALANCO           PIC £££9.99CR.

01         LINHA-BALANCO.
           05      FILLER                  PIC X(12).
           05      BALANCO-TEXTO           PIC X(11).
           05      FILLER                  PIC X(23).
           05      BALANCO-QT              PIC £££9.99CR.

PROCEDURE DIVISION.
PROGRAMA-PRINCIPAL.
           PERFORM ESTADO-INC
           PERFORM DECLARACÕES-PRINT
           PERFORM FECHO.
ESTADO-INIC.
           MOVE SPACE TO FIM-FICH-CONTA
           OPEN INPUT FICHCNT OUTPUT FICHOUT
           MOVE SPACES TO LINHA-BALANCO LINHA-TRANS.
FECHO.
           CLOSE FICHCNT FICHOUT
           STOP RUN.
LER-PROXIMO-FICH-CONTA.
           READ FICHCNT
                   AT END MOVE «E» TO FIM-FICH-CONTA.
ESCREVER-CABECALHOS-PAG.
           WRITE LINHA-OUT FROM CABECALHOS-PAG
           AFTER ADVANCING PAGE.
ESCREVER-CABECALHO-CONTA.
           WRITE LINHA-OUT FROM CABECALHO-CONTA
                   AFTER AVANCING 3 LINES
ESCREVER-CABECALHOS-DET.
           WRITE LINHA-OUT FROM CABECALHOS-DET
                   AFTER ADVANCING 2 LINES.
ESCREVER-LINHA-BALANCO.
           WRITE LINHA-OUT FROM LINHA-BALANCO
                   AFTER ADVANCING 2 LINES.
           MOVE SPACES TO LINHA-BALANCO.
```

```
ESCREVER-LINHA-TRANS.
        WRITE LINHA-OUT FROM LINHA-TRANS
                AFTER ADVANCING 2 LINES.
        MOVE SPACES TO LINHA-TRANS.

DECLARACÕES-PRINT.
        PERFORM ENCONTRAR-TIPO-1
        PERFORM PROCESSAR-LOTE UNTIL FIM-DE-FICHEIRO-CONTA.
ENCONTRAR-TIPO-1.
        MOVE «F» TO ESTADO-PESQUISA
        PERFORM PROCURA-1 UNTIL FIM-PESQUISA.
PROCURA-1.
        PERFORM LER-PRÓXIMO-FICH-CONTA
        IF FIM-DE-FICHEIRO-CONTA
                MOVE «V» TO ESTADO-PESQUISA
        ELSE
                IF CONTA-REG-TIPO =1
                        MOVE «V» TO ESTADO-PESQUISA
                ELSE
                        PERFORM ERRO-TIPO-1-ESPERADO.
ERRO-TIPO-1-ESPERADO.
        DISPLAY «REGISTO TIPO 1 ESPERADO - ENCONTRADO»,
                «CONTA-REG-TIPO,» «CONTA-NUMERO» UPON WORKSTN.
PROCESSAR-LOTE.
        PERFORM  PRINT-CABECALHOS
        PERFORM GUARDAR-DETALHES-CONTA
        MOVE «F» TO ESTADO-LOTE.
        PERFORM CICLO-DET UNTIL LOTE-COMPLETO
        PERFORM IMPRIMIR-QT-CRED.
CICLO-DET.
        PERFORM LER-PROXIMO-FICH-CONTA
        PERFORM TRATAR-REGISTO-CORRENTE.
PRINT-CABECALHOS.
        PERFORM ESCREVER-CABECALHOS-PAG
        PERFORM IMPRIMIR-CABECALHO-CONTA
        PERFORM ESCREVER-CABECALHOS-DET
        PERFORM IMPRIMIR-BALANCO-TRANS.
IMPRIMIR-CABECALHO-CONTA.
        MOVE CONTA-NUMERO TO NUM-CONTA-CAB
        MOVE CONTA-NOME TO NOME-CONTA-CAB
        PERFORM ESCREVER-CABECALHO-CONTA.
IMPRIMIR-BALANCO-TRANS.
        MOVE «TRANSPORTE» TO BALANCO-TEXTO
        MOVE CONTA-TRANS TO BALANCO-QT
        PERFORM ESCREVER-LINHA-BALANCO.
GUARDAR-DETALHES-CONTA.
        MOVE CONTA-NUMERO TO NUM-CONTA-CORRENTE
        MOVE CONTRA-TRANS TO BALANCO-CORRENTE.
TRATAR-REGISTO-CORRENTE.
        IF FIM-DE-FICHEIRO-CONTA
                MOVE «V» TO ESTADO-LOTE
        ELSE
                IF CONTA-REG-TIPO = 1
                        MOVE «V» TO ESTADO-LOTE
```

```
                ELSE
                        PERFORM  PROCESSAR-REG-DET.
PROCESSAR-REG-DET.
        IF CONTA-NUMERO NOT EQUAL NUM-CONTA-CORRENTE
                PERFORM  ERRO-NUM-CONTA-INVALIDO
        ELSE
                IF CONTA-REG-TIPO = 2
                        PERFORM  PROCESSAR-VENDA
                ELSE
                        PERFORM  PROCESSAR-RECIBO.
ERRO-NUM-CONTA-INVALIDO.
        DISPLAY «NUMERO DE CONTA INVALIDO»
        DISPLAY «ENCONTRADO:», CONTA-NUMERO, «NO LOTE PARA:»,
                NUM-CONTA-CORRENTE.
PROCESSAR-VENDA.
        COMPUTE BALANCO-CORRENTE =
                BALANCO-CORRENTE + PRECO-FACTURA
        MOVE DATA-VENDA TO TRANS-DATA
        MOVE NUM-FACTURA TO TRANS-NUM-FACTURA
        MOVE PRECO-FACTURA TO TRANS-DEBITO
        MOVE BALANCO-CORRENTE TO TRANS-BALANCO
        PERFORM ESCREVER-LINHA-TRANS.
PROCESSAR-RECIBO.
        COMPUTE BALANCO-CORRENTE = BALANCO-CORRENTE — VAL-PAGA
        MOVE DATA-PAGA TO TRANS-DATA
        MOVE COD-REG-PAGA TO TRANS-COD-REG
        MOVE VAL-PAGA TO TRANS-CREDITO
        MOVE BALANCO-CORRENTE TO TRANS-BALANCO
        PERFORM ESCREVER-LINHA-TRANS.
IMPRIMIR-QT-CRED.
        MOVE «QUANTIA DEDUZIDA» TO BALANCO-TEXTO
        MOVE BALANCO-CORRENTE TO BALANCO-QT
        PERFORM ESCREVER-LINHA-BALANCO.
```

Seguidamente apresenta-se um exemplo de um possível ficheiro de *input*. Os campos são separados por espaços por questões de legibilidade, mas para o *input* do programa acima tem de ser escrito sem espaços entre os campos.

```
3 QB0012 23/08/82 C003 00100
1 CL0358 CIGRECI LDA                    05825
2 CL0358 02/08/82 G0351 1250
2 CL0358 05/08/82 T0057 2760
3 CLO358 08/08/82 C025 05825
2 CL0358 09/08/82 M1087 4368
2 CLO388 12/08/82 T0066 2205
3 CLO358 15/08/82 CO73 10583
2 CLO358 18/08/82 M1095 2870
1 CM0001 CECILIA MARTINS                00295
<fim-do-input>
```

O *output* correspondente a estes lotes de *input* deve consistir em duas mensagens de erro e duas declarações impressas na forma apresentada a seguir.

REGISTO TIPO 1 ESPERADO — ENCONTRADO 3 QB0012
NUMERO DE CONTA INVALIDO
ENCONTRADO: CL0388 NO LOTE PARA: CL0358

XYZ BOOCKSELLERS

CONTA: CLO358 NOME: CIGRECI LDA

DATA	REFERENCIA	CREDITO	DEBITO	BALANCO
		TRANSPORTE		£58.25
02/08/82	GO351		£12.50	£70.75
05/08/82	T0057		£27.60	£98.35
08/08/82	CO25	£58.25		£40.10
09/08/82	M1087		£43.68	£83.78
15/08/82	CO73	£105.83		£20.05CR
18/08/82	M1095		£28.70	£8.65
		QUANTIA DEDUZIDA		£8.65

XYZ BOOCKSELLERS

CONTA: CM0001 NOME: CECILIA MARTINS

DATA	REFERENCIA	CREDITO	DEBITO	BALANCO
		TRANSPORTE		£2.95
		QUANTIA DEDUZIDA		£2.95

Exercícios 8

8.1. De modo a verificar que não se perderam quaisquer declarações, acidental ou deliberadamente, a XYZ BOOCKSELLERS deseja numerar as páginas do mapa consecutivamente. Considere o modo em que o desenho e o programa de COBOL apresentados na secção 8.7 podem ser modificados de forma a adicionar o número de página: *a*) no princípio de cada página do mapa, ou, *b*) no fim de cada página do mapa.

8.2. A especificação para o problema no estudo de caso discutido na secção 8.7 é modificada de modo a incluir um registo de transaccão extra de tipo 4. Este é um registo de «arrasto» para um lote e, por isso um lote deve agora consistir em: um registo de tipo 1, um número qualquer de registos de tipo 2 ou 3 e um registo de tipo 4. Um lote pode consistir no registo de tipo 1 seguido de um registo de tipo 4, caso não existam vendas no mês corrente. O registo de tipo 4 contém os seguintes campos:

Registo de arrasto

Tipo de registo	4
Número de conta	2 letras e 4 dígitos
Transporte	5 dígitos, pénis, com sinal

O desenho de programa e o programa de COBOL têm de ser modificados como se segue:

a) Se um lote não termina com um registo de tipo 4, é assinalado um erro;
b) É também assinalado erro se o balanço final calculado difere do balanço a fazer apresentado no registo de tipo 4.

Em cada caso, a quantia impressa no mapa deve ser a que foi calculada para o lote.
Trabalhe o desenho de programa apresentado na secção 8.7 e considere as alterações necessárias para implementar esta nova composição.

8.3. Os seguintes extractos foram retirados de um programa de um principiante que não leu este livro.

```
. . . . . .
DATA  DIVISION.
FILE  SECTION.
FD        FICHCNT
          LABEL  RECORDS  ARE  STANDARD.
01        REG-CONTA.
          05        ID-CLIENTE                    PIC 9(6).
          . . . . . .

PROCEDURE  DIVISION.
INICIO.
          OPEN  INPUT  FICHCLI  OUTPUT  FICH-PRINT
          MOVE  0  TO  ID-CLIENTE
          PERFORM  PROCESSAR-CLIENTE  UNTIL  ID-CLIENTE  =  999999
          . . . . . .

PROCESSAR-CLIENTE.
          READ  FICHCLI
                    AT  END  MOVE  999999  TO  ID-CLIENTE.
          IF  ID-CLIENTE  NOT  =  999999
                    PERFORM  PROCESSAR-REGISTO-CORRENTE.
OUTRAS-COISAS.
          . . . . . .
```

Explique qual a razão por que o ciclo PROCESSAR-CLIENTE não irá terminar correctamente quando se atingir o fim do ficheiro.

8.4 Volte atrás ao estudo de caso da secção 6.7 e considere a forma de incluir um melhor controle das linhas de impressão por forma a fornecer um melhor *output*. As linhas de conta devem ser impressas a dois espaços, começando-se uma nova página com cabeçalhos completos sempre que tiverem sido impressas vinte e cinco linhas de detalhe.

Exercícios de programação 8

8.5. O *input* para programa de contas para a Brickbats Builders Merchants é em lotes, consistindo cada um num registo de entrega seguido de vários registos de produtos vendidos. Os conteúdos dos registos são os seguintes:

Registo entrega	
Tipo de registo	D
Número da factura	1 letra e 4 dígitos
Número de conta	8 dígitos
Nome do cliente	25 caracteres
Morada da entrega	50 caracteres
Média de desconto	2 dígitos
Método de pagamento	CB — conta bancária PC — pagamento na caixa PE — pagamento contra entrega
Produto vendido	
Tipo de registo	I
Número da factura	como acima
Número do produto	6 dígitos
Quantia encomendada	3 dígitos
Preço de retalho (UNITÁRIO)	5 dígitos, pénis

Desenhe e implemente um programa de COBOL para fazer uma conta para cada lote. Um lote deve conter um registo de entrega e, pelo menos, um registo de venda de produto. Qualquer registo de venda cuja factura não esteja de acordo com o registo de entrega corrente deve ser rejeitado com uma mensagem de erro adequada <utilize o DISPLAY>.

Cada conta deve ser produzida em páginas separadas, com um cabeçalho adequado. Por exemplo:

BRICKBATS BUILDERS MERCHANTS

FACTURA N.: R1001 NUMERO DE CONTA: 00001329

NOME: RICARDO MARTINS DESCONTO: 12%

ENTREGAR EM: R. DO LAVAIUM, 112, LISBOA

NUM. PRODUTO	QUANTIDADE	PRECO UNITARIO	PRECO COM DESCONTO
000001	100	£1.00	£88.00
123456	12	£0.15	£1.58
346700	3	£12.50	£35.00
001245	1	£225.00	£198.00
643218	7	£0.93	£5.73
		TOTAL DE VENDA	£326.31
		IVA - 15%	£48.95
		TOTAL DEDUZIDO	£375.26

Qualquer cálculo monetário que não inclua um número inteiro de pénis deve ser arredondado para o número inteiro mais próximo. Uma conta para pagamento contra entrega deverá conter a frase «PAGAMENTO CONTRA ENTREGA» adicionada abaixo do total deduzido.

9. FICHEIROS MÚLTIPLOS

O modelo de computador introduzido no capítulo 1 e utilizado nos capítulos subsequentes baseou-se no pressuposto de que a máquina tem apenas um ficheiro de *input* e um ficheiro de *output*. Esta suposição sublinha a estrutura de programa mostrada no apêndice 1, que forneceu uma cadeia suficiente para um certo número de tipos diferentes de programas.

Neste capítulo considera-se a estrutura de programas que utiliza mais do que um ficheiro de *input* ou mais do que um ficheiro de *output*. O capítulo prévio descreveu as linhas gerais para a utilização de ficheiros num programa e estas linhas são aplicadas para mostrar como utilizar com segurança ficheiros múltiplos de *input* e *output*.

Quando um programa utiliza dois ou mais ficheiros de *input*, estes precisam, normalmente, de ser combinados de uma forma determinada para produzir o *output* do programa. Isto pode causar problemas na escolha de uma estrutura de controlo segura para o programa. Na maioria dos casos, os ficheiros de *input* serão ordenados, e isto conduz-nos a um dos problemas *standard* do processamento de dados coemrciais — a actualização de ficheiros sequenciais. Duas soluções gerais para este problema são sublinhadas e comparadas, e o estudo de caso deste capítulo apresenta um exemplo de uma aplicação desta técnica.

9.1 — FICHEIROS MÚLTIPLOS DE «INPUT» E «OUTPUT»

A descrição do «Computador COBOL», na secção 1.1, supunha que a máquina consumia um único ficheiro de registos e que produzia um único ficheiro de *output* de registos. Este modelo foi já ligeiramente modificado pela introdução, na secção 7.2, da declaração Display, que permite que um pequeno volume de *output* seja conduzido para um terceiro canal de *output*.

A amplição da máquina abstracta para incluir ficheiros múltiplos de *input* e *output* é bastante linear. O esqueleto de programa apresentado no apêndice 1, utilizado como base para todos os programas analisados até aqui, pode também ser facilmente aumentado para permitir multificheiros de *input* e *output*. Muitos dos problemas ocorrem ao tentar-se construir programas seguros para lidarem com multificheiros de *input*.

Cada ficheiro de *input* de um programa terá a sua área de *input* associada, independente das outras áreas de outros ficheiros. O estado de um ficheiro de *input* pode ser definido utilizando uma condição de fim de ficheiro separada para cada ficheiro de *input*. Do mesmo modo, cada ficheiro de *output* tem a sua área de *output* própria, que é independente das áreas associadas a outros ficheiros. Isto mantém-se mesmo que existam dois ou mais ficheiros de *output* dirigidos para a linha de impressora. As linhas dirigidas para os dois ficheiros não serão comuns, porque o *software* do computador em operação irá garantir que os dois ficheiros apareçam separadamente.

Ficheiros de «input» múltiplos

A partir das secções 8.2, 8.3 e 8.4 é possível juntar um conjunto de linhas gerais para escrever e utilizar um ficheiro de *input* dentro do esquema apresentado no apêndice 1. Estas linhas para estabelecer um ficheiro de *input* são resumidas em baixo.

a) ENVIRONMENT DIVISION — cada ficheiro de *input* a processar deve ser especificado numa cláusula Select da forma:

```
SELECT  <nome-ficheiro-input>
        ASSIGN  TO  <nome-externo-ficheiro>.
```

O <nome-ficheiro-input> é o nome interno do ficheiro que será utilizado para referenciar este ficheiro ao longo do programa e construído de acordo com as regras para nomes definidos pelo utilizador (ver 3.2.). O <nome-externo-ficheiro> especifica o nome de um ficheiro e o formato deste nome depende do sistema a utilizar.

b) DATA DIVISION — cada ficheiro deve ter uma descrição de ficheiro e uma ou mais descrições de registo associadas na FILE SECTION. Deve também ter uma marca de fim de ficheiro definida na WORKING-STORAGE SECTION. A forma geral de uma descrição de ficheiro é:

```
FD      <nome-ficheiro-input>
        LABEL  RECORDS  OMITTED/STANDARD.
01      <registo-input-1>.
        ......
01      <registo-input-2>.
        ......
```

Se um ficheiro tem mais de uma descrição de registo associada, todos os registos partilham a mesma área de *input*.

A marca de fim de ficheiro deve ser definida como parte do VECTOR-ESTADO e assume a forma geral:

```
01      VECTOR-ESTADO.
        05      FIM-<nome-ficheiro-input>          PIC X.
                88      FIM-DE-<nome-ficheiro-input>      VALUE «E».
```

c) PROCEDURE DIVISION — cada ficheiro deve ser aberto para *input* antes que possa ser lido, a condição AT END deve ser lidada correctamente e o ficheiro deve ser fechado antes que o processamento termine. Um ficheiro, e a sua marca de fim de ficheiro, deve ser inicializado em ESTADO–INIC. e o ficheiro deve ser terminado em FECHO, devendo ser definido um apropriado procedimento de leitura. A forma geral da inicialização é:

```
ESTADO-INIC.
        MOVE  SPACE  TO  END-<nome-ficheiro-input>
        OPEN  INPUT  <nome-ficheiro-input>
        ......
```

Quando o processamento do ficheiro está completo, ele deve ser separado do programa; a forma geral do procedimento de terminação é:

```
FECHO.
        CLOSE <nome-ficheiro-input> ......
        STOP  RUN.
```

A forma geral do procedimento de leitura é:

```
LER-PROXIMO-<nome-ficheiro-input>.
        READ  <nome-ficheiro-input>
                AT END MOVE «E» TO FIM-<nome-ficheiro-input>.
```

Num programa que utilize dois ou mais ficheiros de *input* é necessário seguir este conjunto de linhas para cada um dos ficheiros.

Por exemplo, suponha que um programa processa dois ficheiros de *input:* FICH-CONTA, um ficheiro de apoio, e FICHEIRO-MOVIM, um ficheiro criado via teclado. Seguindo então as linhas apresentadas em cima, seriam afectadas as seguintes partes do programa:

```
ENVIRONMENT  DIVISION.
. . . . .
INPUT-OUTPUT  SECTION.
FILE-CONTROL.
        SELECT  FICH-CONTA  ASSIGN  TO  <nome-externo-1>.
        SELECT  FICHEIRO-MOVIM  ASSIGN  TO  <nome-externo-2>.
        . . . . .
DATA  DIVISION.
FILE  SECTION.
FD      FICH-CONTA  LABEL  RECORDS  STANDARD.
01      REG-CONTA.
        . . . . .
FD      FICHEIRO-MOVIM
        LABEL  RECORDS  OMITTED.
01      REG-EXISTE-MOVIM.
        . . . . .
01      REG-ADICIONA-MOVIM.
        . . . . .
01      REG-APAGA-MOVIM
        . . . . .
WORKING-STORAGE  SECTION.
01      VECTOR-ESTADO.
        05      FIM-FICH-CONTA                  PIC X.
                FIM-DE-FICHEIRO-CONTA                   VALUE «E».
        O5      FIM-FICHEIRO-MOVIM              PIC X.
                FIM-DE-FICHEIRO-MOVIM                   VALUE «E».
        . . . . .
PROCEDURE  DIVISION.
. . . . .
ESTADO-INIC.
        MOVE  SPACE  TO  FIM-FICH-CONTA
        MOVE  SPACE  TO  FIM-FICHEIRO-MOVIM
        OPEN  INPUT  FICH-CONTA  FICH-MOVIM
        . . . . .
FECHO.
        CLOSE  FICH-CONTA  FICHEIRO-MOVIM . . . . .
        STOP  RUN.
. . . . .
LER-PROXIMO-FICH-CONTA.
        READ  FICHCNT
        AT  END  MOVE  «E»  TO  FIM-FICH-CONTA.
LER-PROXIMO-FICHEIRO-MOVIM.
        READ  FICHEIRO-MOVIM
        AT  END  MOVE  «E»  TO  FIM-FICHEIRO-MOVIM.
. . . . .
```

Ficheiros de «output» múltiplos

Abaixo é dado um sumário das linhas gerais para o estabelecimento de um ficheiro de *output*, utilizando o esqueleto de programa do apêndice 1.

a) ENVIRONMENT DIVISION — requer-se uma cláusula SELECT para cada ficheiro. A forma geral é:

```
SELECT  <nome-ficheiro-output>
        ASSIGN  TO  <nome-externo-ficheiro>.
```

O <nome-ficheiro-output> é o nome utilizado para o ficheiro dentro do programa. O <nome-externo-ficheiro> tem de estar conforme às regras do sistema de COBOL utilizado.

b) DATA DIVISION — cada ficheiro deve ter uma descrição de ficheiro e uma descrição de registo associada na FILE SECTION. Deve também ter uma ou mais descrições de registo na WORKING–STORAGE SECTION. A forma geral de uma descrição de ficheiro é:

```
FD  <nome-ficheiro-output>
        LABEL  RECORDS  OMITTED/STANDARD.
01      <registo-ficheiro-output>          PIC . . .
```

Cada descrição é uma estrutura de dados que começa com o número de nível 01. Por exemplo:

```
01      <registo-output-1>.
        . . . . .
01      <registo-output-2>.
        . . . . .
```

c) PROCEDURE DIVISION — cada ficheiro deve ser aberto para *output* antes que possa ser escrito e deve ser fechado depois que o *output* foi completado. Um ficheiro deve ser inicializado em ESTADO-INIC e terminado em FECHO e devem ser definidos procedimentos de escrita apropriados, um para cada descrição de registo da WORKING-STORAGE SECTION. A forma geral da inicialização é:

```
ESTADO-INIC.
        . . . . .
        OPEN INPUT . . . . .
             OUTPUT  <nome-ficheiro-output>
        MOVE  SPACES  TO  <registo-output-1>.
```

Apenas os registos que não contenham dados constantes devem ser inicializados com espaços. No fim do programa, o procedimento de terminação é:

```
FECHO.
        CLOSE . . . . . <nome-ficheiro-output>
        STOP  RUN.
```

Os procedimentos de escrita baseiam-se numa das formas *standard:*

```
ESCREVER-<registo-output-1>.
        WRITE  <registo-ficheiro-output>  FROM  <registo-output-1>
        MOVE  SPACES  TO  <registo-output-1>.
```

```
ESCREVER-<registo-output-2>.
WRITE <registo-ficheiro-output> FROM <registo-output-2>.
```

A primeira forma é utilizada para registos que contenham apenas variáveis, enquanto a segunda é apropriada para registos contendo constantes definidas por cláusulas VALUE. Estes procedimentos podem, evidentemente, tornar-se mais complicados pela inclusão de movimentos de papel, ou paginação, para ficheiros de linha de impressora.

Num programa que utiliza vários ficheiros de *output* é necessário garantir que as linhas gerais apresentadas em cima estão de acordo com cada ficheiro de *output*.

Por exemplo, suponha que se requer um programa para fazer o *output* de dois ficheiros: FICH–CLIENTE, para o depósito de apoio, e FICH-MAPA, para a impressora. O primeiro ficheiro, FICH–CLIENTE, tem um registo associado, CLIENTE-DET, constituído inteiramente por dados variáveis. O segundo ficheiro, FICH-MAPA, tem três registos associados: CABECALHO-SUMARIO, contendo apenas dados constantes, SUMARIO-DET, consistindo inteiramente em dados variáveis, e TOTAL-SUMARIO, contendo uma mistura de constantes e variáveis. Seguindo então as regras apresentadas acima, seriam afectadas as seguintes partes do programa:

```
ENVIRONMENT DIVISION.
.....
INPUT-OUTPUT SECTION.
FILE-CONTROL.
        SELECT FICH-CLIENTE ASSIGN TO <nome-output-externo-1>.
        SELECT FICH-MAPA ASSIGN TO <nome-output-externo-2>.
.....
DATA DIVISION.
FILE SECTION.
FD      FICH-CLIENTE
        LABEL RECORDS STANDARD.
01      REG-CLIENTE                          PIC . . .
FD      FICHEIRO-MAPA
        LABEL RECORDS OMITTED.
01      LINHA-SUMARIO.                       PIC X(120).
.....
WORKING-STORAGE SECTION.
.....
01 CLIENTE-DET.
.....
01      CABECALHO-SUMARIO.
        .....
01      SUMARIO-DET.
        .....
01      SUMARIO-TOTAL.
        .....
.....
PROCEDURE DIVISION.
.....
ESTADO-INIC.
        .....
        OPEN INPUT .....
                OUTPUT FICH-CLIENTE FICH-MAPA
        MOVE SPACES TO CLIENTE-DET SUMARIO-DET.
FECHO.
        CLOSE ..... FICH-CLIENTE FICH-MAPA
        STOP RUN.
.....
```

```
ESCREVER-CLIENTE-DET.
        WRITE REG-CLIENTE FROM CLIENTE-DET
        MOVE SPACES TO CLIENTE-DET.
ESCREVER-CABECALHO-SUMARIO.
        WRITE LINHA-SUMARIO FROM CABECALHO-SUMARIO
                AFTER ADVANCING PAGE.
ESCREVER-SUMARIO-DET.
        WRITE LINHA-SUMARIO FROM SUMARIO-DET
                AFTER ADVANCING 2 LINES
        MOVE SPACES TO SUMARIO-DET.
ESCREVER-SUMARIO-TOTAL.
        WRITE LINHA-SUMARIO FROM SUMARIO-TOTAL
                AFTER ADVANCING 3 LINES.
.....
```

Os procedimentos ESCREVER-CABECALHO-SUMARIO, ESCREVER-SUMARIO-DET e ESCREVER-SUMARIO-TOTAL apresentados acima imprimirão um cabeçalho no início de cada página, com as linhas de detalhe a dois espaços, e deixarão duas linhas em branco antes da linha de total. Eles podem tornar-se mais complicados pela adição de um contador de linhas e paginação, tal como foi descrito na secção 8.5.

9.2 — ACTUALIZAÇÃO DE FICHEIROS SEQUENCIAIS

Uma importante suposição que tem sido inerente ao tipo de problemas tratados até aqui é que os dados são fornecidos como *input* para o programa, são consumidos e utilizados para produzir *output* para a impressora ou um instrumento semelhante para leitura humana. Um dos grandes poderes dos computadores é que grandes quantidades de dados podem ser armazenados sob uma forma legível pela máquina e actualizados pelos programas. O estudo de caso da secção 8.7 era irrealista porque não há razão para escrever no teclado os detalhes de conta, tais como o nome do cliente, cada vez que o programa é corrido. Os detalhes de conta deveriam ter sido armazenados num ficheiro legível pela máquina e actualizados utilizando os dados das transacções individuais.

A estrutura de ficheiros sublinhada para todos os ficheiros que foram utilizados para exemplo até agora tem sido a sequencial. Isto significa que para ser acedido um registo qualquer no ficheiro tem de se aceder previamente a todos os seus antecessores. Não é possível aceder ao trigésimo terceiro registo directamente ou aceder ao registo para o «Joe Bloggs» sem correr o ficheiro desde o princípio até que o registo requerido seja encontrado.

Quando os registos de um ficheiro sequencial são ordenados numa determinada ordem, por exemplo por número de conta, o ficheiro é sequencial ordenado. Isto significa que a actualização sistemática do ficheiro é possível pela utilização da mesma ordem para o «ficheiro mestre» e as actualizações, geralmente designadas por «ficheiro de movimentos». No entanto, o ficheiro mestre deve continuar a ser processado registo a registo desde o princípio — os registos individuais não podem ser acedidos aleatoriamente.

Ao ordenar o ficheiro, diz-se que o campo ou o conjunto de campos que definem a ordenação é a chave do ficheiro. Esta chave pode ser de qualquer tipo e tem uma ordenação definidas; tipicamente, a chave pode ser numérica, alfabética, ou uma mistura de letras e dígitos, mas podem incluir-se quaisquer outros caracteres especiais. Geralmente, a chave do ficheiro é qualquer identificador único definido especialmente, associado a alguma «coisa», por exemplo um número de conta, um número de factura ou um número parcial. Contas, por exemplo, podem ser identificadas pelo nome do cliente, mas, apesar de estes nomes poderem ser ordenados, existe o problema da unicidade dos nomes.

A ordenação de registos num ficheiro pode ser especificada em termos de uma chave composta, isto é, uma combinação de dois ou mais campos. Estes campos não precisam necessariamente de ser contíguos, mas a ordem pela qual estes são especificados é significante. Suponha que um ficheiro contém registos relacionados com várias contas diferentes, cada uma identificada com um número de conta, e que podem existir vários registos diferentes para uma determinada conta, cada uma contendo uma data. O ficheiro pode, então, ser ordenado por ordem crescente de número de conta e, relativamente a cada conta, por ordem de datas. O extracto de um ficheiro assim ordenado pode aparecer com esta configuração:

Número de conta	Data
. . . . .	
00013576	050982
00013576	070982
00013576	070982
00013576	140982
00014621	030982
00014621	110982

Neste caso, o número de conta é designado por chave principal e a data por chave secundária. Alternativamente, esta ordenação pode ser designada por ordem de data em relação ao número de conta. Uma ordenação completamente diferente resultaria da definição da data como chave principal e do número de conta como chave secundária.

É possível especificar ordenações mais complexas, por exemplo com a chave principal ordenada ascendentemente e a chave secundária descendentemente. Para simplificar, convencionar-se-á neste livro que, sempre que se utilizar uma chave composta, todos os elementos são dispostos na mesma ordem ascendente ou descendentemente.

A utilização de ficheiros sequenciais ordenados implica que é necessário ter um meio de ordenar os registos no ficheiro pela chave, numa ordem especificada. Os sistemas maiores de computador têm um utilitário SORT (ordem) que irá ordenar ficheiros ao serem fornecidos posição, tipo e comprimento de cada um dos campos que formam a chave.

Existe também um utilitário para o SORT de registos dentro de um programa de COBOL, mas isso está para além do âmbito deste livro, por duas razões:

a) A interligação com o sistema operativo é complexa e varia de sistema para sistema;

b) O módulo SORT é uma parte operacional dos *ANS '74 COBOL* e pode não estar implementado para COBOL de sistemas de computadores mais pequenos.

Desta forma, ir-se-á supor que existem utilitários para a ordenação de ficheiros numa determinada ordem e o problema da actualização dos mesmos irá ser considerado.

Para actualizar um ficheiro ordenado sequencialmente, é necessário fazer uma cópia do mesmo, incorporando as actualizações e alterações requeridas. Assim, terão de ser considerados pelo menos três ficheiros: «mestre-velho», «mestre-novo» e o de «movimentos». O ficheiro mestre-velho é o ficheiro que vai ser actualizado, utilizando os movimentos, para produzir o ficheiro novo. Se o ficheiro mestre-velho e o de movimentos forem ambos ordenados pela mesma chave, o novo ficheiro mestre será produzido com a mesma ordenação. Na prática, existirá possivelmente um quarto ficheiro para o *output* de mensagens de erro e talvez resumos de alterações, etc.

A actualização sistemática de um ficheiro sequencial só é possível se o ficheiro estiver ordenado. Assim, na parte restante deste livro, o termo «actualização de ficheiro sequencial» será convencionado como sendo a actualização de um ficheiro sequencial utilizado como um ficheiro de movimentos ordenado de modo semelhante. O diagrama *standard* que resume a actualização de um ficheiro sequencial é apresentado na figura 9.1.

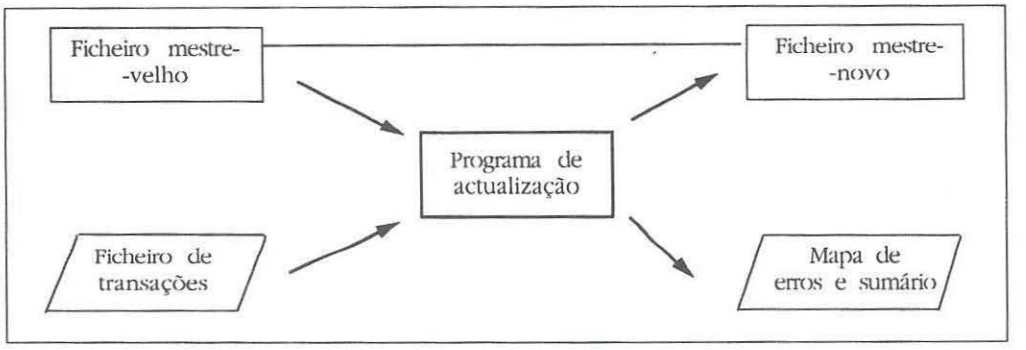


Fig. 9.1

A actualização de um ficheiro sequencial é frequentemente associada a um «ciclo» de processamento em que o novo ficheiro mestre de uma actualização se transforma no velho mestre da actualização seguinte, e assim sucessivamente. O sistema de processamento de dados tradicional faz uma actualização do ficheiro mestre pelo menos uma vez por dia, usando os movimentos efectuados durante esse dia. Assim nasce um modelo de processamento. Se existem três ficheiros mestres, identificados por A, B e C, estes são utilizados como se segue:

	Ficheiro mestre	
	Velho	Novo
Segunda-feira	A	B
Terça-feira	B	C
Quarta-feira	C	A
Quinta-feira	A	B
Etc.		

Durante o processamento de quarta-feira, por exemplo, o novo ficheiro mestre escreverá de novo (destruindo) o velho ficheiro mestre de segunda-feira. No entanto, se ocorrer uma catástrofe durante o processamento, causando a perda de dados no velho ficheiro mestre corrente (C), pode ser recriado a partir do ficheiro mestre de terça-feira (B) e do ficheiro de transacções de terça-feira. Este ciclo de três ficheiros é designado pelo sistema «avô-pai-filho», mas pode ser alargado a qualquer número de gerações, dependendo dos requesitos de segurança do sistema a ser desenvolvido. Nos primórdios do processamento de dados, cada um dos ficheiros A, B e C teria existido em banda magnética. No entanto, o mesmo procedimento pode ser aplicado a ficheiros existentes em disco magnético.

Existem métodos alternativos para organizar e actualizar ficheiros mestres, sendo alguns destes analisados no capítulo 11.

Nas duas secções seguintes, o objectivo é desenvolver esqueletos de desenhos de programa que irão executar a actualização de um ficheiro sequencial. São sublinhados dois algoritmos diferentes, um baseado nas técnicas utilizadas até agora neste livro e outro para demonstrar que pela introdução de duas novas e simples ideias pode ser desenvolvido um algoritmo melhor. Incluem-se também dois algoritmos para realçar que para um problema complexo existem muitas soluções em que frequentemente é difícil de encontrar a «melhor».

A forma geral dos *inputs* para o programa será discutida nesta secção e desenvolvidos depois os dois algoritmos nas secções 9.3 e 9.4.

Existem dois conjuntos ordenados de *input* que vêm para o programa de actualização, sendo estes combinados para produzir um único conjunto de *output* ordenado — isto, em terminologia de computação, é a união de dois ficheiros.

Para os propósitos de desenvolvimento destes desenhos de programa, ou algoritmos, serão feitas as seguintes suposições:

a) Existem três ficheiros, chamados mestre velho, mestre novo e movimentos;

b) Os registos dos três ficheiros contêm um campo de identificação semelhante, chamado chave-mestre-velho, chave-mestre-novo e chave-movimentos, respectivamente;

c) Os registos dos ficheiros mestre-velho e movimentos estão por ordem ascendente de valor de chave, isto é: o primeiro registo de cada ficheiro contém um valor de chave mais baixo e o último o valor mais alto;

d) Pode existir mais do que um movimento associado a um valor de chave particular. Neste caso, o grupo de movimentos é suposto aparecer, no ficheiro de movimentos, na ordem em que surgem as transacções;

e) Existe um campo nos movimentos, chamado tipo-movimentos, que especifica o tipo de movimentos:

- <Soma>, cria um novo registo;
- <Apaga>, destrói um registo existente;
- <Altera>, transforma um registo existente.

Os campos identificativos podem ser numéricos, alfabéticos ou uma mistura dos dois, desde que sejam do mesmo formato em cada ficheiro, por forma a poderem ser comparados. O tipo de movimento pode ser identificado por um número, uma letra ou uma palavra-chave; o modo como o tipo é identificado constitui um pormenor de implementação.

Para garantir que estes movimentos apareçam na ordem correcta, cada um devia ser «marcado» com a data e tempo de origem. Cada registo de movimento deve incluir um campo contendo esta «marca» que permite a sua distinção, em termos de tempo, de qualquer outro movimento com o mesmo valor de chave. A «marca» pode ser utilizada como chave secundária para a ordenação dos registos de movimento. Se o *input* das transacções está a ser feito directamente a partir dum terminal de computador, a «marcação» pode ser feita automaticamente pelo sistema operativo.

A ordem segundo a qual os movimentos são processados é significativa. Suponha que não existe nenhum registo de mestre-velho com o valor da chave 2864, então, a sequência de movimentos:

<apaga> 2864 <altera> 2864 <soma> 2864

não é válida porque se está a tentar apagar e depois actualizar um registo que não existe. Os movimentos <apaga> e <altera> serão rejeitados e será criado um novo registo. Os mesmos três movimentos numa ordem diferente:

<soma> 2864 <altera> 2864 <apaga> 2864

constituem uma sequência válida de movimentos.

A restrição dos algoritmos que apenas lidam com ficheiros ordenados ascendentemente é necessária de modo a que comparações de chaves possam ser especificadas. A ordem ascendente é escolhida por ser a mais usada na prática. A alteração de algoritmos para a operação em ficheiro de ordenação descendente é bastante simples.

Os algoritmos desenvolvidos não partem do pressuposto de que o ficheiro de movimentos contém dados perfeitos. O programa deve ser capaz de lidar com tipos de movimentos inválidos e com valores de chaves sem tipo definido que conduzem a tentativas inválidas de actualização de registos, etc.

Os algoritmos lidam com o caso mais geral do problema da actualização de ficheiros sequenciais. Assim, supõe-se que os registos mestre podem ser actualizados através de qualquer número de transacções, incluindo o caso em que o novo registo é criado por um movimento e

actualizado depois pelos movimentos seguintes. O problema mais geral inclui também a possibilidade de ser apagado um registo com um valor de chave determinado, sendo criado depois um novo registo com aquele valor de chave.

Numa aplicação particular, podem ser especificadas regras relacionadas com a interacção de movimentos com o mesmo valor de chave. Por exemplo, pode não ser permitido que o novo registo seja criado com a mesma chave dum registo existente que acabou de ser apagado. É também possível que possam ser colocadas restrições à eliminação de registos. Em algumas aplicações, os registos podem somente ser apagados se não tiverem sido alterados na actualização corrente ou, alternativamente, se não foram alterados num período determinado de tempo.

Os algoritmos apresentados nas secções 9.3 e 9.4 foram projectados para lidarem com o problema mais generalizado e podem ser adicionadas restrições se tal for pretendido. Deve ser realçado que estes algoritmos são apenas dois de uma variedade infinita de possibilidades.

Uma técnica alternativa normalmente usada consiste em utilizar um programa separado de validação de dados nos registos de movimentos que, tanto quanto possível, rejeite movimentos inválidos e verifique outros campos. Os movimentos validos são então ordenados utilizando a chave-movimento como chave principal e o tipo de movimento como chave secundária. As transacções com um valor de chave particular são arrumadas na ordem: <soma>, <altera> e <apaga>. Apesar de isto restringir as sequências de movimentos que são permitidas, admite as de valor pratico mais elevado.

9.3 — PRIMEIRA TENTATIVA DE UM ALGORITMO DE ACTUALIZAÇÃO

O problema geral da actualização de ficheiros sequenciais foi discutido na secção 9.2, acima. Nesta secção, o objectivo é o desenvolvimento de um algoritmo para a actualização de um ficheiro sequencial no estilo do desenho utilizado até agora neste livro.

O processo de união começa pela obtenção de um registo de cada um dos ficheiros de *input* mestre-velho e movimentos. Se a chave do registo de movimentos corrente e a chave do registo do mestre-velho forem comparadas, existem três possibilidades, apreciadas abaixo.

a) Chave-movimento maior do que chave-mestre-velho, o que significa que o registo de mestre–velho não vai ser actualizado e pode ser copiado sem alterações para o ficheiro mestre-novo;

b) Chave-movimento inferior a chave-mestre-velho, o que significa que o movimento deve ser do tipo <soma>; uma exigência de criação de um novo registo, de outro modo, será inválida;

c) As chaves são iguais, o que significa que o registo mestre-velho corrente vai ser actualizado pelo movimento corrente e possivelmente pelos movimentos seguintes.

Sempre que uma destas possibilidades aconteça, deve ser tomada a acção apropriada, incluindo a movimentação no ficheiro ou ficheiros relevantes prontos para a comparação seguinte. O ciclo deve terminar logo que um dos ficheiros tenha sido completamente utilizado.

Ao terminar o ciclo de controlo principal, é provável que um dos ficheiros de *input* tenha ainda alguns registos por processar. Se o ficheiro mestre-velho não foi processado completamente, os registos remanescentes devem ser copiados para o ficheiro mestre-novo, que é efectivamente a possibilidade *a*), acima. Alternativamente, se o ficheiro de movimentos não foi processado completamente, os únicos movimentos válidos estarão relacionados com o estabelecimento, e possivelmente actualização, de um novo registo, que é efectivamente a possibilidade *b*), acima.

O nível superior do desenho de programa pode ser então:

```
juntar-ficheiros
        ler-proximo-mestre-velho
        ler-proximo-movimentos
        until fim-de mestre-velho ou fim-de-movimentos do
                if chave-movimento > chave-mestre-velho then
                        copiar-velho-para-novo
                else
                        if chave-movimento < chave-mestre-velho then
                                possivel-reg-novo
                        else
                                actualizar-reg-existente
                        endif
                endif
        enduntil
        until fim-de-mestre-velho do
                copiar-velho-para-novo
        enduntil
        until fim-de-movimentos do
                possivel-reg-novo
        enduntil
```

É agora necessário considerar os três refinamentos para trabalhar com os três casos possíveis. O primeiro destes, «copiar-velho-para-novo», é bastante simples: o registo mestre-velho deve ser copiado para registo mestre-novo, o registo mestre-novo escrito para o ficheiro mestre-novo e o ficheiro mestre-velho adiantado em um registo. Assim, o refinamento pode ser:

```
copiar-velho-para-novo
        mover reg-mestre-velho para reg-mestre-novo
        escrever-reg-mestre-novo
        ler-proximo-mestre-velho
```

Dado que os registos mestre-velho e mestre-novo serão exactamente do mesmo formato, pode ser utilizado um *move* de grupo para a movimentação.

O refinamento seguinte, «possivel-reg-novo», deve ter um movimento para estabelecer o novo registo; caso contrário, terá ocorrido um erro. Então, este refinamento tem a seguinte forma:

```
possivel-reg-novo
        if tipo-mov = <soma> then
                estabelecer-reg-novo
                procurar-actualizações
        else
                erro-mov-novo-esperado
                ler-proximo-movimentos
        endif
```

Se for encontrado um movimento do tipo <soma>, o novo registo necessita de ser construído, no reg-mestre-novo, utilizando os detalhes do registo de movimentos. É depois preciso existir um ciclo, em «procurar-actualizações», para unir quaisquer movimentos subsequentes envolvendo este novo registo no reg-mestre-novo. Se o movimento não for do tipo <soma>, é necessário reportar um erro e adiantar o ficheiro de movimentos.

O último dos refinamentos de nível 2 é «actualizar-reg-existente». Existem dois tipos de movimentos válidos, <apaga> e <altera>, que podem ser aplicados a um registo existente. Se o tipo de movimento for <apaga>, ambos os ficheiros necessitam de ser adiantados e nada deve

ser escrito no ficheiro mestre-novo. Se o tipo de movimento for <altera>, o registo mestre-velho corrente pode ser copiado para registo mestre-novo. Isto pode ser actualizado tantas vezes quantas as necessárias por movimentos seguintes, o reg-mestre-novo escrito no ficheiro mestre-novo e o ficheiro mestre-velho adiantado e preparado para a comparação seguinte. O refinamento sugerido é então:

```
actualizar-reg-existente
        if tipo-mov = <apaga> then
                ler-proximo-mestre-velho
                ler-proximo-movimentos
        else
                if tipo-mov = <altera> then
                        mover reg-mestre-velho para reg-mestre-novo
                        unir-mov-em-reg
                        escrever-reg-mestre-novo
                        ler-proximo-mestre-velho
                else
                        inv-mov-erro
                        ler-proximo-movimentos
                endif
        endif
```

Os refinamentos que lidam com erros não vão ser especificados neste ponto; podem ser implementados pela utilização de um ficheiro de *output* separado ou declarações *display*.

Os dois refinamentos seguintes a considerar são «procurar-actualizações» e «unir-mov-em-reg». Têm funções similares a cumprir, mas resultam de condições diferentes. O refinamento «procurar-actualizações» tem de estabelecer, primeiro, se existem actualizações para o novo registo e, em caso afirmativo, aplicá-las. Ao refinamento «unir-mov-em-reg» é dado um movimento do tipo <altera> e ele deve aplicar esta e quaisquer outras actualizações subsequentes. Se «procurar-actualizações estiver codificado de modo a encontrar a primeira actualização, caso exista, para o novo registo, pode utilizar-se «unir-mov-em-reg». O refinamento sugerido é:

```
procurar-actualizações
        ler-proximo-movimentos
        if not fim-de-movimentos then
                if chave-mov = chave-mestre-novo then
                        unir-mov-em-reg
                endif
        endif
        escrever-reg-mestre-novo
```

Existe uma actualização apenas quando há outro movimento e a sua chave condiz com a chave-mestre-novo. Se qualquer destas condições falhar, o registo criado de novo é escrito no ficheiro mestre-novo e o controlo volta para o ciclo de controlo principal.

O refinamento de «unir-mov-em-reg» apresenta um problema semelhante ao de cima — a actualização acaba normalmente quando a chave-mov deixa de condizer com a chave-mestre-novo, mas pode também terminar com o fim de ficheiro de movimentos. Esta condição dupla sugere a utilização de um campo booliano para controlar o ciclo, que pode ser designado «actual-completa».

O refinamento sugerido é:

```
unir-mov-em-reg
        mover falso para actual-completa
        until actual-completa do
                actualizar-reg
```

```
                ler-proximo-movimentos
                if fim-de-movimentos then
                        mover verdadeiro para actual-completa
                else
                        if chave-mov not = chave-mestre-novo then
                                mover verdadeiro para actual-completa
                        endif
                endif
        enduntil
```

Finalmente, para completar a descrição da actualização de um ficheiro sequencial, é preciso considerar o refinamento «actualizar-reg». Existe aqui uma dificuldade com a especificação do problema. Normalmente, uma actualização será um movimento do tipo <altera>. Um movimento do tipo <soma> é obviamente inválido, mas que se passa em relação a um movimento do tipo <apaga>? Aceitando como válido um movimento do tipo <apaga> neste ponto, implica que o nOvo registo possa ser criado, ou um registo existente actualizado e destruído por um movimento subsequente. Se isto não for permitido, o refinamento é bastante simples:

```
actualizar-reg
        if tipo-mov = <altera> then
                modificar-reg-novo
        else
                inv-mov-erro
                ler-proximo-movimentos
        endif
```

O refinamento «modificar-reg-novo» utiliza simplesmente os dados do registo de tipo <altera> para actualizar os dados da área do reg-mestre-novo. Todos os outros tipos de movimento são tratados como inválidos, a acção apropriada especificada em «inv-mov-erro», e o ficheiro de movimentos adiantado para o registo seguinte.

Uma tentativa para resolver o problema mais geral, incluindo as eliminações, conduz ao movimento apresentado abaixo:

```
actualizar-reg
        if tipo-mov = <altera> then
                modificar-reg-novo
        else
                if tipo-mov = <apaga> then
                        destruir-reg-novo
                        ler-proximo-movimentos
                else
                        inv-mov-erro
                        ler-proximo-movimentos
                endif
        endif
```

Os refinamentos «modificar-reg-novo» e «inv-mov-erro» têm as mesmas funções que os descritos acima. O refinamento «destruir-reg-mestre-novo» deve reinicializar a área reg-mestre-novo para forçar uma saída do ciclo de actualização e deixar a área num estado seguro para a operação seguinte. Após apagar o registo, o ficheiro de movimentos é avançado para o registo seguinte.

Com um desenho desta complexidade é muito raro conseguir-se fazer tudo correctamente da primeira vez através do desenvolvimento do processo. Frequentemente é necessário modificar o desenho depois de uma cuidadosa verificação no papel. Se o desenho acima, para o pro-

blema mais geral, for verificado cuidadosamente no papel, vem à luz um problema com eliminações de registos. Os casos especiais em que um registo existente é actualizado e apagado, e em que o novo registo é criado e depois apagado, não são tratados convenientemente.

Apesar de o refinamento «destruir-reg-mestre-novo» reinicializar a área reg-mestre-novo, este registo vazio será ainda escrito no novo ficheiro mestre-novo nos refinamentos «actualizar-reg-existente» e «procurar-actualizações». Estes registos em branco irão provocar, subsequentemente, estragos sempre que se pretender actualizar o ficheiro mestre-novo. A solução para este problema é utilizar um campo booliano para indicar se um reg-mestre-novo deve ser escrito no ficheiro.

Pode definir-se um campo booliano «escrita-mestre-novo», que é verdadeiro se um registo deve ser escrito no ficheiro novo-mestre e falso em caso contrário. Em «actualizar-reg-existente», este campo booliano é colocado em verdadeiro antes de se juntarem as transacções ao registo corrente, em «unir-mov-em-reg», e, se ele for ainda verdadeiro mais tarde na actualização, o registo deve ser escrito no reg-novo-mestre. Assim, o refinamento previsto é:

```
actualizar-reg-existente
        if tipo-trans = <apaga> then
                ler-proximo-mestre-velho
                ler-proximo-movimentos
        else
                if tipo-trans = <altera> then
                        mover reg-mestre-velho para reg-mestre-novo
                        mover verdadeiro para escrita-mestre-novo
                        unir-mov-em-reg
                        if escrita-mestre-novo then
                                escrever-mestre-novo
                        endif
                        ler-proximo-mestre-velho
                else
                        inv-mov-erro
                        ler-proximo-movimentos
        endif
```

A modificação de «procurar-actualizações» segue um modelo semelhante. Supõe-se que o reg-mestre-novo corrente vai ser escrito no ficheiro mestre-novo, a menos que seja apagado por um movimento seguinte em «unir-mov-em-reg». O refinamento revisto é:

```
procurar-actualizações
        mover verdadeiro para escrita-mestre-novo
        ler-proximo-movimentos
        if not fim-de-movimentos then
                if chave-mov = chave-mestre-novo then
                        unir-mov-em-reg
                endif
        endif
        if escrita-mestre-novo then
                escrever-reg-mestre-novo
        endif
```

A circunstância única sob a qual o reg-novo-mestre não é escrito no ficheiro mestre-novo é quando o mesmo é destruído por um movimento do tipo <apaga>. Assim, o refinamento «destruir-reg-mestre-novo» deve incluir uma operação para colocar o campo booliano escrita-mestre-novo com o valor falso.

O esboço para este refinamento seria então:

```
destruir-reg-mestre-novo
        mover espaços para reg-mestre-novo
        mover falso para escrita-mestre-novo
```

Os espaços são movidos para reg-mestre-novo para garantir que esta área esteja correctamente inicializada se o movimento seguinte for do tipo <somar>. É possível que este refinamento inclua também a impressão de uma mensagem indicando que o registo foi deletado (apagado), imprimindo também a sua chave.

O desenho de programa completo para o problema geral, incluindo todos os deletes, é apresentado em baixo e deve ser lido cuidadosamente utilizando combinações diferentes de movimentos e de registos mestre-velho.

```
juntar-ficheiros
        ler-proximo-mestre-velho
        ler-proximo-movimentos
        until fim-de-mestre-velho or fim-de-movimentos do
                if chave-movimento > chave-mestre-velho then
                        copiar-velho-para novo
                else
                        if chave-movimento < chave-mestre-velho then
                                possivel-reg-novo
                        else
                                actualizar-reg-existente
                        endif
                endif
        enduntil

        until fim-de-mestre-velho do
                copiar-velho-para-novo
        enduntil

        until fim-de-movimentos do
                possivel-reg-novo
        enduntil

copiar-velho-para-novo
        mover reg-mestre-velho para reg-mestre-novo
        escrever-reg-mestre-novo
        ler-proximo-mestre-velho

possivel-reg-novo
        if tipo-mov = <soma> then
                estabelecer-reg-novo
                procurar-actualizações
        else
                erro-mov-novo-esperado
                ler-proximo-movimentos
        endif

actualizar-reg-existente
        if tipo-trans = <apaga> then
        ler-proximo-mestre-velho
        ler-proximo-movimentos
```

```
            else
                    if tipo-trans = <altera> then
                            mover reg-mestre-velho para reg-mestre-novo
                            mover verdadeiro para escrita-mestre-novo
                            unir-mov-em-reg
                            if escrita-mestre-novo then
                                    escrever-mestre-novo
                            endif
                            ler-proximo-mestre-velho
                    else
                            inv-mov-erro
                            ler-proximo-movimentos
                    endif

procurar-actualizações
        mover verdadeiro para escrita-mestre-novo
        ler-proximo-movimentos
        if not fim-de-movimentos then
                if chave-mov = chave-mestre-novo then
                        unir-mov-em-reg
                endif
        endif
        if escrita-mestre-novo then
                escrever-reg-mestre-novo
        endif
unir-mov-em-reg
        mover falso para actual-completa
        until actual-completa do
        actualizar-reg
        ler-proximo-movimentos
        if fim-de movimentos then
                mover verdadeiro para actual-completa
        else
                if chave-mov not = chave-mestre-novo then
                        mover verdadeiro para actual-completa
                endif
        endif
enduntil

actualizar-reg
        if tipo-mov = <altera> then
                modificar-reg-novo
        else
                if tipo-mov = <apaga> then
                        destruir-reg-novo
                        ler-proximo-movimentos
                else
                        inv-mov-erro
                        ler-proximo-movimentos
                endif
        endif

destruir-reg-mestre-novo
        mover espaços para reg-mestre-novo
        mover falso para escrita-mestre-novo
```

Na prática, um programa de actualização de um ficheiro sequencial completo incluirá, provavelmente, um utilitário para impressão de um mapa de alterações — possivelmente imprimindo os detalhes de cada registo criado, modificado ou deletado pelo programa.

Revisão crítica a este algoritmo

Existem várias soluções para qualquer problema complexo de programação. Depende dos critérios utilizados para julgar uma determinada solução particular a determinação de qual a «melhor» entre elas. Um critério muito importante para julgar um desenho de programa é a clareza do algoritmo — é óbvio o seu funcionamento? A clareza do desenho de programa conduz a outra importante consideração — até que ponto é fácil alterar o programa? No processamento de dados comerciais, espera-se que os programas tenham uma vida longa e que sejam frequentemente alterados, dadas as alterações das circunstâncias nos negócios. Isto significa que os programas são frequentemente «sujeitos a manutenção» por programadores que não foram responsáveis pelo desenho original para implementação desses programas.

A clareza e a facilidade de manutenção são mais importantes do que a eficiência do programa. Um programa altamente eficiente, cheio de «truques» inteligentes de programação, é inútil se ninguém, além do programador original, o entender, especialmente se o mesmo não estiver disponível para consultas. Incidentalmente, programas escritos para ganhar o máximo de eficiência são também difíceis de transportar para outro tipo de computador, porque tendem a utilizar sistemas peculiares de um dado sistema de COBOL.

Existe um certo número de críticas ao algoritmo apresentado em cima, que indicam que deveria haver uma solução para este problema. O nível superior da descrição do programa é reproduzido a seguir.

```
juntar-ficheiros
        ler-proximo-mestre-velho
        ler-proximo-movimentos
        until fim-de-mestre-velho or fim-de-movimentos do
                if chave-movimento > chave-mestre-velho then
                        copiar-velho-para-novo
                else
                        if chave-movimento < chave-mestre-velho then
                                possivel-reg-novo
                        else
                                actualizar-reg-existente
                        endif
                endif
        enduntil

        until fim-de-mestre-velho do
                copiar-velho-para-novo
        enduntil

        until fim-de-movimentos do
                possivel-reg-novo
        enduntil
```

Olhando para ele, existem duas críticas principais possíveis de serem feitas. Em primeiro lugar, não é imediatamente aparente por que é que existem três ciclos no programa, nem é obvio que pelo menos dois deles sejam de facto efectivos numa determinada execução do programa.

Segundo, e provavelmente mais importante, não é óbvio o que na realidade executa o ciclo principal do programa. Uma execução do ciclo não processa um reg-mestre-velho porque, por vezes, o reg–mestre-velho corrente é ignorado. Por outro lado, uma execução do ciclo não pro-

cessa sempre um movimento, em alguns casos não existem movimentos relevantes, enquanto noutras circunstâncias vários movimentos podem ser consumidos pelo ciclo.

Assim, uma pessoa pode ser levada a pensar que o ciclo de controlo principal lida com todos os registos associados a uma determinada chave. Isto não é bem assim, porque um registo mestre-velho pode ser deletado por um movimento numa execução do ciclo e um novo registo criado no ciclo seguinte com a mesma chave.

Olhando mais atentamente para o desenho de programa, depara-se-nos uma área mais problemática, que é a do delete de registos. Se é necessário uma acção especial de cada vez que um registo é deletado, existem então dois pontos no programa que têm de ser considerados: «actualizar–reg-existente» e «actualizar-reg». Isto indica que talvez seja possível melhorar o programa — como regra geral, cada função do programa deve ter um ponto claramente definido no qual seja implementado.

Outra medida da complexidade deste algoritmo é a contagem do número de pontos diferentes nos quais existem acessos aos ficheiros. Existem quatro utilizações de «ler-proximo-mestre-velho» para obter outro reg-mestre-velho, oito utilizações de «ler-proximo-movimentos» para adiantar o ficheiro de movimentos e três utilizações de «escrever-reg-mestre-novo» para escrever um registo no ficheiro mestre-novo. Regra geral, quanto menos pontos diferentes de acesso aos ficheiros existirem, mais claro será o algoritmo.

Finalnente, outra crítica que pode ser feita é o facto de «unir-mov-em-reg» ser utilizado quer por «actualizar-reg-existente», quer por «procurar-actualizações» (depois de criar um novo registo). Qualquer procedimento que seja utilizado em dois ou mais pontos do programa é potencialmente perigoso. Pode ser feita uma alteração em «unir-mov-em-reg», devido a possíveis alterações requeridas em «actualizar-reg-existente», e isto pode, inadvertidamente, afectar a acção de «procurar-actualizações».

9.4 — O ALGORITMO DE «LINHA DE BALANÇO»

O problema da actualização dos ficheiros sequenciais foi estudado por muitas pessoas e tem sido advogada uma enorme variedade de soluções. A solução em 9.3 foi resolvida através de técnicas explicadas até agora neste livro e baseadas no ciclo até ao fim de ficheiro. Alargando as técnicas disponíveis, e olhando para o problema de modo ligeiramente diferente, é possível encontrar uma solução melhor e mais clara. O método utilizado, o algoritmo da linha de balanço, foi descrito em primeiro lugar por Dijkstra em *A Discipline of Programming (Uma Disciplina de Programação).*

Tal como já foi indicado, uma dificuldade surgida na solução apresentada em 9.3 é que o ciclo de controlo principal é controlado pelo estado dos ficheiros, quando o que está, na realidade, no coração do problema é a chave dos registos. Os dados de *input* são agrupados naturalmente, quando um dos registos é formado por um grupo de registos todos com a mesma chave. Um grupo pode conter no máximo um registo mestre-velho, mas pode incluir também um número qualquer de registos de movimento. O ciclo de controlo principal de um programa deve processar um grupo de registos e deve continuar até que não existam mais grupos. A chave de um grupo pode ser estabelecida através da comparação de chaves do movimento corrente e dos registos mestre-velho atribuindo um valor mais baixo a um campo, digamos «chave-corrente». O processo deve então lidar com todos os registos que contenham valor «chave-corrente».

Pode utilizar-se uma técnica alternativa para lidar com o fim de ficheiro para conduzir o programa através do valor de chave. Suponha que define uma nova operação chamada

leitura-proximo-<nome-ficheiro-input>

que irá tentar ler outro registo a partir de <nome-ficheiro-input> e, se não existir nenhum registo disponível, irá estabelecer a chave do registo num valor impossível, demasiado alto. Por exemplo, leitura-proximo-transacções tentaria ler um registo de movimentos e, se não existisse nen-

hum registo disponível, colocaria chave-mov com <high-value> (valor mais alto possível de definir para esse campo).

O algoritmo utiliza um campo booliano, «mestre-presente», para conter o estado presente da área do registo mestre-novo. Se mestre-presente for verdadeiro, então a área reg-mestre-novo contém dados válidos; caso contrário, reg-mestre-novo está vazio.

Com base nas observações técnicas descritas acima, o nível superior do desenho para a actualização do ficheiro sequencial é:

```
linha-balanço
        leitura-proximo-movimentos
        leitura-proximo-mestre-velho
        encontrar-chave-corrente
        until chave-corrente = <high-value> do
                processar-grupo-chave-corrente
                encontrar-chave-corrente
        enduntil
```

O algoritmo começa com uma tentativa de leitura de um registo de cada um dos ficheiros de *input* e estabelece um valor para chave-corrente. O programa entra então em ciclo até que ambos os ficheiros sejam lidos na totalidade, indicado pela chave-corrente, que toma o *high-value* associado com as operações de leitura-proximo. Durante o ciclo, o grupo associado com a chave-corrente é processado, sendo depois estabelecido o novo valor de chave-corrente.

O primeiro refinamento é bastante linear:

```
encontrar-chave-corrente
        if chave-movimento < chave-mestre-velho then
                mover chave-movimento para chave-corrente
        else
                mover chave-mestre-velho para chave-corrente
        endif
```

O outro refinamento deste nível, «processar-grupo-chave-corrente», precisa de especificar as condições de início do grupo, processar todos os movimentos que condigam com chave-corrente e depois terminar o grupo. Este refinamento é apresentado em baixo.

```
processar-grupo-chave-corrente
        iniciar-grupo
        until chave-movimento not = chave-corrente do
                processar-mov-corrente
                leitura-proximo-movimentos
        enduntil
        finalizar-grupo
```

O primeiro refinamento deste procedimento é «inicializar–grupo, que define as condições de início para um grupo de registos. Se o grupo corrente contém um registo mestre-velho, este registo é copiado para reg-mestre-novo, o mestre–presente é colocado em verdadeiro para indicar que existem dados válidos em reg-mestre-novo e o ficheiro mestre-velho é avançado. Se não existe nenhum registo mestre-velho no grupo corrente, o mestre presente é colocado em falso porque não existem dados válidos em reg-mestre-novo. O refinamento pode ser então assim desenvolvido:

```
iniciar-grupo
        if chave-mestre-velho = chave-corrente then
                mover reg-mestre-velho para reg-mestre-novo
                mover verdadeiro para mestre-presente
```

```
                ler-proximo-mestre-velho
        else
                mover falso para mestre-presente
        endif
```

O refinamento «processar-mov-corrente» é simplesmente uma selecção com quatro fases:

```
processar-mov-corrente
        if tipo-mov = <soma> then
                cria-registo
        else
                if tipo-trans = <apaga> then
                        destroi-registo
                else
                        if tipo-trans = <altera> then
                                altera-registo
                        else
                                inv-mov-erro
                        endif
                endif
        endif
```

Finalmente, neste nível de refinamento, o grupo é completado pela escrita de um registo no ficheiro mestre-novo, caso existam dados válidos em reg-mestre-novo:

```
finalizar-grupo
        if mestre-presente then
                escrever-reg-mestre-novo
        endif
```

Os três refinamentos importantes que vêm de «processar–mov-corrente» estão agora bastante claros. Se o movimento é do tipo <soma> para criar um registo, se mestre-presente é verdadeiro, existe um erro. Se o contrário acontecer, deve estabelecer-se um novo registo copiando dados do movimento para reg-mestre-novo e mestre-presente deve ser colocado verdadeiro para indicar a presença deste novo dado. Assim, o refinamento é:

```
criar-registo
        if mestre-presente then
                erro-criação-ilegal
        else
                estabelecer-reg-novo
                mover verdadeiro para mestre-presente
        endif
```

Um movimento do tipo <apaga> apenas é válido se mestre–presente for verdadeiro, indicando que existe um registo para deletar. Sendo assim, o reg-mestre-novo é reinicializado e mestre-presente colocado em falso; caso contrário, ocorre um erro:

```
destruir-registo
        if mestre-presente then
                mover espaços para reg-mestre-novo
                mover falso para mestre-presente
        else
                erro-delete-ilegal
        endif
```

Do mesmo modo, um movimento do tipo <altera> só é válido se mestre-presente for verdadeiro:

```
altera-registo
        if mestre-presente then
                modificar-registo
        else
                erro-alteração-ilegal
        endif
```

Alguns refinamentos estão ainda por desenvolver — aqueles que dependem do problema mas não alteram a estrutura total do desenho. Existem seis refinamentos por especificar:

a) Inv-mov-erro, para lidar com movimentos de tipo inválido;
b) Erro-criação-ilegal, tentativa de criação de um novo registo quando já existe um com a mesma chave;
c) Estabelecer-reg-novo, copia os detalhes de um movimento do tipo <soma> para reg-mestre–novo;
d) Erro-delete-ilegal, tentativa de apagar um registo inexistente;
e) Modificar-registo, executa emendas num registo existente, especificado num movimento de tipo <altera>;
f) Erro-alteração-ilegal, tentativa de alteração de um registo inexistente.

No estudo de caso da secção 9.5 é dado um exemplo completo da implementação deste algoritmo.

O nível superior do desenho de programa para o algoritmo linha–balanço, descrito nesta secção, é mais claro do que a do algoritmo anterior, descrito na secção 9.3. Não existem ciclos extras para lidarem com a «cauda» do ficheiro mestre–velho ou movimentos, mas existe apenas um ciclo único que lida com o grupo corrente de registos. A estrutura do programa no algoritmo linha–balanço é um reflexo melhor da estrutura de dados.

Outra razão para o facto de o algoritmo linha–balanço ser mais claro é a técnica de conduzir o programa pelos valores das chaves e de tratar as condições de fim de ficheiro como valores especiais de chave. Isto evita as duplas condições inevitáveis no primeiro algoritmo, em que antes de comparar chaves era necessário verificar as ocorrências de fim-de-ficheiro-movimentos.

Na primeira tentativa do algoritmo, o delete de registo era muito desordenado, dado que um registo poderia ser deletado em dois locais do desenho de programa. Este problema desaparece no algoritmo linha–balanço, em que cada tipo de movimento tem um ponto claramente definido, no desenho no qual ganha efeito. Se, por exemplo, qualquer acção especial for associada à eliminação, obviamente que pertence a «destruir-registo».

O número de pontos de acesso ao ficheiro é muito mais pequeno no algoritmo linha balanço. Existem apenas duas utilizações de «leitura-proximo-mestre-velho» e «leitura-proximo-movimentos», em cada caso uma vez para inicialização e depois dentro de um ciclo, e existe apenas uma utilização de «escrever-reg-mestre-novo».

Implementação em COBOL

Para permitir a implementacão do algoritmo apresentado acima, as técnicas disponíveis têm de ser alargadas para completarem a ideia de <high-values> e a implementação de «leitura-próximo», etc.

A primeira parece ser muito simples, dado que o COBOL fornece uma figurativa especial constante, chamado HIGH-VALUES, que representa qualquer número de ocorrências do mais al-

to valor que pode ser representado no sistema de COBOL usado. A utilização desta constante é semelhante a SPACES. Por exemplo:

```
MOVE-HIGH-VALUES  TO  CHAVE-MOV
```

irá preencher CHAVE-MOV com o número apropriado de caracteres para o *high-value*. Pode utilizar-se esta técnica desde que não possa ocorrer uma chave contendo o *high-value*. Isto será verdadeiro para a maior parte das aplicações.

No entanto, existe uma armadilha na definição de *high-values* que irá causar dificuladades em muitos sistemas de COBOL. No exemplo acima poder-se-ia supor que CHAVE-MOV é do tipo numérico, HIGH-VALUES seria o maior valor numérico que caberia em CHAVE-MOV. Em muitos sistemas, isto não acontece e o valor associado a HIGH-VALUE será fixado como o mais alto código na «collating sequence» (sequência colateral), que provavelmente não será um carácter numérico. Assim, no exemplo acima, se CHAVE-MOV é do tipo numérico, podem ocorrer resultados indesejáveis, dado que esta declaração MOVE envolve uma transferência de dados alfanuméricos para dados numéricos.

Se este problema surgir, uma das soluções consiste em definir todos os campos-chave com alfanuméricos, mesmo que eles contenham normalmente apenas dados numéricos. Isto é muito simples, a menos que existam solicitações para operações aritméticas dos valores-chave ou para adições de *output*. Uma solução alternativa consiste em utilizar um valor numérico para todos os 9's como <high-value> desde que eles não possam ocorrer como valor válido de chave. Esta técnica é utilizada no estudo de caso da secção 9.5.

Existe uma constante figurativa análoga, chamada LOW-VALUE, que representa qualquer número de ocorrências com o valor mais baixo que pode ser representado pelo sistema de COBOL utilizado. Ela pode ser útil como valor de começo artificial para uma sequência de valores ou quando os ficheiros estão por ordem descendente do valor de chave. A discussão acima acerca do tipo da constante figurativa HIGH-VALUES aplica-se também a LOW-VALUES.

A implementação de leitura-proximo-<nome-ficheiro-input> necessita de utilizar uma forma alternativa da declaração *read:*

```
READ  <nome-ficheiro-input>  INTO  <area-registo-input>
        AT END . . . . .
```

Esta declaração constitui uma tentativa de leitura para um registo de <nome-ficheiro-input>. Se a leitura for bem sucedida, a área de *input* será copiada para <area-registo-input>, que deve ser uma estrutura de dados de número de nível 01, na WORKING-STORAGE-SECTION. Se a leitura não for bem sucedida, devido à não existência de registos para processamento, executar-se-ão as declarações que se seguem a AT END. Isto pode ser considerado equivalente a:

```
READ  <nome-ficheiro-input>
        AT END . . . . .
IF  <leitura-sucedida>
        MOVE  <area-input>  TO  <area-nome-registo>.
```

Poderia parecer que a implementação de leitura-proximo-<nome-ficheiro-input> poderia ter como base:

```
READ  <nome-ficheiro-input>
        AT END MOVE <high-value> TO <chave-reg-input>.
```

onde <chave-reg-input> é especificado na área de *input* para <nome-ficheiro-input>. No entanto, logo que ocorre a condição AT END, a área de *input* do ficheiro deixa de ser acessível; assim, não pode ter um valor para si movido, nem podem ser utilizados campos da área de *input* para operações lógicas.

A implementação sugerida para leitura-proximo-<nome-ficheiro-input> é:

```
LEITURA-PROXIMO-<nome-ficheiro-input> INTO <area-nome-registo>
        AT END MOVE <high-value> TO <area-reg-chave>.
```

Existe uma pequena desvantagem em ler para uma área de trabalho, e, assim, a redefinição deve ser utilizada para definir tipos de registo alternativos. No desenho de programa sublinhado acima, o ficheiro de movimentos tem três tipos de registo e poderiam ser definidos como se segue:

```
01      AREA-MOV.
        05      TIPO-MOV . . . . .
        05      CHAVE-MOV . . . . .
        05      TIPO-MOV-NOVO.
                . . . . .
        05      TIPO-MOV-DELETE  REDEFINES  TIPO-MOV-NOVO.
                . . . . .
        05      TIPO-MOV-ALTERA  REDEFINES  TIPO-MOV-NOVO.
                . . . . .
```

Cada uma das estruturas de dados TIPO-MOV-NOVO, TIPO-MOV-DELETE e TIPO-MOV-ALTERA deve ter o mesmo comprimento.

A definição da operação leitura-proximo para o ficheiro seria então:

```
LEITURA-PROXIMO-MOVIMENTOS.
        READ  MOVIMENTOS  INTO  AREA-MOV
                AT END MOVE <high-value> TO CHAVE-MOV.
```

A descrição de ficheiro para o MOVIMENTOS necessita de conter apenas um registo que defina caracteres suficientes para conter o maior registo do ficheiro. Por exemplo:

```
FD      MOVIMENTOS
        LABEL  RECORDS  OMITTED.
01      REG-MOV  PIC  X(<comprimento>)
```

onde <comprimento> é escolhido por forma a conter o registo maior do ficheiro de transacções.

9.5 — **ESTUDO DE CASO**

Este estudo de caso é uma ilustração da utilização do algoritmo «linha-balanço» para a actualização de ficheiros sequenciais discutida na secção 9.4. A especificação do problema é:

O banco de Ruritania mantém um ficheiro mestre de registos de contas contendo os seguintes dados:

Número de conta	8 dígitos
Nome da conta	30 caracteres
Balanço corrente	8 dígitos (pénis, com sinal)
Data da última actualização	6 dígitos (DDMMAA)

Estes registos estão armazenados por ordem ascendente de número de conta e não existe nenhuma com o número 99999999.

Movimentos relacionados com estas contas são gravados utilizando três tipos de movimentos:

Soma cliente

Tipo de registo	A
Número de conta	8 dígitos
Data da adição	6 dígitos (DDMMYY)
Nome da conta	30 caracteres
Balanço de abertura	8 dígitos (pénis)

Apaga cliente

Tipo de registo	D
Número de conta	8 dígitos
Data da supressão	6 dígitos (DDMMYY)
Código de autorização	3 letras seguidas de 2 dígitos

Alteração de balanço

Tipo de registo	C
Número de conta	8 dígitos
Data da mudança	6 dígitos (DDMMYY)
Tipo de alteração	1 = levantamento em dinheiro
	2 = cheque
	3 = cativos
	4 = dinheiro/cheques descontados
	5 = crédito a transportar
Quantia	6 dígitos
Número de referência	5 dígitos

Requer-se um programa para actualizar o ficheiro mestre utilizando um ficheiro ordenado de movimentos. Os movimentos estão ordenados por ordem ascendente de número de conta, mas, dentro de cada número de conta, pela ordem em que as actualizações foram surgindo. Os movimentos foram validados por um programa prévio, de modo a garantir que todos os campos relevantes contêm dados numéricos e que o tipo de alteração é representado por um número de 1 a 5.

O programa deve produzir um novo ficheiro mestre de registos actualizados, juntamente com dois mapas impressos: um sumário de todas as emendas feitas ao ficheiro e um mapa de erros encontrados nos dados dos movimentos. O sumário de emendas deve incluir uma linha para:

a) Cada registo deletado, mostrando o código de autorização e o balanço de fecho;
b) Todos os novos registos criados, mostrando o nome da conta e o balanço de abertura;
c) O balanço final emendado de qualquer registo actualizado durante a execução do programa.

Antes de iniciarmos o desenho de programas, podem ser definidos nomes para os campos de dados nos registos associados com o mestre e com o movimentos. O ficheiro mestre-velho contém quatro campos:

reg-mestre-velho

velho-num-conta	número de conta
velho-nome-conta	nome de conta
velho-balanço	balanço actual
velho-ult-data	data da última actualização

O ficheiro novo-mestre tem uma estrutura correspondente:

```
reg-mestre-novo
        novo-num-conta          número de conta
        novo-nome-conta         nome de conta
        novo-balanço            balanço emendado
        novo-ult-data           data da última actualização
```

O ficheiro de movimentos tem três tipos de registo, que terão de ser definidos numa estrutura de registo como se segue:

```
reg-movimento
        mov-tipo                A, C ou D
        mov-num-conta           número de conta
        mov-data                data do movimento
        mov-adição
                mov-nome-conta  nome de conta
                mov-aber-balanço        balanço de abertura
        mov-apaga == mov-adição
                mov-auto-apaga  código de autorização da supressão
        mov-altera == mov-adição
        mov-alt-tipo            tipo de alteração, 1-5
        mov-alt-quantia         quantia retirada/descontada
        mov-alt-numref          número de referência para alteração
```

A estrutura básica de desenho de programa irá estar de acordo com a definida na secção 9.4 com a adição de codificação extra para a impressão do ficheiro de emendas. O desenho delineado na secção 9.4 com as alterações de modo a incluir nomes e valores para a presente questão é apresentado a seguir. Como não existe nenhuma conta com o número 99999999, poderemos utilizar este número como um *high-value* para terminar o ciclo principal.

```
actualizar-contas
        leitura-proximo-movimentos
        leitura-proximo-mestre  velho
        encontrar-num-conta-corrente
        until num-conta-corrente = 99999999 do
                processar-num-conta-corrente
                encontrar-num-conta-corrente
        enduntil

encontrar-num-conta-corrente
        if mov-num-conta < velho-num-conta then
                mover  mov-num-conta  para  num-conta-corrente
        else
                mover  velho-num-conta  para  num-conta-corrente
        endif

processar-num-conta-corrente
        iniciar-grupo
        until move-num-conta not = num-conta-corrente do
                processar-mov-corrente
                leitura-proximo-movimentos
        enduntil
        finalizar-grupo
```

```
iniciar-grupo
        if velho-num-conta = num-conta-corrente then
                mover reg-mestre-velho para reg-mestre-novo
                mover verdadeiro para mestre-presente
                leitura-proximo-mestre-velho
        else
                mover falso para mestre-presente
        endif

processar-mov-corrente
        if tipo-mov = «A» then
                criar-registo
        else
                if tipo-trans = «D» then
                        destruir-registo
                else
                        if tipo-trans = «C» then
                                alterar-registo
                        else
                                erro-mov-ilegal
                        endif
                endif
        endif

finalizar-grupo
        if mestre-presente then
                escrever-reg-mestre-novo
        endif

criar-registo
        if mestre-presente then
                erro-criação-ilegal
        else
                estabelecer-reg-novo
                mover verdadeiro para mestre-presente
        endif

destruir-registo
        if mestre-presente then
                mover espaços para reg-mestre-novo
                mover falso para mestre-presente
        else
                erro-delete-ilegal
        endif

alterar-registo
        if mestre-presente then
                modificar-registo
        else
                erro-alteração-ilegal
        endif
```

Observando agora as modificações adicionais necessárias para a impressão do sumário de emendas, existem três tipos de registos a escrever no ficheiro para abates e emendas aos balanços. Tomemos o primeiro deles, o abate: o único ponto do programa onde um registo é aba-

tido (deletado) é em «destruir-registo», de forma que pode alterar-se para conter a necessária impressão. O refinamento revisto é:

```
destruir-registo
        if mestre-presente then
                imprimir-delete-sumário
                mover espaços para reg-mestre-novo
                mover falso para mestre-presente
        else
                erro-delete-ilegal
        endif
```

Da mesma forma, o segundo tipo de emenda, adição, pode ser abordado duma forma perfeitamente linear, pois todos os novos registos são adicionados em «criar-registo». Assim, este pode ser revisto como se segue:

```
criar-registo
        if mestre-presente then
                erro-criação-ilegal
        else

estabelecer-reg-novo
        mover verdadeiro para mestre-presente
        imprimir-adição-sumario
endif
```

O terceiro tipo de registo sumário, para movimentos emendados, requer mais atenção e pensamento. Os registos são actualizados em «alterar-registo», mas não é necessário um registo sumário por cada alteração ao registo mestre, sendo somente necessário imprimir o balanço final emendado. Assim, em «alterar-registo» é de notar que o balanço foi emendado, mas nada deve ser imprimido neste ponto. Se ocorrer que um registo tenha sido emendado, então a impressão deve ser incluída em «finalizar-grupo».

Imagine que se define um campo booliano chamado «reg-alterado» que deva ser colocado verdadeiro se se executarem emendas num registo. Então, no início de cada grupo ele terá de ser colocado em falso. Assim, «iniciar-grupo» será modificado para:

```
iniciar-grupo
        mover falso para reg-alterado
        if velho-num-conta = num-conta-corrente then
                mover reg-mestre-velho para reg-mestre-novo
                mover verdadeiro para mestre-presente
                leitura-proximo-mestre-velho
        else
                mover falso para mestre-presente
        endif
```

Sempre que um registo é actualizado, será necessário colocar o valor verdadeiro em «reg-alterado», o que implica uma alteração em «alterar-registo», como se segue:

```
alterar-registo
        if mestre-presente then
                modificar-registo
                mover verdadeiro para reg-alterado
        else
                erro-alteração-ilegal
        endif
```

No fim de cada grupo será necessário testar se alguma emenda foi efectuada e, se tal acontecer, imprimir um registo sumário. Não existe razão para se testar se existiram emendas se o reg-mestre–novo não contiver dados; assim, a forma modificada de «finalizar-grupo» poderia ser:

```
finalizar-grupo
        if mestre-presente then
                if reg-alterado then
                        imprimir-altera-sumario
                endif
                        escrever-reg-mestre-novo
        endif
```

Seria agora acertado verificar no papel todo o desenho antes de se descer a codificação pormenorizada dos níveis mais baixos de refinamento. Se o desenho acima for verificado através de várias combinações de movimentos, apercebemo-nos de que o problema com o desenho do *output* do mapa sumário é menor.

Suponha que ocorre o seguinte grupo:

```
velho-num-conta = 00001234
mov-num-conta = 00001234 mov-tipo = C (altera)
mov-num-conta = 00001234 mov-tipo = D (apaga)
mov-num-conta = 00001234 mov-tipo = A (soma)
```

O sumário impresso seria da forma:

```
supressão              com o balanço de fecho
adição          com o balanço de abertura
emenda          com o mesmo balanço da abertura, acima
```

A última linha não é necessária e irá provavelmente causar confusões. O problema é causado pela criação de um novo registo sem que se tenha recolocado «reg-alterado» em falso. Assim, «cria-registo» tem de ser posteriormente alterado para:

```
criar-registo
        if mestre-presente then
                erro-criação-ilegal
        else
                estabelecer-reg-novo
                mover verdadeiro para mestre presente
                imprimir-adição-sumario
                mover falso para reg-alterado
        endif
```

Agora, se o desenho já foi testado de novo no papel, este deverá funcionar correctamente mesmo para situações que envolvam emendas, adições e abates. Agora falta só desenvolver o número variado de refinamentos de nível mais baixo que foram deixados para trás.

O primeiro refinamento a ser considerado é «estabelecer-reg-novo», que é bastante linear:

```
estabelecer-reg-novo
        mover mov-num-conta para novo-num-conta
        mover mov-data para novo-ult-data
        mover mov-nome-conta para novo-nome-conta
        mover mov-aber-balanço para novo-balanço
```

Temos então para actualizar o reg-mestre-velho existente o refinamento «modificar-registo». Este também é bastante linear:

```
modificar-registo
        mover mov-data para novo-ult-data
        if mov-alt-tipo < 4 then
                calcular novo-balanço = novo-balanço — mov-alt-quantia
        else
                calcular novo-balanço = novo-balanço + mov-alt-quantia
        endif
```

Existem agora dois grandes grupos de refinamentos remanescentes — para lidar com a impressão dos registos do mapa sumário e os que lidam com erros. É necessário decidir em ambos os casos quais os conteúdos dos registos de *output* e depois desenvolver os refinamentos.

Para os registos do sumário poderiam ser requeridos os campos número de conta, nome de conta, o balanço corrente e uma mensagem indicando o tipo de registo no mapa sumário. Assim, o registo poderia ser definido como:

```
reg-sumario
        sumar-num-conta         número de conta
        sumar-nome-conta        nome de conta
        sumar-balanço           balanço corrente
        sumar-auto-cod          código da autorização, só abates
        sumar-tipo              tipo de registo sumário
```

Então, os três refinamentos: «imprimir-delete-sumario», «imprimir-adição-sumario» e «imprimir-altera-sumario», são lineares. Todos estes refinamentos terão de preencher o reg-sumario a partir do reg-mestre-novo e adicionar a mensagem apropriada. Assim, todos eles poderão utilizar um procedimento comum. Os refinamentos poderiam ser:

```
imprimir-delete-sumario
        mover mov-auto-apaga para sumar-auto-cod
        mover «registo abatido» para sumar-tipo
        imprimir-reg-sumario

imprimir-adição-sumario
        mover «adição de conta» para sumar-tipo
        imprimir-reg-sumario

imprimir-altera-sumario
        mover «balanço emendado» para sumar-tipo
        imprimir-reg-sumario

imprimir-reg-sumario
        mover novo-num-conta para sumar-num-conta
        mover novo-nome-conta para sumar-nome-conta
        mover novo-balanço para sumar-balanço
        escrever-reg-sumario
```

O reg-erro terá de conter campos para reter as informações de tipo de movimento, número de conta, data do movimento, nome de conta, quantia e uma mensagem para o tipo de erro. Assim, o registo poderia ser definido da seguinte forma:

reg-erro

erro-mov-tipo	tipo de movimento
erro-num-conta	número de conta
erro-mov-data	data do movimento
erro-nome-conta	nome de conta, se existir
erro-quantia	quantia envolvida, se existir
erro-tipo	tipo de erro

Poderia ser adoptada uma estratégia similar, a utilizada para os registos sumário, extraindo os elementos comuns da construção do reg-erro. Os refinamentos então seriam:

```
erro-mov-ilegal
        mover «tipo movimento ilegal» para erro-tipo
        imprimir-reg-erro

erro-criação-ilegal
        mover mov-nome-conta para erro-nome-conta
        mover mov-aber-balanço para erro-quantia
        mover «criar: conta existente» para erro-tipo
        imprimir-reg-erro

erro-delete-ilegal
        mover «abate: registo inexistente» para erro-tipo
        imprimir-reg-erro

erro-alteração-ilegal
        mover mov-alt-quantia para erro-quantia
        mover «actualiza: registo inexistente» para erro-tipo
        imprimir-reg-erro

imprimir-reg-erro
        mover mov-tipo para erro-mov-tipo
        mover mov-num-conta para erro-num-conta
        mover mov-data para erro-mov-data
        escrever-reg-erro
```

Finalmente, irá ser necessário produzir cabeçalhos para os dois mapas e necessitarão de ser imprimidos no início do programa. Assim, o nível superior do desenho de programa terá de os incluir. O desenho de programa completo, incluindo esta alteração, é dado abaixo.

```
actualizar-contas
        imprimir-cabeçalhos
        leitura-proximo-movimentos
        leitura-proximo-mestre-velho
        encontrar-num-conta-corrente
        until num-conta-corrente = 99999999 do
                processar-num-conta-corrente
                encontrar-num-conta-corrente
        enduntil

imprimir-cabeçalhos
        escrever-cabec-pag-sumar (após nova página)
        escrever-cabec-col-sumar (após saltar duas linhas)
        escrever-cabec-pag-erro (após nova página)
        escrever-cabec-col-erro (após saltar duas linhas)
```

```
encontrar-num-conta-corrente
        if mov-num-conta < velho-num-conta then
                mover mov-num-conta para num-conta-corrente
        else
                mover velho-num-conta para num-conta-corrente
        endif

processar-num-conta-corrente
        iniciar-grupo
        until move-num-conta not = num-conta-corrente do
                processar-mov-corrente
                leitura-proximo-movimentos
        enduntil
        finalizar-grupo

iniciar-grupo
        mover falso para reg-alterado
        if velho-num-conta = num-conta-corrente then
                mover reg-mestre-velho para reg-mestre-novo
                mover verdadeiro para mestre-presente
                leitura-proximo-mestre-velho
        else
                mover falso para mestre-presente
        endif

processar-mov-corrente
        if tipo-mov = «A» then
                criar-registo
        else
                if tipo-trans = «D» then
                        destruir-registo
                else
                        if tipo-trans «C» then
                                alterar-registo
                        else
                                erro-mov-ilegal
                        endif
                endif
        endif

finalizar-grupo
        if mestre-presente then
                if reg-alterado then
                        imprimir-altera-sumario
                endif
                        escrever-reg-mestre-novo
        endif

criar-registo
        if mestre-presente then
                erro-criação-ilegal
```

```
        else
                estabelecer-reg-novo
                mover verdadeiro para mestre-presente
                imprimir-adição-sumario
                mover falso para reg-alterado
        endif

destruir-registo
        if mestre-presente then
                imprimir-delete-sumario
                mover espaços para reg-mestre-novo
                mover falso para mestre-presente
        else
                erro-delete-ilegal
        endif

alterar-registo
        if mestre-presente then
                modificar-registo
                mover verdadeiro para reg-alterado
        else
                erro-alteração-ilegal
        endif

estabelecer-reg-novo
        mover mov-num-conta para novo-num-conta
        mover mov-data para novo-ult-data
        mover mov-nome-conta para novo-nome-conta
        mover mov-aber-balanço para novo-balanço

modificar-registo
        mover mov-data para novo-ult-data
        if mov-alt-tipo < 4 then
                calcular novo-balanço = novo-balanço — mov-alt-quantia
        else
                calcular novo-balanço = novo-balanço + mov-alt-quantia
        endif

imprimir-delete-sumario
        mover mov-auto-apaga para sumar-auto-cod
        mover «registo abatido» para sumar-tipo
        imprimir-reg-sumario

imprimir-adição-sumario
        mover «adição de conta» para sumar-tipo
        imprimir-reg-sumario

imprimir-altera-sumario
        mover «balanço emendado» para sumar-tipo
        imprimir-reg-sumario
```

imprimir-reg-sumario
mover novo-num-conta para sumar-num-conta
mover novo-nome-conta para sumar-nome-conta
mover novo-balanço para sumar-balanço
escrever-reg-sumario
erro-mov-ilegal
mover «tipo movimento ilegal» para erro-tipo
imprimir-reg-erro

erro-criação-ilegal
mover mov-nome-conta para erro-nome-conta
mover mov-aber-balanço para erro-quantia
mover «criar: conta existente» para erro-tipo
imprimir-reg-erro

erro-delete-ilegal
mover «abate: registo inexistente» para erro-tipo
imprimir-reg-erro

erro-alteração-ilegal
mover mov-alt-quantia para erro-quantia
mover «actualiza: registo inexistente» para erro-tipo
imprimir-reg-erro

imprimir-reg-erro
mover mov-tipo para erro-mov-tipo
mover mov-num-conta para erro-num-conta
mover mov-data para erro-mov-data
escrever-reg-erro

Após a verificação cuidadosa deste desenho no papel, o programa de COBOL correspondente pode ser desenvolvido.

```
PROCESS LET
IDENTIFICATION DIVISION.
PROGRAMA-ID. ACTUAL.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-S34
OBJECT-COMPUTER. IBM-34.
*       <nome-computador> deve ser substituído pelo nome
*       do computador usado para correr os programas.

INPUT-OUTPUT SECTION.
FILE-CONTROL.
        SELECT NMOVIM ASSIGN TO DISK-NMOVIM.
        SELECT MESTREV ASSIGN TO DISK-MESTREV.
        SELECT MESTREN ASSIGN TO DISK-MESTREN.
        SELECT MAPSUM ASSIGN TO PRINTER-MAPSUM.
        SELECT MAPERRO ASSIGN TO PRINTER-MAPERRO.
*       <sistema-input> e <sistema-output> devem estar conformes
*       com as regras do sistema utilizado.
```

```
DATA DIVISION.
FILE SECTION.
FD  NMOVIM
    LABEL RECORDS STANDARD.
01  MOV-INPUT-AREA                    PIC X(53).
FD  MESTREV
    LABEL RECORDS STANDARD.
01  MESTREV-INPUT-AREA                PIC X(52).
FD  MESTREN
    LABEL RECORDS STANDARD.
01  MESTREN-OUTPUT-AREA               PIC X(52).
FD  MAPSUM
    LABEL RECORDS OMITTED.
01  LINHA-SUMARIO                     PIC X(81).
FD  MAPERRO
    LABEL RECORDS OMITTED.
01  LINHA-ERRO                        PIC X(93).
WORKING-STORAGE SECTION.
01  VECTOR-ESTADO.
    05      ESTADO-REG                 PIC X.
            88      REG-ALTERADO                    VALUE «V».
    05      ESTADO-MESTRE              PIC X.
            88      MESTRE-PRESENTE                 VALUE «V».
    05      NUM-CONTA-CORRENTE         PIC 9(8).
01  REG-MOVIMENTO.
    05      MOV-TIPO                   PIC A.
    05      MOV-NUM-CONTA              PIC 9(8).
    05      MOV-DATA                   PIC 9(6).
    05      MOV-ADICAO.
            10      MOV-NOME-CONTA     PIC X(30).
            10      MOV-ABER-BALANCO   PIC 9(6)V99.
    05      MOV-APAGA REDEFINES MOV-ADICAO.
            10      MOV-AUTO-APAGA     PIC X(5).
            10      FILLER             PIC X(33).
    05      MOV-ALTERA REDEFINES MOV-ADICAO.
            10      MOV-ALT-TIPO       PIC 9.
            10      MOV-ALT-QUANTIA    PIC 9999V99.
            10      MOV-ALT-NUMREF     PIC 9(5).
            10      FILLER             PIC X(26).
01  REG-MESTRE-VELHO.
    05      VELHO-NUM-CONTA            PIC 9(8).
    05      VELHO-NOME-CONTA           PIC X(30).
    05      VELHO-BALANCO              PIC S9(6)V9.
    05      VELHO-ULT-DATA             PIC 9(6).
01  REG-MESTRE-NOVO.
    05      NOVO-NUM-CONTA             PIC 9(8).
    05      NOVO-NOME-CONTA            PIC X(30).
    05      NOVO-BALANCO               PIC S9(6)V99.
    05      NOVO-ULT-DATA              PIC 9(6).
01  CABEC-PAG-SUMAR.
    05      FILLER                     PIC X(24)    VALUE SPACES.
    05      FILLER                     PIC X(33)    VALUE
            «MAPA DE EMENDAS AO FICH. MESTRE».
01  CABEC-COL-SUMAR.
    05      FILLER                     PIC X(7)     VALUE «NUMERO».
```

```
    05      FILLER                  PIC X(12)       VALUE  SPACES.
    05      FILLER                  PIC X(12)       VALUE  «NOME  CONTA».
    05      FILLER                  PIC X(13)       VALUE  SPACES.
    05      FILLER                  PIC X(7)        VALUE  «BALANCO».
    05      FILLER                  PIC X(5)        VALUE  SPACES.
    05      FILLER                  PIC X(6)        VALUE  «AUTOR.».
    05      FILLER                  PIC X(2)        ALUE  SPACES.
    05      FILLER                  PIC X(15)       VALUE «TIPO DE ACTUAL.».
01  REG-SUMARIO.
    05      SUMAR-NUM-CONTA         PIC 9(8).
    05      FILLER                  PIC XX.
    05      SUMAR-NOME-CONTA        PIC X(30).
    05      FILLER                  PIC XX.
    05      SUMAR-BALANCO           PIC  ££££,££9.99DB.
    05      FILLER                  PIC XX.
    05      SUMAR-AUTO-COD          PIC X(5).
    05      FILLER                  PIC XX.
    05      SUMAR-TIPO              PIC X(17).
01  CABEC-PAG-ERRO.
    05      FILLER                  PIC X(34)       VALUE  SPACES.
    05      FILLER                  PIC X(24)              VALUE
                «MAPA  MOVIMENTOS  ERRADOS».
01  CABEC-COL-ERRO.
    05      FILLER                  PIC X(4)        VALUE  «TIPO».
    05      FILLER                  PIC XXX         VALUE  SPACES.
    O5      FILLER                  PIC X(6)        VALUE  «NUMERO».
    05      FILLER                  PIC XXXX        VALUE  SPACES.
    05      FILLER                  PIC X(4)        VALUE  «DATA».
    05      FILLER                  PIC X(12)       VALUE  SPACES.
    05      FILLER                  PIC X(10)       VALUE  «N0ME  CONTA».
    05      FILLER                  PIC X(15)       VALUE  SPACES.
    05      FILLER                  PIC X(7)        VALUE  «QUANTIA».
    05      FILLER                  PIC X(9)        VALUE  SPACES.
    05      FILLER                  PIC X(12)       VALUE «TIPO DE ERRO».
01  REG-ERRO.
    05      FILLER                  PIC X.
    05      ERRO-MOV-TIPO           PIC X.
    05      FILLER                  PIC XXXX.
    05      ERRO-NUM-CONTA          PIC 9(8).
    05      FILLER                  PIC XX.
    05      ERRO-MOV-DATA           PIC 9(6).
    05      FILLER                  PIC XX.
    05      ERRO-NOME-CONTA         PIC X(30).
    05      FILLER                  PIC XX.
    05      ERRO-QUANTIA            PIC  £££££,££9.99.
    05      FILLER                  PIC XX.
    05      ERRO-TIPO               PIC (24).
    PROCEDURE  DIVISION.
    PROGRAMA-PRINCIPAL.
    PERFORM  ESTADO-INIC
    PERFORM  ACTULIZAR-CONTAS
    PERFORM  FECHO.
ESTADO-INIC.
    OPEN  INPUT  NMOVIM  MESTREV
          OUTPUT  MESTREN  MAPSUN  MAPERRO
```

```
        MOVE SPACES TO REG-MESTRE-NOVO
                REG-SUMARIO REG-ERRO.
FECHO
        CLOSE NMOVIN MESTREN MESTREV
                MAPSUM MAPERRO
        STOP RUN
LEITURA-PROXIMO-MOVIMENTOS.
        READ NMOVIM INTO REG-MOVIMENTO
                AT END MOVE 99999999 TO MOV-NUM-CONTA.
LEITURA-PROXIMO-MESTRE-VELHO.
        READ MESTREV INTO REG-MESTRE-VELHO
                AT END MOVE 99999999 TO VELHO-NUM-CONTA.
ESCREVER-REG-MESTRE-NOVO.
        WRITE MESTREN-OUTPUT-AREA FROM REG-MESTRE-NOVO
        MOVE SPACES TO REG-MESTRE-NOVO.
ESCREVER-CABEC-PAG-SUMAR.
        WRITE LINHA-SUMARIO FROM CABEC-PAG-SUMAR
                AFTER ADVANCING PAGE.
ESCREVER-CABEC-COL-SUMAR.
        WRITE LINHA-SUMARIO FROM CABEC-COL-SUMAR
                AFTER ADVANCING 2 LINES.
ESCREVER-REG-SUMARIO.
        WRITE LINHA-SUMARIO FROM CABEC-COL-SUMAR
                AFTER ADVANCING 2 LINES.
ESCREVER-REG-SUMARIO.
        WRITE LINHA-SUMARIO FROM REG-SUMARIO
                AFTER ADVANCING 1 LINES
        MOVE SPACES TO REG-SUMARIO.
ESCREVER-CABEC-PAG-ERRO.
        WRITE LINHA-ERRO FROM CABEC-PAG-ERRO
                AFTER ADVANCING PAGE.
ESCREVER-CABEC-COL-ERRO.
        WRITE LINHA-ERRO FROM CABEC-COL-ERRO
                AFTER ADVANCING 2 LINES.
ESCREVER-REG-ERRO.
        WRITE LINHA-ERRO FROM REG-ERRO
                AFTER ADVANCING 1 LINES
        MOVE SPACES TO REG-ERRO.
ACTUALIZAR-CONTAS.
        PERFORM IMPRIMIR-CABECALHOS
        PERFORM LEITURA-PROXIMO-MOVIMENTOS
        PERFORM LEITURA-PROXIMO-MESTRE-VELHO
        PERFORM ENCONTRAR-NUM-CONTA-CORRENTE
        PERFORM CICLO-CONTA UNTIL NUM-CONTA-CORRENTE = 99999999.
CICLO-CONTA.
        PERFORM PROCESSAR-NUM-CONTA-CORRENTE
        PERFORM ENCONTRAR-NUM-CONTA-CORRENTE.
IMPRIMIR-CABECALHOS.
        PERFORM ESCREVER-CABEC-PAG-SUMAR
        PERFORM ESCREVER-CABEC-COL-SUMAR
        PERFORM ESCREVER-PAG-ERRO
        PERFORM ESCREVER-CABEC-COL-ERRO.
```

```
    ENCONTRAR-NUM-CONTA-CORRENTE.
            IF MOV-NUM-CONTA < VELHO-NUM-CONTA
                    MOVE MOV-NUM-CONTA TO NUM-CONTA-CORRENTE
            ELSE
                    MOVE VELHO-NUM-CONTA TO NUM-CONTA-CORRENTE.
    PROCESSAR-NUM-CONTA-CORRRENTE.
            PERFORM INICIAR-GRUPO
            PERFORM CICLO-MOV
                    UNTIL MOV-NUM-CONTA NOT = NUM-CONTA-CORRENTE
            PERFORM FINALIZAR-GRUPO.
    CICLO-MOV.
            PERFORM PROCESSAR-MOV-CORRENTE
            PERFORM LEITURA-PROXIMO-MOVIMENTOS.
    INICIAR-GRUPO.
            MOVE «F» TO ESTADO-REG
            IF VELHO-NUM-CONTA = NUM-CONTA-CORRENTE
                    MOVE REG-MESTRE-VELHO TO REG-MESTRE-NOVO
                    MOVE «V» TO ESTADO-MESTRE
                    PERFORM LEITURA-PROXIMO-MESTRE-VELHO
            ELSE
            MOVE «F» TO ESTADO-MESTRE.
    PROCESSAR-MOV-CORRENTE.
            IF MOV-TIPO = «A»
                    PERFORM CRIAR-REGISTO
            ELSE
            IF MOV-TIPO = «D»
                    PERFORM DESTRUIR-REGISTO
            ELSE
                    IF MOV-TIPO = «C»
                            PERFORM ALTERAR-REGISTO
                    ELSE
                            PERFORM ERRO-MOV-ILEGAL.
    FINALIZAR-GRUPO.
            IF MESTRE-PRESENTE
                    PERFORM VERIFICAR-REG-ALTERADO
                    PERFORM ESCREVER-REG-MESTRE-NOVO.
    VERIFICAR-REG-ALTERADO.
            IF REG-ALTERADO
                    PERFORM IMPRIMIR-ALTERA-SUMARIO.
    CRIAR-REGISTO.
            IF MESTRE-PRESENTE
                    PERFORM ERRO-CRIACAO-ILEGAL
    ELSE
            PERFORM ESTABELECER-REG-NOVO
            MOVE «V» TO ESTADO-MESTRE
            PERFORM IMPRIMIR-ADICAO-SUMARIO
            MOVE «F» TO ESTADO-REG.
DESTRUIR-REGISTO.
    IF MESTRE-PRESENTE
            PERFORM IMPRIMIR-DELETE-SUMARIO
            MOVE SPACES TO REG-MESTRE-NOVO
            MOVE «F» TO ESTADO-MESTRE
```

```
        ELSE
                PERFORM-DELETE-ILEGAL.
ALTERAR REGISTO.
        IF MESTRE-PRESENTE
                PERFORM MODIFICAR-REGISTO
                MOVE «V» TO ESTADO-REG
        ELSE
                PERFORM ERRO-ALTERACAO-ILEGAL.
ESTABELECER-REG-NOVO.
        MOVE MOV-NUM-CONTA TO NOVO-NUM-CONTA
        MOVE MOV-DATA TO NOVO-ULT-DATA
        MOVE MOV-NOME-CONTA TO NOVO-NOME-CONTA
        MOVE MOV-ABER-BALANCO TO NOVO BALANCO.
MODIFICAR-REGISTO.
        MOVE MOV-DATA TO NOVO-ULT-DATA
        IF MOV-ALT-TIPO < 4
                COMPUTE NOVO-BALANCO = NOVO-BALANCO — MOV-ALT-QUANTIA
        ELSE
                COMPUTE NOVO-BALANCO = NOVO-BALANCO + MOV-ALT-QUANTIA.
        IMPRIMIR-DELETE-SUMARIO.
                MOVE MOV-AUTO-APAGA TO SUMAR-AUTO-COD
                MOVE «REGISTO ABATIDO» TO SUMAR TIPO
                PERFORM IMPRIMIR-REG-SUMARIO.
        IMPRIMIR-ADICAO-SUMARIO.
                MOVE «ADICAO DE CONTA» TO SUMAR-TIPO
                PERFORM IMPRIMIR-REG-SUMARIO.
        IMPRIMIR-ALTERA-SUMARIO.
                MOVE «BALANCO EMENDADO» TO SUMAR-TIPO
                PERFORM IMPRIMIR-REG-SUMARIO.
        IMPRIMIR-REG-SUMARIO.
                MOVE NOVO-NUM-CONTA TO SUMAR-NUM-CONTA
                MOVE NOVO-NOME-CONTA TO SUMAR-NOME-CONTA
                MOVE NOVO-BALANCO TO SUMAR BALANCO
                PERFORM ESCREVER-REG-SUMARIO.
        ERRO-MOV-ILEGAL.
                MOVE «TIPO MOVIMENTO ILEGAL» TO ERRO-TIPO
                PERFORM IMPRIMIR-REG-ERRO.
        ERRO-CRIACAO-ILEGAL.
                MOVE MOV-NOME-CONTA TO ERRO-NOME-CONTA
                MOVE MOV-ABER-BALANCO TO ERRO-QUANTIA
                MOVE «CRIAR: CONTA EXISTENTE» TO ERRO-TIPO
                PERFORM IMPRIMIR-REG-ERRO.
        ERRO-DELETE-ILEGAL.
                MOVE «ABATE: REG INEXISTENTE» TO ERRO-TIPO
                PERFORM IMPRIMIR-REG-ERRO.
        ERRO-ALTERACAO-ILEGAL.
                MOVE MOV-ALT-QUANTIA TO ERRO-QUANTIA
                MOVE «ACTUAL.: REG INEXISTENTE» TO ERRO-TIPO
                PERFORM IMPRIMIR-REG-ERRO.
        IMPRIMIR-REG-ERRO.
                MOVE MOV-TIPO TO ERRO-MOV-TIPO
                MOVE MOV-NUM-CONTA TO ERRO-NUM-CONTA
                MOVE MOV-DATA TO ERRO-MOV-DATA
                PERFORM ESCREVER-REG-ERRO.
```

Para ilustrar a operação deste programa são dados em baixo exemplos de um ficheiro mestre-velho e um de movimentos. Os campos são separados por espaços por questões de legibilidade e os balanços nos registos do mestre-velho são apresentados com sinais de menos, quando apropriado. Este não é o modo pelo qual os registos seriam apresentados pelo programa acima.

Mestre-velho

```
01002546  C. MARRECO          00000123      090982
01002681  N. MARRECO          00001036—     060982
02001481  D. MELO             00100200      020982
02001732  D. MELO             00001000—     260882
02002812  F. J. DA SILVA       00000000      090982
02002915  J. T. DA SILVA       00221316      030982
02003016  M. SILVA PRETO       00056521      200882
02006058  N. N. MARCOS         00102532      030982
03001231  J. TRAVASSOS         00001225—     060982
03002314  T. J. V. AZEVEDO     00256210—     010982
```

Movimentos

```
D 01003515 100982 RCP01
C 01003515 2 002595 01258
D 01004218 100982 RCP02
A 02001533 100982 N. MARRECO                00001000
F 02001732 100982 RCP03
A 02002915 J. T. DA SILVA                   00000100
C 02003016 100982 1 005000 99999
C 02003016 100982 2 015010 00258
C 02003016 100982 3 002500 10281
C 03001231 100982 5 010432 56214
```

O ficheiro mestre-novo resultante da execução do programa seria:

Mestre-novo

```
01002546  C. MARRECO          00000123      090982
02001481  D. MELO             00100200      020982
02001533  D. MELO             00001000      100982
02001732  D. MELO             00001000—     260882
02002812  F. J. DA SILVA       00000000      090982
02002915  J. T. DA SILVA       00221316—     030982
02003016  M. SILVA PRETO       00034011      100982
02006058  N. N. MARCOS         00102532      030982
03001231  J. TRAVASSOS         00009207      100982
03002314  T. J. V. AZEVEDO     00256210—     010982
```

Os sumários das actualizações e dos erros associados a esta actualização são mostrados nas figuras 9.2 e 9.3, respectivamente.

NUMERO	NOME CONTA	BALANCO	AUTOR.	TIPO DE ACTUAL.
01002546	C. MARRECO	£1.23	RCP01	REGISTO ABATIDO
02001553	N. MARRECO	£10.00		ADICAO DE CONTA
02003016	M. SILVA PRETO	£340.11		BALAN. EMENDADO
03001231	J. TRAVASSOS	£92.07		BALAN. EMENDADO

Fig. 9.2

TIPO	NUMERO	DATA	NOME CONTA	QUANTIA	TIPO DE ERRO
C	01003515	100982		£25.95	ACTUAL.: REG INEXISTENTE
D	01004218	100982			ABATE: REG INEXISTENTE
F	02001732	100982			TIPO MOVIMENTO ILEGAL
A	02002915	100982	J. T. DA SILVA	£1.00	CRIAR: CONTA EXISTENTE

Fig. 9.3

Exercícios 9

9.1. O Banco de Ruritania deseja modificar o mapa delineado para o estudo de caso da secção 9.5. Os primeiros dois dígitos do número de conta servem para identificar diferentes categorias de clientes. Assim, cada vez que ocorra um alteração nos dois primeiros dígitos do número de conta, o mapa sumário deve começar uma nova página. Considere o modo de incorporar esta alteração no desenho de programa e programa apresentados na secção 9.5.

Debruce-se de novo sobre a primeira tentativa de definição de algoritmo da secção 9.3 e considere em que ponto do desenho deve ser introduzido um tipo semelhante de teste para a alteração de categoria.

9.2. Desenhe um conjunto de dados de teste para testar consistentemente o programa da secção 9.5. [Pista: divida esta tarefa em várias subtarefas; por exemplo: considere as diferentes combinações de movimentos com registos mestre-velho, repare na criação e subsequente actualização de registos, identifique os modos pelos quais os erros podem ocorrer, teste as várias possibilidades para cada tipo de movimento.]

9.3. Analise o algoritmo de linha-balanço da secção 9.4 e sugira a forma segundo a qual ele será modificado para poder lidar com ficheiros ordenados por ordem descendente do valor de chave.

Quais são as alterações necessárias na primeira tentativa de algoritmo, apresentado na secção 9.3, para lidar com ficheiros ordenados descendentemente?

Exercícios de programação 9

9.4. Uma agência de revenda de motores mantém um grande *stock* de peças sobresselentes para vários tipos de motores. Foi criado um ficheiro mestre de registos de *stock*, contendo cada um os seguintes campos:

Referência da peça	10 dígitos
Breve descrição da peça	25 caracteres
Quantidade disponível	4 dígitos
Stock mínimo	4 dígitos
Quantidade a encomendar	4 dígitos
Peças já encomendadas	R = peças encomendadas

Este ficheiro está ordenado ascendentemente por referência de peça e deve ser actualizado regularmente por um ficheiro de movimentos contendo as alterações. O primeiro tipo de movimento diz respeito a uma venda que resulta de uma retirada do *stock*, representada da seguinte forma:

Retirada do stock

Tipo de movimento	RT
Referência da peça	10 dígitos
Quantidade retirada	3 dígitos

Se, depois de actualizada a quantidade existente, ela ficar com quantidade abaixo do *stock* mínimo e a marca indicativa de peça a encomendar não for igual a «R», deve ser imprimida uma mensagem que indique a quantidade a encomendar e a marca indicativa de peça a encomendar colocada em «R». Se duma retirada do *stock* resultar que a quantidade existente fique negativa, deve ser impressa uma mensagem de aviso.

Após a recepção do *stock*, cria-se um registo de movimentos para cada tipo de peça recebida:

Stock recebido

Tipo de movimento	RE
Referência da peça	10 dígitos
Quantidade recebida	4 dígitos

Os dados deste registo devem ser utilizados para actualizar a quantidade existente e a marca indicativa de peça a encomendar colocada num valor diferente de «R».

Para criar um registo para uma nova peça, a informação completa a ser introduzida é a seguinte:

Peça nova

Tipo de movimento	PN
Referência da peça	10 dígitos
Breve descrição da peça	25 caracteres
Quantidade inicial em *stock*	4 dígitos
Stock mínimo	4 dígitos
Quantidade a encomendar	4 dígitos

O Departamento de Peças necessita de um utilitário para alterar os níveis de *stock* mínimo ou de quantidades a encomendar para uma determinada peça. Estas alterações são especificadas por movimentos do seguinte formato:

Alterações

Tipo de movimento	AL
Referência da peça	10 dígitos
Alteração requerida	1 = *stock* mínimo 2 = quantidade a encomendar 3 = ambos, *stock* e quantidade
Novo *stock* mínimo	4 dígitos
Nova quantidade a encomendar	4 dígitos

Finalmente, tem de existir um utilitário para a eliminação de registos relacionados com o *stock* obsoleto. Isto dá origem a um movimento de tipo:

Stock obsoleto

Tipo de movimento	SO
Referência de peça	10 dígitos

Um registo de *stock* obsoleto deve apenas ser removido do ficheiro mestre se a quantidade existente for igual a zero.

Pretende-se um programa que aceite um ficheiro de movimentos, ordenado ascendentemente por referência de peça, e que actualize o ficheiro mestre de acordo com esta referência de peça, e que actualize o ficheiro mestre de acordo com esta ordenação. Este programa deve imprimir dois mapas: um com movimentos inválidos e outro resumindo as acções do programa, tais como registos adicionados, eliminados ou encomendados, e as acções pretendidas, tais como novas encomendas e balanços de *stocks* negativos encontrados.

10. TABELAS

A informação apresentada sob a forma tabular aparece com muita frequência na vida quotidiana. Num jornal vulgar podem aparecer tabelas resumindo os benefícios dos seguros de vida para um determinado prémio e idade de proponente, tabelas de clubes de futebol e tabelas de taxas do nível de conversão. A característica de todas elas é a repetição de informação semelhante — os valores são repetidos para muitos números de chave ou combinações de números de chave.

Alguns dos dados introduzidos nos computadores são repetitivos, por exemplo valores determinados para cada mês ou dados para uma grande organização apresentados sob a forma de um conjunto de valores semelhantes para cada secção ou departamento. Frequentemente se pretende que os programas de computador produzam um *output* em forma de tabela através da análise ou do resumo dos dados de *input*. Este capítulo debruça-se exactamente sobre a representação de tabelas em programas de COBOL.

A abordagem aqui utilizada consiste em apresentar exemplos e em tentar depois extrair técnicas mais gerais. A primeira secção relaciona-se com os dados repetitivos nos registos de *input* processados por um programa. Segue-se uma secção que vai lidar com dados tabulares construídos dentro do programa. As duas secções seguintes desenvolvem este tema mais em pormenor, conduzindo-nos a um algoritmo simples para a construção de tabelas resumindo os dados de *input*. A descrição de tabelas mais complexas é tratada com brevidade, dado que uma cobertura mais minuciosa sobre este assunto se encontra para além do âmbito deste livro. Finalmente, o estudo de caso ilustra o desenvolvimento de um programa de COBOL completo com a utilização de tabelas.

10.1 — DADOS REPETITIVOS NUM REGISTO DE «INPUT»

Suponha que existe um ficheiro de estatísticas de vendas para uma organização retalhista com várias filais, tal como uma cadeia de supermercados. Cada registo do ficheiro contém os seguintes dados:

identificativo da filial
valores de vendas em janeiro, último ano
valores de vendas em Fevereiro, último ano
. . . . .
valores de vendas em Dezembro, último ano

Um dado registo sumariza os valores de vendas mensais, para cada mês do ano comercial em curso, para uma dada filial.

Olhando para cada registo individual independentemente, o tipo de análise requerido poderia ser o valor anual de vendas, a média de vendas mensais ou a razão melhor/pior das vendas por cada filial. Números semelhantes seriam requeridos para toda a organização, pela utlização dos valores acumulados dos resultados das filiais.

Para resolver qualquer destes problemas é necessário um processo de representação de dados repetitivos, tais como as vendas mensais do exemplo acima. Fazendo uso das técnicas desenvolvidas neste livro até agora, poder-se-ia utilizar a seguinte descrição de registo:

```
reg-vendas
        filial-id           identificação da filial
        val-vendas-jan      valor de vendas em Janeiro
        val-vendas-fev      valor de vendas em Fevereiro
        .....
        val-vendas-dez      valor de vendas em Dezembro
```

Para ilustrar o uso desta aproximação, considere dois problemas simples envolvendo o processamento individual dos registos para produzir valores para uma filial. Para se encontrar o valor de vendas anual de uma determinada filial, isto envolveria a escrita de uma operação do seguinte formato:

```
calcular-vendas-anual
        calcular valor-vendas-ano =
                val-vendas-jan + val-vendas-fev +
                val-vendas-mar + . . . . . + val-vendas-dez
```

Encontrar o valor máximo de vendas mensais durante um ano envolveria uma sequência de operações como se segue:

```
encontrar-venda-maior
    mover val-vendas-jan para val-max-vendas
    if val-vendas-fev > val-max-vendas then
            mover val-vendas-fev para val-max-vendas
    endif
    . . . . . .
    if val-vendas-dez > val-max-vendas then
            mover tal-vendas-dez para val-max-vendas
    endif
```

No desenho acima, as reticências, «. . . . . », foram utilizadas para representar repetições de construções similares. Ao ser traduzida para COBOL, não poderão existir estas representações — todas as operações terão de ser exaustiva e explicitamente escritas.

Estes fragmentos de desenho poderiam ser traduzidos para COBOL, incorporados num programa que funcionasse perfeitamente. No entanto, o método é completamente desajeitado e a perspectiva de processamento de cinquenta e dois valores de vendas semanais para o ano anterior seria muito duvidosa.

Assim, requer-se uma notação que permita colectar uma série de campos similares, dando-lhes um nome e fazendo a sua descrição à medida que cada campo é retirado para ser utilizado. Em COBOL, esta estrutura de dados é chamada tabela e cada um dos campos nela definida chama-se elemento da tabela. Para a representação dos dados do exemplo acima poder-se-ia utilizar a seguinte estrutura:

```
reg-vendas
        filial-id                   identificação da filial
        val-vendas                  valor de vendas por mes
                repete-se 12 vezes
```

Os elementos têm de ser identificados por um índice. O índice deve ser um valor inteiro (número inteiro), sendo neste caso um número de um a doze, visto existirem doze elementos ne-

sta tabela. Para identificar um elemento, deve ser dado o nome da tabela seguido do índice do elemento requerido entre parênteses. Neste exemplo, os elementos são:

```
val-vendas (1)    valor de vendas de Janeiro
val-vendas (2)    valor de vendas de Fevereiro
. . . . .
val-vendas (12)   valor de vendas de Dezembro
```

O índice tem de ser uma constante; na verdade, a força das tabelas vem do facto de serem utilizados campos, que contenham números inteiros, como índices. Voltando a problemas já discutidos: se quiséssemos encontrar o valor de vendas anuais para uma determinada filial, necessitaríamos agora de escrever uma sequência de operações do tipo:

```
calcular-vendas-anual
        mover 0 para valor-vendas-ano
        mover 1 to num-mes
        until num-mes > 12 do
                calcular valor-vendas-ano =
                        val-vendas-ano = val-vendas (num-mes)
                calcular num-mes = num-mes + 1
        enduntil
```

O campo «num-mes» deve ser do tipo inteiro (sem ponto decimal) com pelo menos dois dígitos para poder conter valores até doze. O campo «val-vendas-ano» deve ser capaz de suportar um valor doze vezes maior do que o valor máximo possível para um só elemento da tabela «val-vendas».

Para ilustrar a acção deste procedimento, consideremos o grupo simplificado de valores dado na figura 10.1.

val-vendas											
1	2	3	4	5	6	7	8	9	10	11	12
3.5	6.7	4.9	8.1	5.3	6.7	7.4	8.2	2.4	3.8	5.4	6.3

Fig. 10.1

Existem dois campos temporários, «valor-vendas-ano» e «num-mes», inicializados, respectivamente, com zero e um. Assim, antes de se entrar no ciclo until, o estado destes dois campos temporários é:

```
valor-vendas-ano        num-mes

      0.0                 1
```

A condição «num-mes > 12» é falso, do que resulta a execução do corpo do ciclo. A primeira operação é:

```
calcular valor-vendas-ano =
        val-vendas-ano + val-vendas (num-mes)
```

que significa adicionar o elemento da tabela val-vendas indexado por num-mes ao valor corren-

te de val-vendas-ano e colocar o resultado em val-vendas-ano. O valor de num-mes é 1 e o valor de val-vendas (1) é 3.5; daí que 3.5 é adicionado a zero, fornecendo o valor 3.5, que é colocado em val-vendas-ano. O estado dos campos temporários é agora:

valor-vendas-ano	num-mes
3.5	1

A segunda operação é:

```
calcular num-mes = num-mes + 1
```

que adiciona 1 ao valor de num-mes. No fim do ciclo, o valor dos campos temporários é:

valor-vendas-ano	num-mes
3.5	2

Seguindo de novo o ciclo: a condição «num-mes > 12» continua falsa e o valor de val-vendas (2), que é 6.7, é adicionado a valor-vendas-ano, dando um resultado de 10.2 para este campo. O valor de num-mes é aumentado numa unidade e o estado, após a segunda execução do ciclo, é:

valor-vendas-ano	num mes
10.2	3

O leitor pode seguir as sucessivas execuções do ciclo até que eventualmente, no fim da décima primeira execução do ciclo, o estado dos campos temporários é:

valor-vendas-ano	num-mes
68.7	13

o que causa uma saída do ciclo com o campo val-vendas-ano contendo o requerido total.

Para encontrar o valor máximo de vendas na tabela é também possível construir um ciclo until para varrer a tabela, processando os elementos um de cada vez. O procedimento seguinte poderia ser utilizado para encontrar o valor máximo de vendas mensais durante o ano.

```
encontrar-venda-maior
        mover val-vendas (1) para val-max-vendas
        mover 2 para num-mes
        until num-mes > 12 do
        if val-vendas (num-mes) > val-max-vendas then
                mover val-vendas (num-mes) para val-max-vendas
        endif
                calcular val-vendas = val-vendas + 1
        enduntil
```

O campo de dados «num-mes» necessita de novo de ser um inteiro de dois dígitos e «val-max–vendas» necessita de ser um campo do mesmo tipo dos elementos da tabela «val-vendas». O leitor erá rapidamente testar o procedimento acima utilizando os valores de dados fornecidos na figura 10.1. O estado inicial dos campos de dados temporários é:

val-max-vendas	num-mes
3.5	2

e o estado final, à saída do ciclo, seria:

val-max-vendas	num-mes
8.2	13

O ciclo «for»

Relembrando as versões dos dois procedimentos «calcular-vendas-anual» e «encontrar-venda-maior», que processam os elementos da tabela val-vendas, existe uma semelhança óbvia entre as estruturas dos ciclos. Nos dois casos a sequência é:

a) Inicializar: fornecer um valor inicial a num-mes (e a outros campos, possívelmente);
b) Testar: verificar se o valor de num-mes é maior do que o valor final requerido, o que, ao verificar-se, originará a saída do ciclo;
c) Executar as operações dentro do corpo do ciclo;
d) Incrementar: adicionar 1 ao valor de num-mes;
e) Ciclo: retornar ao ponto *b*).

Neste caso, «num-mes» é chamado campo de controlo do ciclo.
Esta estrutura é tão comum, em programas envolvendo o uso de tabelas que o COBOL, tal como outras linguagens de programação, tem um tipo especial de ciclo incorporando os passos acima descritos. Na linguagem de desenho, este será chamado o ciclo for, que tem a seguinte estrutura:

```
for <campo-controlo> = <valor-inicial> to <valor-final> do
        <procedimento>
endfor
```

A acção desta construção é definida como se segue:

a) Colocar o valor de <campo-de-controlo> com o valor de <valor-inicial>;
b) Se o valor de <campo-de-controlo> for maior do que <valor-final>, então saltar para o passo *f*);
c) Executar <procedimento>;
d) Incrementar o valor de <campo-de-controlo> numa unidade;
e) Voltar ao passo *b*);
f) Continuar a executar o procedimento seguinte a endfor.

Note que, tal como o until, o ciclo for testa primeiro a condição antes de executar o corpo do ciclo. Assim, se <valor-inicial> for maior do que <valor-final>, o corpo do ciclo não será executado de maneira nenhuma.
Os dois exemplos prévios poderão ser reescritos utilizando ciclos for como se segue:

```
calcular-vendas-anual
        mover 0 para valor-vendas-ano
        for num-mes = 1 to 12 do
                calcular valor-vendas-ano =
                        val-vendas-ano + val-vendas (num-mes)
        endfor
```

```
encontrar-venda-maior
        mover val-vendas (1) para val-max-vendas
        for num-mes = 2 to 12 do
                if val-vendas (num-mes) > val-max-vendas then
                        mover val-vendas (num-mes) para val-max-vendas
                endif
        endfor
```

A estrutura geral do programa

Os fragmentos de programa acima foram desenvolvidos sem nenhuma referência a uma estrutura geral do desenho de programa. Estes procedimentos podem ser colocados dentro de um simples desenho de programa que processe todos os registos do ficheiro e imprima os totais das vendas e valor máximo de venda para cada filial.

O nível superior do desenho de programa será a estrutura familiar para processar cada registo no ficheiro:

```
sumarios-filial
        print-cabecalhos
        ler-proximo-filial
        until fim-de-filial do
                lidar-com-filial
                ler-proximo-filial
        enduntil
```

O refinamento de «print-cabecalhos» deverá imprimir os cabeçalhos necessários para as páginas do mapa de *output* e é deixado para a imaginação do leitor.

Após «lidar-com-filial», irão existir três refinamentos posteriores, dois desenvolvidos acima, para total de vendas e máximo de vendas por mês, e um procedimento para imprimir os resultados. Assim, o refinamento poderia ser:

```
lidar-com-filial
        calcular-vendas-anual
        encontrar-venda-maior
        imprimir-resultados-filial
```

Os dois procedimentos desenvolvidos acima entram neste ponto.

```
calcular-vendas-anual
        mover 0 para valor-vendas-ano
        for num-mes = 1 to 12 do
                calcular valor-vendas-ano =
                        val-vendas-ano + val-vendas (num-mes)
        endfor

encontrar-venda-maior
        mover val-vendas (1) para val-max-vendas
        for num-mes = 2 to 12 do
                if val-vendas (num-mes) > val-max-vendas then
                        mover val-vendas (num-mes) para val-max-vendas
                endif
        endfor
```

Finalmente, o procedimento «imprimir-resultados-filial» deveria imprimir uma linha sumário para a filial e poderia ser da forma:

```
imprimir-resultados-filial
        mover filial-id para sumário-filial-id
        mover val-vendas-ano para sumário-val-vendas-ano
        mover val-max-vendas para sumario-val-max-vendas
        escrever-linha-sumario
```

O exemplo acima demonstra que o uso de tabela para os dados de *input* não afecta a estrutura geral do programa, actuando somente no processamento de cada registo.

Tradução para COBOL

Quando uma tabela consiste na repetição de campos de dados simples, a sua representação em COBOL é perfeitamente linear. O formato do campo repetido é seguido da cláusula:

```
OCCURS <numero-repetiçoes> TIMES
```

onde <numero-repetiçoes> é um numero inteiro constante que define o numero de vezes que o campo deve ser repetido. Os elementos da tabela devem ser referenciados por índices que tomam valores de um até <numero-repetiçoes>.

No exemplo dos valores de vendas mensais, discutidos há mais tempo nesta secção, a descrição do registo poderia ser:

```
01      REG-VENDAS.
        05      FILIAL-ID       PIC AAAA.
        05      VAL-VENDAS      PIC 9(6)V99 OCCURS 12 TIMES.
```

Note que a cláusula *occurs* é parte da descrição do campo; assim, o ponto final que termina a descrição aparece após a cláusula.

A tradução do ciclo for para COBOL utiliza as mesmas técnicas que a tradução de um until. O corpo do ciclo transforma-se num parágrafo e é também utilizada uma variação do verbo PERFORM para implementar o controlo do ciclo. A construção do desenho era:

```
for <campo-controlo> = <valor-inicial> to <valor-final> do
        <procedimento>
endfor
```

que traduzida para COBOL é:

```
        PERFORM <nome-paragrafo>
                VARYING <campo-controlo> FROM <valor-inicial>
                BY 1 UNTIL <campo-controlo> > <valor-final>
. . . . .
<nome-paragrafo>.
        tradução do <procedimento>.
```

Por exemplo, o procedimento:

```
encontrar-venda-maior
        mover val-vendas (1) para val-max-vendas
        for num-mes = 2 to 12 do
                if val-vendas <num-mes> > val-max-vendas then
```

```
                    mover val-vendas <num-mes> para val-max-vendas
                endif
        endfor
```

poderia ser traduzido para COBOL assim:

```
ENCONTRAR-VENDA-MAIOR.
        MOVE VAL-VENDAS (1) TO VAL-MAX-VENDAS
        PERFORM COMPARAR-COM-MAXIMO
                VARYING NUM-MES FROM 2
                BY 1 UNTIL NUM-MES > 12.

COMPARAR-COM-MAXIMO.
        IF VAL-VENDAS <NUM-MES> > VAL-MAX-VENDAS
                MOVE VAL-VENDAS <NUM-MES> TO VAL-VENDAS-MAXIMO.
```

Neste exemplo, o campo de dados NUM-MES terá de ser descrito como:

```
01      CAMPOS-TEMPORARIOS.
        05      NUM-MES                 PIC 99.
```

Uma das razões mais comuns para o insucesso de programas de principiantes que utilizam tabelas com erros de execução é a descrição incorrecta dos índices. No exemplo acima, se o campo NUM–MES fosse descrito com PIC 9, o programa falharia com um erro de índice ou uma mensagem similar, que indicaria que havia sido tentada a utilização de um elemento inexistente da tabela. Isto acontece porque a sequência de valores gerados para NUM-MES será

2, 3, 4, 5, 6, 7, 8, 9, 0

É impossível representar o valor 10 num único dígito, e, assim, o valor em memória é truncado para zero. Dado que VAL-VENDAS (0) não existe, o programa não é bem sucedido. Ao descrever um campo que vai ser utilizado como índice, deve garantir-se sempre que ele é suficientemente grande para deter todos os valores de índice possíveis.

Um problema associado a este e que surge ocasionalmente deve-se ao facto de o índice ultrapassar em um valor o fim de tabela. Se uma tabela tiver vinte valores, depois da execução do ciclo for o índice irá conter o valor vinte e um. Logo, uma tabela com noventa e nove elementos necessita de um índice com três dígitos.

A opção PERFORM ... VARYING pode não ser implementada em compiladores ANS COBOL de baixo nível; no apêndice 5.5 é apresentada uma tradução alternativa do ciclo for.

10.2 — TABELAS NA «WORKING-STORAGE»

Existe um grupo de problemas que podem ser classificados como problemas de contagem de frequência. Um exemplo deste tipo de problemas, referido em muitos livros de programação, é a acumulação de resultados eleitorais. Suponha que existem dez candidatos numa eleição, numerados de um a dez. Existe um ficheiro que contém um registo para cada voto, apresentando o número do candidato votado. Os registos estarão na ordem pela qual os votos foram aparecendo, e um modo possível de abordar este ficheiro consiste em percorrer todo o ficheiro fazendo corresponder os votos a cada um dos candidatos.

Podemos visualizar este processo feito com uma pilha de boletins de voto, um lápis e um papel. O processo começa escrevendo-se os números dos dez candidatos na folha de papel. Tirando os boletins de voto um a um, lê-se o número do candidato e adiciona-se um ao total do

candidato correspondente. Depois de processados todos os boletins de voto, os totais finais de cada candidato estão representados no papel e pode proceder-se à selecção do vencedor.

Num programa, a folha de papel utilizada para acumular os totais é representada por uma tabela em armazenamento temporário. Esta tabela terá dez elementos, um para cada candidato. À medida que se processa cada um dos registos de voto, o elemento correspondente da tabela será incrementado em um de modo a apresentar outro voto para um candidato particular. No fim do programa podem imprimir-se os conteúdos da tabela para se apresentarem os resultados de voto para cada candidato. O candidato vencedor será identificado pelo maior valor da tabela.

O programa será complicado pelo facto de os dados terem de ser verificados para garantir que não se introduzem números de candidato inválidos. Uma pessoa que utilize o método equivalente do lápis e do papel reconhecerá um número de candidato inválido e poderá ignorá-lo. Se um número de candidato que não se encontre no intervalo de um a dez for lido pelo programa e se não houver uma verificação, o programa não será executado devido a um erro de execução.

Comparativamente, este problema é simples pelo facto de os candidatos serem identificados por um grupo de números contíguos. Se os candidatos fossem identificados pelo nome, o problema seria mais difícil de se resolver por um programa, apesar de a pessoa que está a utilizar lápis e papel não necessitar de alterar os seus métodos. Isto acontece porque as pessoas são muito melhores a identificar padrões do que os computadores.

No processamento de dados existem muitos problemas semelhantes. Ir-se-á agora estudar um exemplo particular com algum pormenor, para introduzir a ideia de uma tabela em armazenamento temporário e mostrar uma variedade de técnicas para utilização da mesma.

Sempre que um artigo ou conjunto de artigos é vendido por uma loja, é escrito um registo num ficheiro de acumulações. Cada registo contém:

Data da venda;
Número de caixa registadora;
Número do artigo;
Quantidade vendida;
Preço por artigo.

Estes dados poderiam ser recolhidos automaticamente por verificações no ponto de venda, como se faz nos grandes armazéns. Este ficheiro de acumulações é utilizado pelos inspectores de lojas e é chamado diário de transacções. Uma das tarefas principais dos inpectores e analisar as vendas de um artigo específico ou grupo de artigos e compará-las com os números de controlo de *stocks*. Este ficheiro alcança proporções tais que não é possível organizá-lo por número de artigo e retirar aqueles que se pretende verificar. Assim, uma solução possível é escrever um programa que vá pesquisar todo o ficheiro acumulando totais para o artigo especificado.

Na sua simplicidade, quando os artigos especificados constituírem um grupo contíguo, este problema é similar à contagem de votos numa eleição. Isto torna-se mais difícil quando existem limites nos números de artigo envolvidos e muito mais complexo se se definir um grupo de números aleatórios. O problema poderia ser estendido noutras direcções, por exemplo, acumular os valores das vendas e a quantidade vendida para um dado artigo, convencionando que o preço não seja necessariamente constante para um determinado tipo de artigo.

O problema mais simples seria escrever um programa que produza um sumário das quantidades vendidas para cada um dos artigos numerados, digamos, por exemplo, de um a vinte e cinco. Terá de existir uma tabela, chamada, por exemplo, «quantidade-total», contendo vinte e cinco elementos para acumular as quantidades. À medida que cada registo do ficheiro diário é processado, ter-se-á de verificar se o mesmo pertence ao grupo requerido. Se o número de produto se encontrar dentro dos limites um a vinte e cinco, a quantidade vendida pode ser acumulada ao elemento apropriado da tabela. Após o processamento do ficheiro diário, podem imprimir-se os resultados.

Cada elemento da tabela deve ser inicializado com zeros, da mesma forma que um campo comum utilizado para acumular o número de registos lidos.

A estrutura básica do desenho de programa requerido é:

```
acumular-totais
        inicializar-tabela
        ler-proximo-ficheiro-diario
        until fim-de-ficheiro-diario do
                actualizar-tabela
                ler-proximo-ficheiro-diario
        enduntil
        imprimir-tabela
```

A estrutura desta forma é semelhante para um programa que execute a acumulação de totais simples a partir de um dado ficheiro (ver exemplo 2.5). A única diferença é que a tabela de campos é usada para fazer a acumulação.

O primeiro refinamento é «inicializar-tabela», que deve colocar a zero os valores de todos os elementos. Isto pode ser feito utilizando um ciclo for com o seguinte formato:

```
inicializar-tabela
        for qt-indice = 1 to 25 do
                mover 0 para qt-total (qt-indice)
        endfor
```

O campo temporário «qt-indice» é associado com a tabela «qt-total» e deve ser definido por forma a conter qualquer valor correcto de índice para a dita tabela. O nome «qt-indice» foi escolhido para demonstrar o uso deste campo como índice e a sua associação a «qt-total».

Antes de se definir o refinamento de «actualizar-tabela», é necessário definir os nomes dos campos do registo de *input*. Um grupo possível de nomes é:

```
reg-diario
        diario-data-venda       data da venda
        diario-num-cx           numero de caixa registadora da venda
        diario-num-artigo       número do artigo
        diario-qt-vendida       quantidade vendida
        diario-preco-artigo     preço do artigo
```

Para a actualização do elemento da tabela correspondente ao número de artigo corrente é:

```
calcular qt-total (diario-num-artigo) =
        qt-total (diario-num-artigo) + diario-qt-vendida
```

No entanto, somente devem ser acumuladas na tabela as quantidades vendidas para artigos de número entre um e vinte e cinco, inclusive. Assim, o refinamento poderia ser:

```
actualizar-tabela
        if diario-num-artigo > 0 and diario-num-artigo < 26 then
                calcular qt-total (diario-num-artigo) =
                        qt-total (diario-num-artigo) + diario-qt-vendida
        endif
```

Após processar todos os registos do ficheiro diário, o programa necessita de imprimir os conteúdos da tabela. Suponha que o registo de output é:

```
linha-sumário
        sumario-num-artigo
        sumario-qt-total
```

Então, para listar os conteúdos da tabela, somente se necessita de um ciclo for com o seguinte formato:

```
imprimir-tabela
        for qt-indice = 1 to 25 do
                mover qt-indice para sumario-num-artigo
                mover qt-total (qt-indice)
                        para sumario-qt-total
                escrever linha-sumario
        endfor
```

Existem dois pontos principais a realçar acerca deste refinamento. Primeiramente, os números de artigo para a linha de sumário estão a ser gerados pelo ciclo for pela sua correspondência com os índices da tabela, mas atenção a que os mesmos não se encontram armazenados na tabela. Em segundo lugar, como se requer uma linha por cada elemento da tabela, «escrever-linha-sumario» deve estar dentro do ciclo for.

Se o leitor continua com alguma dúvida cerca da operacionalidade deste programa, aconselha-se então que seja feito um teste manual.

O tamanho da tabela não é importante; assim, para se executar um teste manual menos aborrecido, sugere-se que substitua 25 por 5 (e 26 por 6), no desenho acima. Desenhe agora uma tabela só de cinco elementos, como se mostra na figura 10.2.

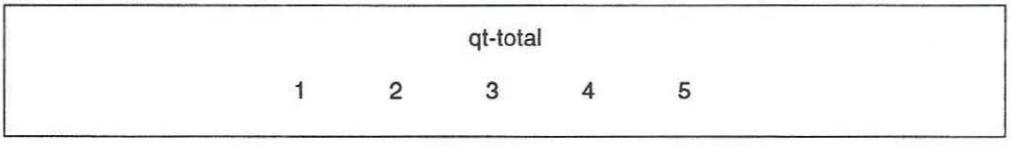


Fig. 10.2

Agora trabalhe o desenho de programa cumprindo as instruções, utilizando os seguintes dados de *input*:

Diário-num-artigo	Diário-qt-vendida
2	3
36	16
3	12
2	4
0	5
4	26
1	2
3	24

O *output* do programa seria:

1	2
2	7
3	36
4	26
5	0

Implementação em COBOL

A única diferença entre a implementação deste programa e a do dado em 9.1 é que «qt-total» necessita de ser descrito na WORKING-STORAGE SECTION do programa.

Assim, no programa de COBOL correspondente apareceria o seguinte:

```
WORKING-STORAGE  SECTION.
01      VECTOR-ESTADO.
        05      FIM-FICH-DIARIO          PIC X.
                88  FIM-DE-FICH-DIARIO                 VALUE «F».
01      TAB-QT-TOTAL.
        05      QT-TOTAL                 PIC 9(6)       OCCURS 25 TIMES.
        05      QT-INDICE                PIC 99.
```

O nome de campo para o índice é somente um nome de campo normal, mas deve ser escolhido de modo a realçar o uso do mesmo. Se o campo para o índice é descrito imediatamente após os elementos da tabela, e na mesma estrutura de dados, isto irá ajudar alguém que esteja porventura a ler o programa a associar o índice com a tabela.

Existe uma regra em COBOL que diz que uma cláusula *occurs* não deve estar associada com um campo de nível 01.

Transformações mais complexas

Suponha que, apesar do facto de os números de artigo estarem no limite de 1 a 25, os inspectores ou fiscais quiseram acumular os totais para cada um dos números de 1016 a 1040. Isto, aparentemente, traria um problema diferente do resolvido acima, mas repare que continuam a ser na mesma vinte e cinco números de artigo diferentes. O que se requer é que sejam acumuladas as quantidades para o número de artigo 1016 para o primeiro elemento da tabela, 1017 para o segundo, etc., sendo o elemento 1040 associado com o vigésimo quinto elemento. Isto significa que os números de artigo devem ser dispostos no mapa do seu próprio limite para o limite de 1 a 25. A execução deste mapa é extremamente simples — subtraia 1015 ao número do artigo para ter a correspondente chave na tabela.

A estrutura geral do programa é a mesma da desenvolvida anteriormente nesta secção. Existem só dois refinamentos que são afectados por esta alteração na especificação do problema: «actulizar-tabela» e «imprimir-tabela». A primeira alteração é no sentido de se retirar o valor correcto do limite de valores dos artigos e listá-los segundo os índices da tabela. A versão revista é:

```
actualizar-tabela
        if diario-num-artigo > 1015 and diario-num-artigo < 1041 then
                calcular qt-indice = qt-indice — 1015
                calcular qt-total (diario-num-artigo) =
                        qt-total (diario-num-artigo) + diario-qt-vendida
        endif
```

Para a impressão de um mapa sumário de *output* inteligível, os índices devem ser escritos com os valores reais dos números de artigo. Isto pode ser executado pela simples adição de 1015 ao valor do índice. O refinamento revisto de «imprimir-tabela» poderia ser:

```
imprimir-tabela
        for qt-indice = 1 to 25 do
                calcular sumario-num-artigo = qt-indice + 1015
                mover qt-total (qt-indice) para sumario-qt-total
                escrever linha-sumario
        endfor
```

Existem então três alterações requeridas no desenho de programa: a verificação do limite das chaves, a transposição do número de artigo para índice e do índice para o número de artigo.

O desenho básico pode ser utilizado para resolver qualquer problema onde seja possível transpor do número de artigo para um índice de tabelas. Suponha que o sistema de atribuição de números aos *stocks* está desenhado por forma que artigos similares estejam agrupados. Os inspectores estão mais interessados em verificar as quantidades vendidas dentro de cada grupo do que em artigos individuais. Como exemplo, considere o problema da acumulação de quantidades por cada um dos seguintes grupos de números de artigo: 2000-2009, 2010-2019, 2020-2029, ... 2190-2199. Existem vinte grupos de números de artigo; então, necessitamos de uma tabela de vinte elementos e da seguinte definição:

Diário-num-artigo	Índice
2000 - 2009	1
2010 - 2019	2
2020 - 2029	3
. . . . .	
2190 - 2199	20

Se o último dígito de diario-num-artigo for ignorado, existem vinte números entre 200 e 219. Para atribuir esses valores truncados a uma tabela de vinte elementos, simplesmente subtrair-lhes-íamos 199.

Para ignorar o último dígito de diario-num-artigo, utilizaríamos um truque simples na descrição deste campo. A descrição de diario-num-artigo poderia ser:

```
05      DIARIO-NUM-ARTIGO.
        10      DIARIO-NUM-ARTIGO-1-3       PIC 999.
        10      DIARIO-NUM-ARTIGO-4         PIC 9.
```

A transposição para valor de índice de diario-num-artigo é do seguinte formato:

```
calcular qt-indice = diario-num-artigo-1-3 — 199
```

Uma técnica geralmente mais aplicável e que não depende da estruturação de dados em COBOL é dividir o diario-num-artigo por 10 e retirar a parte fraccionária. Para fazer isto em COBOL, poder-se-ia definir um campo temporário com um dígito a menos do que diario-num-artigo. Por exemplo, supondo que diario-num-artigo tem quatro dígitos:

```
01      CAMPOS-TEMPORARIOS.
        05      TRUNC-NUM-ARTIGO            PIC 999.
```

Então, a operação:

```
calcular trunc-num-artigo = diario-num-artigo / 10
```

retira o último dígito de diario-num-artigo. A completa transposição seria:

```
calcular trunc-num-artigo = diario-num-artigo / 10
calcular qt-indice = trunc-num-artigo — 199
```

Esta transposição pode ser incluída em «actualizar-tabela» fornecendo o seguinte refinamento alterado:

```
actualizar-tabela
        if diario-num-artigo > 1999 and diario-num-artigo < 2200 then
                calcular trunc-num-artigo = diario-num-artigo / 10
                calcular qt-indice = trunc-num-artigo — 199
                calcular qt-total (diario-num-artigo) =
                        qt-total (diario-num-artigo) + (diario-qt-vendida
        endif
```

Deve agora executar-se uma transposição do índice para número de artigo de modo a produzir um *output* legível. Como o programa está então a lidar com limites de valores, seria sensível produzir a linha-sumario como se segue:

```
linha-sumario
        sumario-num-baixo-artigo  número de artigo mais baixo
        sumario-num-alto-artigo   número de artigo mais alto
        sumario-qt-total          quantidade total vendida
                                          para aquele limite de valores
```

Para transformar o valor de índice no número de artigo mais baixo do limite, a operação poderia ser:

```
calcular sumario-num-baixo-artigo = (qt-indice + 199) * 10
```

e para o valor mais alto:

```
calcular sumario-num-alto-artigo = (qt-indice + 199) * 10 + 9
```

O refinamento revisto para imprimir-tabela seria então:

```
imprimir-tabela
        for qt-indice = 1 to 25 do
                calcular sumario-num-baixo-artigo = (qt-indice + 199) * 10
                calcular sumario-num-alto-artigo = (qt-indice + 199) * 10 + 9
                mover qt-total (qt-indice) para sumario-qt-total
                escrever linha-sumario
        endfor
```

Não obstante ser possível estender este estudo à utilização de algoritmos de transposição cada vez mais complexos, não é realmente muito instrutivo fazê-lo. Uma solução mais geral do problema é dada na secção 10.4.

10.3 — ITENS DE GRUPO REPETIDOS

Nos exemplos discutidos nas secções 10.1 e 10.2 convencionou-se que uma tabela consiste num número de repetições de um item elementar. É também possível haver dois itens de grupo repetidos como parte de uma estrutura de dados.

Suponha que o *input* de um programa é composto por registos contendo as informações de uma folha de presenças de um empregado numa semana. Cada registo pode conter: número de empregado, secção do empregado, seis conjuntos de dados diários para tempo normal de trabalho e horas extraordinárias e taxas de pagamento para os dois casos. Esta estrutura de registo pode ser descrita do seguinte modo:

```
reg-empregado
        emp-numero                      número de empregado
        emp-secção                      código da secção do empregado
        emp-horas                       horas de trabalho/dia
                        repete-se seis vezes
                emp-hora-normal         horas trabalho/normal
                emp-horas-extras        horas trabalho/extraordinário
        emp-taxa-normal                 taxa normal de pagamento
        emp-taxa-extras                 taxa extraordinária de pagamento
```

Existem seis elementos «emp-horas» na tabela que podem ser identificados através do uso de um índice, de modo que nos seja possível fazer uma referência a emp-horas (1), emp-horas (num-dia), etc. No entanto, cada elemento da tabela encontra-se dividido em «emp-hora-normal» e «emp-hora–extras». Se a referência for feita a qualquer um destes itens elementares, o nome tem de fazer parte do índice.

A adição do número de horas «normal» que um empregado trabalhou durante uma semana requer um ciclo for com o seguinte formato:

```
mover 0 para total-hora-normal
for num-dia = 1 to 6 do
        calcular total-hora-normal =
                total-hora-normal + emp-hora-normal (num-dia)
endfor
```

Na figura 10.3 demonstra-se como pode ser visualizada a estrutura do registo:

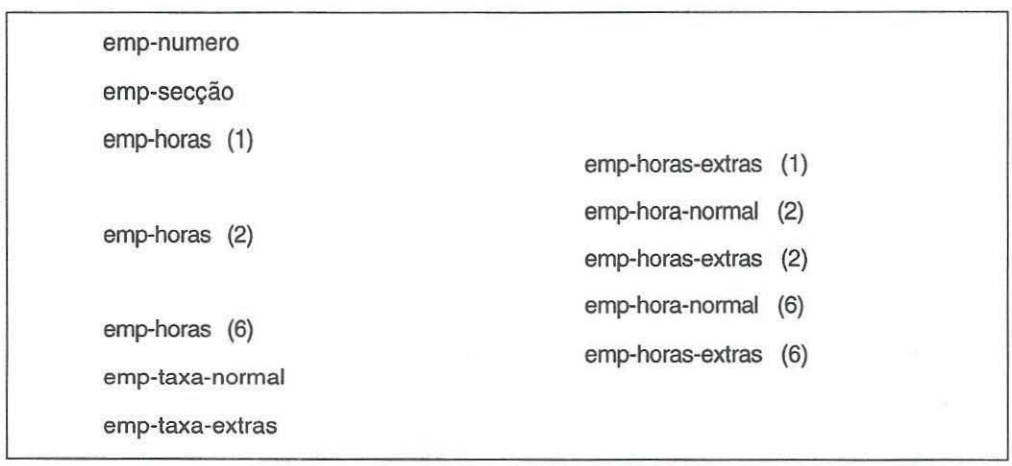


Fig. 10.3

A descrição deste registo em COBOL é extremamente simples. A cláusula *occurs,* que indica repetição, é associada ao nome do item de grupo; todos os itens subordinados ao grupo serão implicitamente repetidos. Uma possível descrição do registo de empregado é:

```
01      REG-EMPREGADO.
        05      EMP-NUMERO              PIC 9(6).
        05      EMP-SECCAO              PIC AA.
        05      EMP-HORAS           OCCURS 6 TIMES.
```

```
        10      EMP-HORA-NORMAL     PIC 9.
        10      EMP-HORA-EXTRAS     PIC 99.
05      EMP-TAXA-EXTRAS             PIC 9V99.
```

O item de grupo a repetir pode ter mais de um nível de campos subordinados. No entanto, se existirem muitos níveis envolvidos na estrutura, que pertençam ao item de grupo repetido, estes devem ser indexados quando referidos. Por exemplo:

```
reg-seguro
        seguro-nome                     nome do segurado (detentor da apólice)
        seguro-apolice                  dados de cada apólice
                repete-se dez vezes
                seguro-num-apolice      número da apólice
                seguro-venc-apolice     data de pagamento da apólice
                        seguro-venc-mes mês de pagamento da apólice
                        seguro-venc-ano ano de pagamento da apólice
                seguro-qt-apolice       valor do seguro
        seguro-idade                    idade do segurado
```

Para determinar o ano em que a terceira apólice se vence, será necessário uma referência a seguro-venc-ano (3). Note que uma tabela pode consistir apenas num número fixo de elementos. Assim, se alguns dos elementos da tabela «seguro-apolice» não forem utilizados, os mesmos terão de ser preenchidos com zeros ou brancos. Da mesma forma, se um segurado tiver mais de dez apólices, será requerido um registo de continuação, ou o tamanho da tabela terá de ser alterado de modo a comportar todos os registos.

É possível ter-se um item repetido dentro de um item de grupo repetido; isto será analisado na secção 10.6.

Um item de grupo repetido pode também ser utilizado para acumular dados. No exemplo principal apreciado na secção 10.2, somente as quantidades vendidas eram acumuladas. Poderia ser mais realista acumular-se a quantidade vendida e o valor de vendas para cada um dos limites de número de artigo especificados. Para executar isto, poder-se-ia definir na WORKING-STORAGE SECTION do programa a seguinte tabela:

```
totais-por-artigo
                repete-se <limite-definido> vezes
        qt-total
valor-total
```

O desenho de programa desenvolvido no fim da secção 10.2 poderia ser aumentado de forma a incluir a acumulação dos valores de vendas. Os dados de *input* são:

```
reg-diario
        diario-data-venda       data da venda
        diario-num-cx           número de caixa registadora da venda
        diario-num-artigo       número do artigo
        diario-qt-vendida       quantidade vendida
        diario-preço-artigo             preço do artigo
```

Para executar o *output* do valor do total das vendas, seria necessário adicionar um campo extra ao registo de output, «linha-sumario», assim:

```
linha-sumario
        sumario-num-baixo-artigo  número de artigo mais baixo
        sumario-num-alto-artigo   número de artigo mais alto
        sumario-qt-total          quantidade total vendida
```

	para aquele limite de valores
sumario-valor-total	total de vendas
	para o limite de valores

O desenho de programa revisto para incluir a totalização de valores é dado abaixo. O nome do índice para a tabela foi alterado para «tpa-indice» pelo facto de ele se referir aos elementos da tabela «totais-por-artigo».

```
acumular-totais
        inicializar-tabela
        ler-proximo-ficheiro-diario
        until fim-de-ficheiro-diario do
                actualizar-tabela
                ler-proximo-ficheiro-diario
        enduntil
        imprimir-tabela

inicializar-tabela
        for qt-indice = 1 to 25 do
                mover 0 para qt-total (tpa-indice)
                mover 0 para valor-total (tpa-indice)
        endfor

actualizar-tabela
        if diario-num-artigo > 1999 and diario-num-artigo < 2200 then
                calcular trunc-num-artigo = diario-num-artigo / 10
                calcular qt-indice = trunc-num-artigo — 199
                calcular qt-total (tpa-indice) = qt-total (tpa-indice) + diario-qt-vendida
                calcular valor-total (tpa-indice) =
                        valor-total (tpa-indice)
                        + diario-qt-vendida * diario-preço-artigo
        endif

imprimir-tabela
        for tpa-indice = 1 to 25 do
                calcular sumario-num-baixo-artigo = (tpa-indice + 199) * 10
                calcular sumario-num-alto-artigo = (tpa-indice + 199) * 10 + 9
                mover qt-total (tpa-indice) para sumario-qt-total
                mover valor-total (tpa-indice) para sumario-valor-total
                escrever linha-sumario
        endfor
```

A descrição da tabela no correspondente programa de COBOL seria agora:

```
WORKING-STORAGE  SECTION.
01      VECTOR-ESTADO.
        05      FIM-FICH-DIARIO                 PIC X.
                88      FIM-DE-FICH-DIARIO              VALUE «F».

01      TAB-TOTAIS-POR-ARTIGO.
        05      TOTAIS-POR-ARTIGO    OCCURS 20 TIMES.
                10      QT-TOTAL                PIC 9(6).
                10      VALOR-TOTAL             PIC 9(8)V99.
        05      TPA-INDICE                      PIC 99.
```

10.4 — PESQUISANDO UMA TABELA

Os algoritmos delineados na secção 10.2 lidaram com um conjunto fixo de elementos que podem ser úteis para certas circunstâncias, mas, em geral, serão muito limitativas.

A solução mais generalizada consiste em ler os elementos necessários a partir de um ficheiro separado e armazená-los numa tabela. Cada elemento desta tabela deve ser um item de grupo composto por itens elementares para conter um artigo, a quantidade a ele associada e o seu valor de vendas. O ficheiro diário é então processado e o número de artigo em cada registo é comparado com os números de artigo da tabela. Se o número de artigo do registo condiz com o número de artigo da tabela, os campos associados para a quantidade e valor de vendas são actualizados pelOs valores do registo.

A estrutura geral do programa é similar à utilizada ao longo das secções 10.2 e 10.3 e é dada abaixo.

```
pesquisar-tabela
        inicializar-tabela
        ler-proximo-ficheiro-diario
        until fim-de-ficheiro-diario do
                actualizar-tabela
                ler-proximo-ficheiro-diario
        enduntil
        imprimir-tabela
```

No entanto, os refinamentos de «inicializar-tabela» e «actualizar-tabela» são agora mais complexos.

A tabela para conter os números de artigo referidos e as quantidades e valores a si associados poderia ser descrita da seguinte forma:

```
pesquisar-artigo
                repete-se <tamanho-tabela> vezes
        pesquisa-num-artigo
        pesquisa-qt
        pesquisa-valor
```

O número de elementos da tabela, fornecido por <tamanho-tabela>, define o número máximo de números de artigo requeridos que podem ser especificados , mas não é necessário que a tabela seja preenchida com números de artigo. Para indexar a tabela, pode ser definido o campo de dados «pt-indice» e, para indicar a posição do último número de artigo na tabela, pode ser definido um campo adicional chamado «pt-limite». Isto significa que em qualquer altura os únicos elementos que contêm informação são «pesquisar-artigo (1)» até «pesquisar-artigo (pt-limite)», inclusive.

Se os campos de dados requeridos forem lidos de um ficheiro chamado «fich-req», no qual todos os registos contêm um campo chamado «req-num-artigo», então a primeira tentativa de refinamento para actualizar-tabela» seria:

```
inicializar-tabela
        mover 0 para pt-indice
        ler-proximo-ficheiro-req
        until fim-de-ficheiro-req do
                calcular pt-indice = pt-indice + 1
                mover req-num-artigo para pesquisa-num-artigo (pt-indice)
                mover 0 para pesquisa-qt (pt-indice)
                mover 0 para pesquisa-valor (pt-indice)
```

```
                ler-proximo-ficheiro-req
        enduntil
        mover pt-indice para pt-limite
```

Este bocado de programa funciona ao longo de «fich-req», inicializando para cada registo um elemento da tabela «pesquisar-artigo» com o valor do número de artigo requerido, colocando a zeros os campos de dados associados. Quando o ficheiro dos números de artigo é esgotado, o campo de dados «pt-limite» é colocado com o valor de índice corrente para mostrar quantos números de artigo já foram armazenados na tabela. O campo de dados «pt-limite» será então tratado como uma constante ao longo do resto do programa.

O refinamento mostrado acima irá trabalhar o ficheiro correctamente até que alguém inadvertidamente coloque mais números de artigo em «fich-req» do que os que podem ser armazenados na tabela. O programa irá então falhar com um erro de índice quando o valor de «pt-indice» exceder o de <tamanho-tabela>. É possível deixar o utilizador com um erro de sistema, mas é de bom-tom prever o erro dentro do programa, fornecer ao utilizador uma mensagem de erro apropriada e tomar a acção apropriada.

Neste caso, existem dois caminhos alternativos óbvios: ou ignorar quaisquer números de artigo que não possam ser armazenados na tabela, ou terminar o programa, sem tentar fazer qualquer acumulação. A última opção será, porventura, a mais sensata, visto que se o utilizador especifica, por exemplo, cinquenta números de artigo de um total de trinta possíveis de armazenar na tabela, isto significa que a tabela é pequena para as suas necessidades.

Assim, se o valor de «pt-indice» exceder o de <tamanho-tabela>, deve ser visualizada uma mensagem e a consequente paragem do programa. Uma forma de executar isto é mostrada abaixo.

```
inicializar-tabela
        mover 0 para pt-indice
        ler-proximo-ficheiro-req
        until fim-de-ficheiro-req do
                calcular pt-indice = pt-indice + 1
                verficar-excesso-tabela
                mover req-num-artigo para pesquisa-num-artigo (pt-indice)
                mover 0 para pesquisa-qt (pt-indice)
                mover 0 para pesquisa-valor (pt-indice)
                ler-proximo-ficheiro-req
        enduntil
        mover pt-indice para pt-limite

verificar-excesso-tabela
        if pt-indice > <tamanho-tabela> then
                display «especificados demasiados artigos»
                display «maximo possivel <tamanho-tabela>»
                fecho
        endif
```

A referência a «fecho» origina a execução do procedimento de fim *standard* do programa e o programa pára porque existe em «fecho» uma instrução de STOP RUN. Se «inicializar-tabela» for executado com êxito, a tabela «pesquisar-artigo» deverá ser propriamente inicializada com valores armazenados no primeiro e seguintes elementos até ao elemento de índice «pt-limite».

Para actualizar a tabela é necessário criar um ciclo de pesquisa que vá comparando o valor de «diario-num-artigo» com o de «pesquisa-num-artigo». Se houver correspondência, os respectivos valores de quantidade e valor devem ser actualizados. Uma primeira abordagem ao refinamento «actualizar-tabela» poderia ser:

```
actualizar-tabela
        for pt-indice = 1 to pt-limite do
                if diario-num-artigo = pesquisa-num-artigo (pt-indice) then
                        acumular-dados
                endif
        endfor

acumular-dados
        calcular pesquisa-qt (pt-indice) =
                pesquisa-qt (pt-indice) + diario-qt-vendida
        calcular pesquisa-valor (pt-indice) =
                pesquisa-valor (pt-indice) + diario-qt-vendida * diario-preço-artigo
```

Esta solução não prevê o facto de, se foi encontrada correspondência na tabela para um número de artigo, já não ser necessário continuar a mesma. Um algoritmo ligeiramente mais eficiente poderia ser:

```
actualizar-tabela
        mover falso para fim-pesquisa
        mover 1 para pt-indice
        until fim-pesquisa do
                if diario-num-artigo = pesquisa-num-artigo (pt-indice) then
                        acumular-dados
                        mover verdadeiro para fim-pesquisa
                else
                        calcular pt-indice = pt-indice + 1
                        if pt-indice > pt-limite then
                                mover verdadeiro para fim-pesquisa
                        endif
                endif
        enduntil
```

O campo booliano fim-pesquisa é utilizado para controlar o ciclo de pesquisa. Inicialmente é colocado em falso e a pesquisa continua até que haja correspondência ou que esta situação não seja possível, quando «pt-indice» ultrapassar «pt-limite».

Para completar a descrição do programa, pode ser adicionado o refinamento final «imprimir-tabela». O registo de *output* pode ser definido como se segue:

```
linha-sumario
        sumario-num-artigo      número de artigo requerido
        sumario-qt-total        quantidade total vendida
        sumario-val-total       valor total de vendas de artigo
```

O refinamento é um ciclo for bastante linear:

```
imprimir-tabela
        for pt-indice = 1 to pt-limite do
                mover pesquisa-num-artigo (pt-indice) para sumario-num-artigo
                mover pesquisa-qt (pt-indice) para sumario-qt-total
                mover pesquisa-valor (pt-indice) para sumario-val-total
                escrever linha-sumario
        endfor
```

Implementação em COBOL

Para implementar o algoritmo acima para COBOL é necessário uma tabela contendo um item de grupo repetido, tal como o descrito na secção 10.4, e dois ficheiros de *input*, um para os números de artigo requeridos e outro para o ficheiro diário. A DATA DIVISION para o programa correspondente poderia ser:

```
DATA DIVISION.
FILE SECTION.
FD FICH-REQ
        LABEL ECORDS OMITTED.
01      REG-REQ.
        05      REQ-NUM-ARTIGO          PIC 9(8).

FD      FICH-DIARIO
        LABEL RECORDS STANDARD.
01      REG-DIARIO.
        05      DIARIO-DATA-VENDA       PIC 9(8).
        05      DIARIO-NUM-CX           PIC 9(4).
        05      DIARIO-NUM-ARTIGO       PIC 9(8).
        05      DIARIO-QT-VENDIDA       PIC 9(4).
        05      DIARI0-PRECO-ARTIGO     PIC 999V99.

FD      FICHOUT
        LABEL RECORDS OMITTED.
01      LINHA-OUT                       PIC X(120).

WORKING-STORAGE SECTION.
01      VECTOR-ESTADO.
        05      FIM-FICH-DIARIO                 PIC X.
                88      FIM-DE-FICH-DIARIO              VALUE «F».
        05      FIM-FICH-REG                    PIC X.
                88      FIM-DE-FICH-REQ                 VALUE «F».

01      TAB-PESQUISAR-ARTIGO.
        05      PESQUISAR-ARTIGO    OCCURS 30 TIMES.
                10      PESQUISA-NUM-ARTIGO PIC 9(8).
                10      PESQUISA-QT         PIC 9(6).
                10      PESQUISA-VALOR      PIC 9(6)V99.
        05      PT-INDICE                   PIC 99.
        05      PT-LIMITE                   PIC 99.

01  LINHA-SUMARIO.
        05      SUMARIO-NUM-ARTIGO          PIC 9(8).
        05      FILLER                      PIC X(4).
        05      SUMARIO-QT-TOTAL            PIC Z(5)9.
        05      FILLER PIC X(4).
        05      SUMARIO-VAL-TOTAL           PIC £££££,££9.99.
```

O tamanho da tabela deve ser escolhido de modo a exceder o número de elementos que se espera que sejam definidos pelo utilizador do programa. No exemplo acima escolheu-se arbitrariamente o valor 30.

Um algotiymo de pesquisa geral da tabela

A ideia de pesquisar uma tabela foi introduzida nesta secção através da utilização de um exemplo introduzido na secção 10.2 e tem sofrido várias alterações desde então. É possível remover as partes do desenho de programa específicas para aquele programa e identificar algumas ramificações para um algoritmo que inicialize e pesquise uma tabela com valores de chaves.

Este algoritmo irá trabalhar com qualquer tipo de chave: alfabética, numérica ou alfanumérica, não obstante alguns dos algoritmos já estudados que dependem de uma transformação de valores numéricos em índices. A forma geral do programa requer a utilização de dois ficheiros de *input:* «ficheiro–req», lista dos valores de chave requeridos, e «ficheiro-dados», contendo os dados «em bruto» para serem analisados. Admitir-se-á que estes dois ficheiros terão a seguinte estrutura de registo:

```
reg-req
        req-chave        valor de chave requerida

reg-dados
        dados-chave      valor de chave do mesmo formato de req-chave
        dados-detalhes   outros campos de dados contendo valores
```

Para executar o *output* dos valores acumulados, suporemos que existe um ficheiro chamado «fich–sumario» que contém registos do seguinte formato:

```
sumario-reg
        sumario-chave          valor de chave requerida
        sumario-detalhes       campo de output para valores acumulados
```

Por forma a acumular os valores de «ficheiro-dados», deverá existir na WORKING-STORAGE SECTION do programa a seguinte tabela:

```
armazem-dados
                repete-se <tamanho-tabela> vezes
        armazem-chave          valor de chave (a partir de ficheiro-req)
        armazem-detalhes               campos de dados para acumulação
```

Para aceder à tabela iremos necessitar de um campo de dados «da-indice» e de um para conter o número máximo de elementos, chamado «da-limite».

A forma geral do algoritmo é dada abaixo. A nova actualização da tabela depende do número e tipo de valores diferentes a serem acumulados. A produção do *output* irá similarmente ser afectada pela estrutura da tabela e é por isso deixada incompleta. Os pontos que são problemas específicos são colocados entre «<» e«>».

```
pesquisar-tabela
        inicializar-tabela
        ler-proximo-ficheiro-dados
        until fim-de-ficheiro-dados do
                actualizar-tabela
                ler-proximo-ficheiro-dados
        enduntil
        imprimir-tabela

inicializar-tabela
        mover 0 para da-indice
        ler-proximo-ficheiro-req
```

```
        until fim-de-ficheiro-req do
                calcular da-indice = da-indice + 1
                verificar-excesso-tabela
                mover req-chave para armazém-chave (da-indice)
                <inicializar-armazem-detalhes>
                ler-proximo-ficheiro-req
        enduntil
        mover da-indice para da-limite

verificar-excesso-tabela
        if da-indice > <tamanho-tabela> then
                <escrever mensagem para utilizador>
                fecho
        endif

actualizar-tabela
        mover falso para fim-pesquisa
        mover 1 para da-indice
        until fim-pesquisa do
                if chaves-dados = armazem-chave (da-indice) then
                        <actualizar dados armazenados>
                        mover verdadeiro para fim-pesquisa
                else
                        calcular da-indice = da-indice + 1
                        if da-indice > da-limite then
                                mover verdadeiro para fim-pesquisa
                        endif
                endif
        enduntil

imprimir-tabela
        imprimir-cabeçalhos
        for da-indice = 1 to da-limite do
                mover armazem-chave <da-indice> para sumario-chave
                <mover armazem-detalhes para sumario-detalhes>
                escrever linha-sumario
        endfor

imprimir-cabeçalhos
        <imprimir cabeçalhos para o output>
```

10.5 — TABELAS MULTIDIMENSIONAIS

Até agora, as tabelas analisadas foram «unidimensionais», que significa um elemento, um item de grupo, repetido um número de vezes. É possível existirem tabelas de duas ou mais dimensões, «tabelas multidimensionais», que têm campos repetidos entre itens de grupo, por sua vez também repetidos, etc. Neste livro, os principais exemplos desenvolvidos (e respectivos exercícios) baseiam-se em tabelas a uma dimensão, mas iremos fazer uma breve introdução a tabelas multidimensionais que o leitor poderá experimentar utilizar, caso o deseje.

Como exemplo de uma tabela multidimensional, considere a seguinte estrutura:

```
tabela-filial
        filial-vendas       vendas anuais da filial
                repete-se 30 vezes
        filial-codigo       identificador da filial
        filial-val-mes      valor de vendas para um mes
                repete-se 12 vezes
```

O elemento da tabela filial-vendas repete-se trinta vezes e dentro de cada um destes elementos existe uma ocorrência de «filial-codigo» e doze ocorrências de «filial-val-mes». Para identificar uma ocorrência particular de «filial-vendas», o nome deve ser associado a um índice, por exemplo: filial–vendas (25), filial-vendas (fv-indice). Para aceder ao «filial-codigo» correspondente, este deve ser também relacionado com um índice, por exemplo: filial-codigo (25), filial-codigo (fv-indice).

Dentro de cada ocorrência do item de grupo repetido filial-vendas existem doze ocorrências de «filial-val-mes». Assim, para identificar cada um deles, irão ser necessários dois índices, um para identificar qual «filial-vendas» é requerido e outro para identificar qual dos elementos «filial-val-mes» dentro deste item de grupo é também requerido. Estes índices são escritos pela ordem em que as cláusulas *occurs* aparecem na estrutura de dados. Portanto, uma referência a

```
filial-val-mes (fv-indice, 3)
```

significa o terceiro elemento filial-val-mes dentro do elemento filial-vendas de índice fv-indice. Alternativamente: encontre o elemento filial-vendas de índice fv-indice e, dentro deste elemento, também a terceira ocorrência de «filial-val-mes».

A estrutura da tabela poderia ser visualizada como na figura 10.4.

A descrição desta tabela num program de COBOL poderia ser da seguinte forma:

```
01      TAB-FILIAL.
        05      FILIAL-VENDAS           OCCURS 30 TIMES.
                10      FILIAL-CODIGO           PIC XXX.
                10      FILIAL-VAL-MES          PIC 9(5)V99
                        OCCURS 12 TIMES.
```

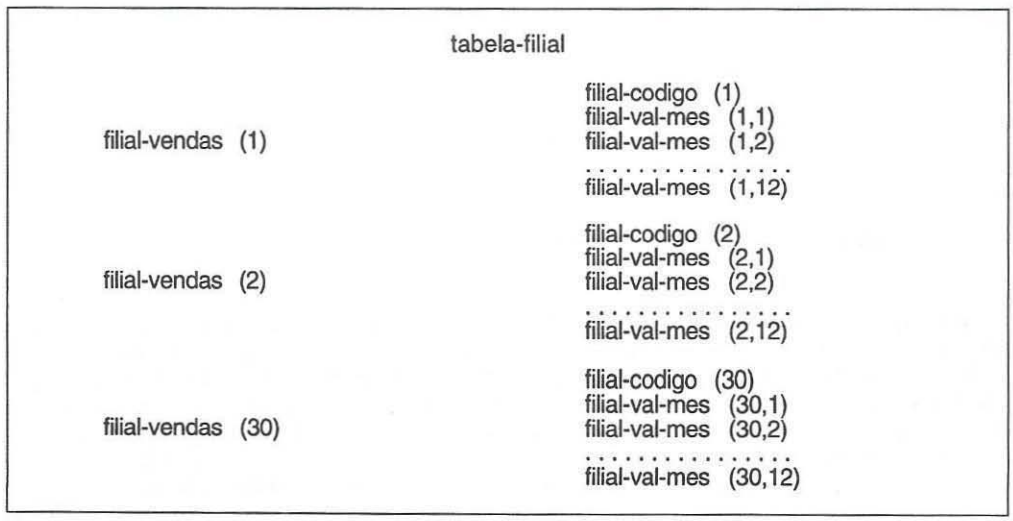


Fig. 10.4

A figura 10.5 mostra um exemplo fictício do tipo de tabelas apresentado com muita frequência pelas companhias de seguros, que mostram a relação entre prémios pagos e o valor do vencimento da apólice, caso os bónus continuem a manter o nível presente.

	Garantia Vida Cober. £5,000		Garantia Vida Cober. £10,000		Garantia Vida Cober. £20,000	
Idade	Prémio mensal	Retorno esperado	Prémio mensal	Retorno esperado	Prémio mensal	Retorno esperado
18 2.85	8,672	4.82	18,545	9.21	39,128	
19 2.96	8,593	4.97	18,427	9.43	38,974	
. . . .	. . .	. . .	. . .	. . .	. . .	
. . . .	. . .	. . .	. . .	. . .	. . .	
50 5.22	5,825	10.13	11,742	20.11	23,818	

Fig. 10.5

Para a representação da informação contida no corpo desta tabela dentro de um programa de COBOL seria necessário uma tabela bidimensional com o seguinte formato:

```
tabela-plano
        plano-linha                         linha para um valor de idade
                        repete-se 33 vezes
                plan-l-idade                idade do proponente
                plan-l-coluna               coluna dos prémios e retornos
                        repete-se 3 vezes
                        plan-l-c-premio     prémio mensal
                        plan-l-c-retorno    retorno esperado
```

O prémio mensal a pagar por uma pessoa de 19 anos para um seguro de vida de £10,000 seria então fornecido por «plan-l-c-premio (2,2)» e o retorno esperado para um requerente de 30 anos de idade e um seguro de £20,000 seria dado por «plan-l-c-retorno (13,3)».

A descrição da tabela correspondente seria:

```
01      TABELA-PLANO.
        05      PLANO-LINHA             OCCURS 33 TIMES.
                10      PLAN-L-IDADE                    PIC 99.
                10      PLAN-L-COLUNA           OCCURS 3 TIMES.
                        15      PLAN-L-C-PREMIO         PIC 99V99.
                        15      PLAN-L-C-RETORNO        PIC 9(5).
```

10.6 — **ESTUDO DE CASO**

Este estudo de caso pretende ilustrar o uso de tabelas, dando um exemplo ligeiramente diferente no que diz respeito à pesquisa da tabela estudada na secção 10.4. A especificação do problema é a seguinte:

O sistema de computador de uma firma comerciante de automóveis, a Honest Joe (Automóveis), Lda., inclui um ficheiro que contém informação sobre todos os veículos vendidos durante o mês anterior. Fazendo parte dessa informação estão o código de modelo do veículo vendido, o valor total da venda e o lucro obtido.

O formato simplificado do registo de *input* é:

```
Registo de venda de veículos
        Código de modelo          3 caracteres
        Valor da venda            7 dígitos (pénis)
        Lucro obtido              6 dígitos (pénis, com sinal)
```

O código de modelo consiste numa mistura de caracteres numéricos e alfabéticos, existindo menos de cinquenta em uso. O ficheiro de vendas de veículos está ordenado por número de factura.

Pretende-se um programa que produza um sumário do número de modelos de cada tipo vendido durante o mês anterior, juntamente com uma média de lucros por cada modelo.

Este problema difere do analisado na secção 10.4, pelo facto de as chaves para a pesquisa não serem fornecidas separadamente, tendo de ser construídas a partir de valores dos códigos de modelo do ficheiro de vendas. Será necessário uma tabela em que se armazene cada um dos diferentes códigos de modelo, juntamente com os dados acumulados para esse modelo. Será também necessário verificar, para cada registo do ficheiro de vendas, se o código do modelo já se encontra na tabela. Se o código do modelo já fizer parte da tabela será necessário actualizar os campos correspondentes; caso contrário, o código deverá ser armazenado na tabela conjuntamente com os seus campos de detalhes.

O programa deve produzir o número de veículos vendidos para cada modelo, a média de vendas e a média de lucros para o mesmo. Assim, a tabela terá de ser um item de grupo repetido formado por quatro itens elementares. Por exemplo:

```
tabela-modelo
        modelo-contador
                        repete-se 50 vezes
                modelo-codigo           código de modelo armazenado
                modelo-vendidos         número de modelos vendidos

                modelo-grosso           valor grosso de vendas deste modelo
                modelo-lucro            valor dos lucros na venda deste modelo
```

O tamanho da tabela pode ser fixado em cinquenta elementos, pois na especificação do problema garantem-nos que «existem menos de cinquenta códigos de modelo em uso».

Logo que todos os dados do ficheiro de vendas de veículos tenham sido acumulados, podem ser calculadas as médias do valor de vendas por grosso e o valor de lucros por cada modelo, procedendo-se depois à sua impressão.

O ficheiro de vendas de veículos irá provavelmente conter uma grande quantidade de informação acerca da venda de cada veículo, mas para os propósitos deste exemplo só serão identificados os campos directamente envolvidos no mesmo. Cada registo do ficheiro de vendas de veículos, «fich–veiculo», irá incluir os seguintes campos:

```
reg-veiculo
        veiculo-codigo          código de modelo de veículo
        veiculo-grosso          valor grosso da venda
        veiculo-lucro           lucro nesta venda
```

A estrutura geral do programa irá seguir a dos exemplos prévios apreciados nas secções 10.2 e 10.4.

```
veiculo-sumario
        inicializar-tabela
        ler-proximo-fich-veiculo
        until fim-de-fich-veiculo do
                actualizar-tabela
                ler-proximo-fich-veiculo
        enduntil
        imprimir-tabela
```

A maneira mais fácil de inicializar a tabela é ler o primeiro registo de «fich-veiculo» e armazenar os detalhes desse registo no primeiro elemento da tabela. Assim, o refinamento de «inicializar-tabela» poderia ser:

```
inicializar-tabela
        ler-proximo-fich-veiculo
        mover-veiculo-codigo para modelo-codigo (1)
        mover 1 para modelo-vendidos (1)
        mover veiculo-grosso para modelo-grosso (1)
        mover veiculo-lucro para modelo-lucro (1)
        mover 1 para mc-limite
```

Este refinamento cria o primeiro elemento da tabela e define o limite corrente de número de elementos através de «mc-limite». Este limite irá ser aumentado à medida que novos elementos forem sendo criados e armazenados.

O próximo passo é definir o refinamento de «actualizar-tabela». Isto requer uma pesquisa à tabela com uma actualização ou a criação de um novo elemento. A estrutura geral deste refinamento é:

```
actualizar-tabela
        mover verdadeiro para modelo-novo
        for mc-indice = 1 to mc-limite do
                if veiculo-codigo = modelo-codigo then
                        mover falso para modelo-novo
                        acumular-detalhes
                endif
        endfor
        if modelo-novo then
                criar-novo elemento
        endif
```

Este refinamento começa por aceitar de uma forma pessimista que o registo corrente irá produzir um código de modelo novo (um que ainda não esteja armazenado na tabela), colocando em verdadeiro o campo de dados booliano modelo-novo. A tabela é então pesquisada para verificar se o código de modelo corrente, «veiculo-codigo», faz já parte da tabela. Se assim for, então o correspondente elemento da tabela é actualizado e o campo de dados booliano modelo-novo colocado em falso, para indicar que o registo corrente foi utilizado por um código de modelo já existente na tabela. Se, no fim do ciclo de pesquisa, «modelo-novo» continuar verdadeiro, então o código de modelo e os seus detalhes correspondentes serão adicionados à tabela.

Existem dois refinamentos de nível mais baixo em «actualizar-tabela» a desenvolver e são os dois bastante lineares:

```
acumular-detalhes
        calcular modelo-vendidos (mc-indice) = modelo-vendidos (mc-indice) + 1
        calcular modelo-grosso (mc-indice) =
                modelo-vendidos (mc-indice) + veiculo-grosso
        calcular modelo-lucro (mc-indice) =
        modelo-vendidos (mc-indice) + veiculo-lucro

criar-novo-elemento
        calcular mc-limite = mc-limite + 1
        mover veiculo-codigo para modelo-codigo (mc-limite)
        mover 1 para modelo-numeros (mc-limite)
        mover veiculo-grosso para modelo-grosso (mc-limite)
        mover veiculo-lucro para modelo-lucro (mc-limite)
```

Recordando o refinamento «inicializar-tabela», existe uma forte semelhança entre ele e «criar-novo-elemento». Isto acontece porque, efectivamente, os dois desenhos executam a mesma função. Assim, o refinamento de «inicializar-tabela» poderá ser modificado para:

```
inicializar-tabela
        ler-proximo-fich-veiculo
        mover 0 para mc-limite
        criar-novo-elemento
```

Após processamento de todos os registos de «fich-veiculo», o refinamento «imprimir-tabela» deve produzir os respectivos sumários de numeros e médias. Pode ser então descrita uma linha de sumário para conter esses valores:

```
linha-sumario
        sumario-codigo          código de modelo
        sumario-vendidos        número de modelos vendidos
        sumario-med-grosso      média grosso de vendas do modelo
        sumario-med-lucro               média de lucros na venda do modelo
```

O refinamento de «imprimir-tabela» é então linear:

```
imprimir-tabela
        imprimir-cabeçalhos
        for mc-indice = 1 to mc-limite do
                mover modelo-codigo (mc-indice) para sumario-codigo
                mover modelo-vendidos para sumario-vendidos
                calcular sumario-med-grosso =
                        modelo-grosso (mc-indice) / modelo-vendidos (mc-indice)
                calcular sumario-med-lucro =
                        modelo-lucro (mc-indice) / modelo-vendidos (mc-indice)
                escrever linha-sumario
        endfor
```

O refinamento de «print-cabeçalhos» poderia envolver a impressão do cabeçalho da companhia no topo da página, seguido dum cabeçalho de mapa para a análise mensal e duas linhas de cabeçalhos de coluna separados dos detalhes por uma linha em branco. Isto poderia ser expresso como se segue:

```
imprimir-cabeçalhos
        escrever-cabec-companhia        (no topo de uma nova página)
        escrever-cabec-mapa             (deixar duas linhas em branco)
```

escrever-cabec-coluna-1 (deixar uma linha em branco)
escrever-cabec-coluna-2
escrever-linha-branco

O desenho de programa completo, incluindo as definições da tabela e do registo, é reproduzido abaixo. O registo de *input* é:

```
reg-veiculo
        veiculo-codigo      código de modelo do veículo
        veiculo-grosso      valor grosso da venda
        veiculo-lucro       lucro nesta venda
```

A tabela, os seus índices e limites associados têm de ser descritos na WORKING-STORAGE SECTION:

```
tabela-modelo
modelo-contador
                repete-se 50 vezes
        modelo-codigo    código de modelo armazenado
        modelo-vendidos  número de modelos vendidos
        modelo-grosso    valor grosso de venda deste modelo
        modelo-lucro     valor dos lucros na venda deste modelo
mc-indice                índice para o contador de modelos
mc-limite                limite dos elementos usados no contador de modelos
```

Em adição a estes campos temporários, o campo de dados booliano modelo-novo deve ser adicionado a VECTOR-ESTADO.

O registo de *output* é perfeitamente linear para este caso:

```
linha-sumario
        sumario-codigo       código de modelo
        sumario-vendidos     número de modelos vendidos
        sumario-reg-grosso   média grosso de vendas do modelo
        sumario-med-lucro          média de lucros na venda do modelo
```

Abaixo é fornecido o desenho completo, utilizando as estruturas de dados criadas acima.

```
veiculo-sumario
        inicializar-tabela
        ler-proximo-fich-veiculoo
        until fim-de-fich-veiculo do
                actualizar-tabela
                ler-proximo-fich-veiculo
        enduntil
        imprimir-tabela

inicializar-tabela
        ler-proximo-fich-veiculo
        mover 0 para mc-limite
        criar-novo-elemento

actualizar-tabela
        mover verdadeiro para modelo-novo
        for mc-indice = 1 to mc-limite do
                if veiculo-codigo = modelo-codigo then
```

```
                        mover falso para modelo-novo
                        acumular-detalhes
                endif
        endfor
        if modelo-novo then
                criar-novo-elemento
        endif

acumular-detalhes
        calcular modelo-vendidos (mc-indice) =
                modelo-vendidos (mc-indice) + 1 calcular modelo-grosso (mc-indice)
                = modelo-vendidos (mc-indice) + veiculo-grosso
        calcular modelo-lucro (mc-indice) =
                modelo-vendidos (mc-indice) + veiculo-lucro

criar-novo-elemento
        calcular mc-limite = mc-limite + 1
        mover veiculo-codigo para modelo-codigo (mc-limite)
        mover 1 para modelo-numeros (mc-limite)
        mover veiculo-grosso para modelo-grosso (mc-limite)
        mover veiculo-lucro para modelo-lucro (mc-limite)

imprimir-tabela
        imprimir-cabeçalhos
        for mc-indice = 1 to mc-limite do
                mover modelo-codigo (mc-indice) para sumario-codigo
                mover modelo-vendidos para sumario-vendidos
                calcular sumario-med-grosso =
                        modelo-grosso (mc-indice) / modelo-vendidos (mc-indice)
                calcular sumario-med-lucro =
                        modelo-lucro (mc-indice) / modelo-vendidos (mc-indice)
                escrever linha-sumario
        endfor

imprimir-cabeçalhos
        escrever-cabec-companhia        (no topo de uma nova página)
        escrever-cabec-mapa             (deixar duas linhas em branco)
        escrever-cabec-coluna-1         (deixar uma linha em branco)
        escrever-cabec-coluna-2
        escrever-linha-branco

PROCESS LET
IDENTIFICATION DIVISION.
PROGRAM-ID. TABELA01.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-S34.
OBJECT-COMPUTER. IBM-S34.
*       <nome-computador> deve ser substituído pelo nome
*       do computador usado para correr os programas.

INPUT-OUTPUT SECTION.
FILE-CONTROL.
        SELECT FVEICUL ASSIGN TO DISK-FVEICUL.
        SELECT MAPA ASSIGN TO PRINTER-MAPA.
*       <sistema-input> e <sistema-output> devem estar conformes
*       com as regras do sistema utilizado.
```

```
DATA DIVISION.
FILE SECTION.
FD      FVEICUL
        LABEL RECORDS STANDARD.
01      REG-VEICULO.
        05      VEICULO-CODIGO              PIC XXX.
        05      VEICULO-GROSSO              PIC 9(5)V99.
        05      VEICULO-LUCRO               PIC S9(4)V99.
FD      MAPA
        LABEL RECORDS STANDARD
01      LINHA-OUT                           PIC X(120).

WORKING-STORAGE SECTION.
01      VECTOR-ESTADO.
        05      FIM-FIC-VEICUL              PIC X.
                88      FIM-DE-FIC-VEICUL           VALUE «F».
        05      INDICA-MODELO                       PIC X.
        88 MODELO-NOVO                             VALUE «V».

01      TABELA-MODELO.
        05      MODELO-CONTADOR    OCCURS 50 TIMES.
                10      MODELO-CODIGO       PIC XXX.
                10      MODELO-VENDIDOS     PIC 99.
                10      MODELO-GROSSO       PIC 9(7)V99.
                10      MODELO-LUCRO        PIC S9(6)V99.
        05      MC-INDICE                   PIC 99.
        05      MC-LIMITE                   PIC 99.

01      CABEC-COMPANHIA.
        05      FILLER                      PIC X(5)  VALUE SPACES.
        05      FILLER                      PIC X(28)  VALUE
                «HONEST JOE (AUTOMOVEIS) LDA.».

01      CABEC-MAPA.
        05      FILLER              PIC X           VALUE SPACE.
        05      FILLER              PIC X(32)       VALUE
                «SUMARIO MENSAL DE VENDAS AUTO».

01      CABEC-COLUNA-1.
        05      FILLER              PIC X           VALUE «MODELO».
        05      FILLER              PIC XXX         VALUE SPACES.
        05      FILLER              PIC X(6)        VALUE «NUMERO».
        05      FILLER              PIC XXX         VALUE SPACES.
        05      FILLER              PIC X(5)        VALUE «MEDIA».
        05      FILLER              PIC XXXX        VALUE SPACES.
        05      FILLER              PIC X(5)        VALUE «MEDIA».
01      CABEC-COLUNA-2.
        05      FILLER              PIC X(6)        VALUE «CODIGO».
        05      FILLER              PIC X(5)        VALUE SPACES.
        05      FILLER              PIC X(8)        VALUE «VENDIDOS».
        05      FILLER              PIC X(5)        VALUE SPACES.
        05      FILLER              PIC X(6)        VALUE «GROSSO».
        05      FILLER              PIC X(5)        VALUE SPACES.
        05      FILLER              PIC X(5)        VALUE «LUCRO».
```

```
01      LINHA-BRANCO.
        05      FILLER                  PIC X           VALUE SPACE

01      LINHA-SUMARIO.
        05      FILLER                  PIC X.
        05      SUMARIO-CODIGO          PIC XXX.
        05      FILLER                  PIC X(6).
        05      SUMARIO-VENDIDOS        PIC Z9.
        05      FILLER                  PIC XXX.
        05      SUMARIO-MED-GROSSO      PIC £££,££9.99.
        05      FILLER                  PIC XX.
        05      SUMARIO-MED-LUCRO       PIC ££££9.99CB.

PROCEDURE DIVISION.
PROGRAMA-PRINCIPAL.
        PERFORM ESTADO-INIC
        PERFORM VEICULO-SUMARIO.
        PERFORM FECHO.
ESTADO-INIC.
        MOVE SPACE TO FIM-FIC-VEICUL
        OPEN INPUT FVEICUL OUTPUT MAPA
        MOVE SPACES TO LINHA-SUMARIO.
FECHO
        CLODE FVEICUL MAPA
        STOP RUN.
LER-PROXIMO-VEICUL0.
        READ FVEICUL
                AT END MOVE «F» TO FIM-FIC-VEICUL.
ESCREVER-LINHA-SUMARIO.
        WRITE LINHA-OUT FROM LINHA-SUMARIO
                AFTER ADVANCING 1 LINES.
        MOVE SPACES TO LINHA-SUMARIO.
ESCREVER-CABEC-COMPANHIA.
        WRITE LINHA-OUT FROM CABEC-COMPANHIA
                AFTER ADVANCING PAGE.
ESCREVER-CABEC-MAPA.
        WRITE LINHA-OUT FROM CABEC-MAPA
                AFTER ADVANCING 3 LINES.
ESCREVER-CABEC-COLUNA-1.
        WRITE LINHA-OUT FROM CABEC-COLUNA-1
                AFTER ADVANCING 2 LINES.
ESCREVER-CABEC-COLUNA-2.
        WRITE LINHA-OUT FROM CABEC-COLUNA-2
                AFTER ADVANCING 1 LINES.
ESCREVER-LINHA-BRANCO.
        WRITE LINHA-OUT FROM LINHA-BRANCO
                AFTER ADVANCING 1 LINES.
VEICULO-SUMARIO.
        PERFORM INICIALIZAR-TABELA.
        PERFORM LER-PROXIMO-VEICULO
        PERFORM PROCESSAR-REG-VEICULO
                UNTIL FIM-DE-FIC-VEICULO
        PERFORM IMPRIMIR-TABELA.
PROCESSAR-REG-VEICULO.
        PERFORM ACTUALIZAR-TABELA
```

```
        PERFORM  LER-PROXIMO-VEICULO.
INICIALIZAR-TABELA.
        PERFORM  LER-PROXIMO-VEICULO
        MOVE O TO MC-LIMITE
        PERFORM  CRIAR-NOVO-ELEMENTO.
ACTUALIZAR-TABELA.
        MOVE «V» TO INDICA-MODELO
        PERFORM  PESQUISAR-TABELA
                VARYING MC-INDICE FROM 1
                BY 1 UNTIL MC-INDICE > MC-LIMITE
        IF  MODELO-NOVO
                PERFORM  CRIAR-NOVO-ELEMENTO.
PESQUISAR-TABELA.
        IF  VEICULO-CODIGO = MODELO-CODIGO (MC-INDICE)
                MOVE «F» TO INDICA-MODELO
                PERFORM  ACUMULAR-DETALHES.
ACUMULAR-DETALHES.
        COMPUTE  MODELO-VENDIDOS (MC-INDICE) =
                MODELO-VENDIDOS (MC-INDICE) + 1
        COMPUTE  MODELO-GROSSO (MC-INDICE) =
                MODELO-VENDIDOS (MC-INDICE0) + VEICULO GROSSO
        COMPUTE  MODELO-LUCRO (MC-INDICE) =
                MODELO-VENDIDOS (MC-INDICE) + VEICULO-LUCRO.
CRIAR-NOVO-ELEMENTO.
        COMPUTE MC-LIMITE = MC-LIMITE + 1
        MOVE  VEICULO-CODIGO TO MODELO-CODIGO (MC-LIMITE)
        MOVE 1 TO MODELO-VENDIDOS (MC-LIMITE)
        MOVE  VEICULO-GROSSO TO MODELO-GROSSO (MC-LIMITE)
        MOVE  VEICULO-LUCRO TO MODELO-LUCRO (MC-LIMITE).
IMPRIMIR-TABELA.
        PERFORM  IMPRIMIR-CABECALHOS
        PERFORM  IMPRIMIR-DETALHES
                VARYING MC-INDICE FROM 1
                BY 1 UNTIL MC-INDICE > MC-LIMITE.
IMPRIMIR-DETALHES.
        MOVE  MODELO-CODIGO (MC-INDICE) TO SUMARIO-CODIGO
        MOVE  MODELO-VENDIDOS (MC-INDICE) TO SUMARIO-VENDIDOS
        COMPUTE  SUMARIO-MED-GROSSO =
        MODELO-GROSSO (MC-INDICE) / MODELO-VENDIDOS (MC-INDICE)
        COMPUTE  SUMARIO-MED-LUCRO =
        MODELO-LUCRO (MC-INDICE) / MODELO-VENDIDOS (MC-INDICE)
        PERFORM  ESCREVER-LINHA-SUMARIO.
IMPRIMIR-CABECALHOS.
        PERFORM  ESCREVER-CABEC-COMPANHIA
        PERFORM  ESCREVER-CABEC-MAPA
        PERFORM  ESCREVER-CABEC-COLUNA-1
        PERFORM  ESCREVER-CABEC-COLUNA-2
        PERFORM  ESCREVER-LINHA-BRANCO.
```

Na figura 10.6 é mostrado um exemplo do *output* deste programa. Note que as médias de vendas e lucros serão truncadas pelas operações de divisão especificadas em IMPRIMIR-DETALHES.

HONEST JOE (AUTOMOVEIS) LDA.

SUMARIO MENSAL DE VENDAS AUTO

MODELO CODIGO	NUMERO VENDIDOS	MEDIA GROSSO	MEDIA LUCRO
XX3	2	£17,142.00	£1215.00
E11	15	£4,976,23	£416.38
65R	1	£6,450.00	£250,00DB
U10	5	£623.44	£27.98
UX3	1	£75.00	£0.00
U93	6	£987.34	£2.85DB

Fig. 10.6

Exercícios 10

10.1. Recordando o estudo de caso da secção 10.6, explique o que se convenciona acerca do número máximo de um dado modelo que se espera vender. Identifique os campos relacionados com esta assunção.

O negócio da Honest Joe expande-se e o número de códigos de modelo aumenta para 150. Especifique as alterações necessárias no desenho de programa e no correspondente programa COBOL.

10.2. A estrutura seguinte aparece num programa de COBOL utilizado por um escritório de câmbios.

```
01      CAMPOS-TEMPORÁRIOS.
        05      QUANTIA-REQUERIDA                   PIC 999V99.
01      TABELA-CAMBIO.
        05      CAMBIO-VALOR OCCURS 40 TIMES.
                10      CAMBIO-CODIGO               PIC AAA.
                10      CAMBIO-TROCA                PIC 9(5)V99.
        05      NUMERO-OCORRENCIAS                  PIC 99.
```

São dados três valores alfabéticos para o código de vários câmbios estrangeiros (CAMBIO-CODIGO), sendo os mesmos armazenados na tabela com a média do câmbio (CAMBIO-TROCA). O valor do câmbio é definido pela razão entre a moeda estrangeira comprada e uma libra esterlina. O valor do câmbio correctamente cotado é fornecido por NUMERO-OCORRENCIAS. O campo QUANTIA-REQUERIDA é usado para expressar uma quantia em libras esterlinas. Partindo do pressuposto de que os campos necessários tenham já sido convenientemente inicializados na estrutura acima, escreva partes de desenho de programa, e as correspondentes traduções para COBOL, para executar as seguintes operações:

a) Actualize a média de câmbio do marco alemão, «GDM», para 4.21;

b) Procure a média de câmbio para o franco francês, «FFR», e calcule a quantia nesta moeda necessária para corresponder ao valor actual de QUANTIA-REQUERIDA.

c) Adicione uma nova moeda estrangeira de código «NGU» à tabela com a média de 3,88.

Dê descrições de quaiquer índices utilizados e também de campos de trabalho exigido.

Exercícios de programação 10

10.3. Desenvolva o exemplo de 10.1 num programa completo de COBOL que imprima a média das melhores e piores vendas de cada filial. Utilize o seguinte formato de desenho de registo de *input* para o programa:

```
Registo de vendas
        Identificador de filial        2 letras e 4 dígitos
        Valor de vendas mensais  6 dígitos (pénis)
                repete-se 12 vezes
```

O programa deve incluir um cabeçalho de página apropriado e cabeçalhos de coluna para o *output*.

Quando tiver conseguido pôr a funcionar este simples programa comparativo, considere a ampliação do programa para incluir no cálculo da média de vendas mensais o melhor e o pior volume de vendas para a total organização. [Pista: será necessário uma tabela na WORKING-STORAGE SECTION para acumular os totais mensais de toda a organização. Após serem processadas as filiais, esta tabela pode ser analisada para fornecer esses três valores.]

10.4. Uma firma de grandes dimensões que cria *software* para um grupo de empresas tem um sistema de numeração de projectos onde os primeiros três dígitos do número de projecto são só para um determinado cliente. Um número de projectos pode ser executado simultaneamente para um mesmo cliente.

Cada empregado da firma preenche uma folha de presenças semanal relatando o tempo que ele ou ela gastou com cada projecto onde esteja envolvido. Os clientes terão então de pagar os tempos baseados nessas informações. Existe, portanto, um ficheiro de grandes dimensões com as informações dos tempos gastos por cada empregado num determinado projecto.

Cada registo contém a seguinte informação:

```
Registo de horas
        Numero de empregado              6 dígitos
        Número da semana                       2 dígitos
        Número do projecto               6 dígitos
        Tempo gasto com o projecto             2 dígitos (horas)
        Preço por hora do empregado      4 dígitos (pénis)
```

Requer-se um programa que verifique, através do processamento de todo o ficheiro acumulado, o tempo total gasto e o valor total a cobrar, para cada cliente.

Considere que todos os projectos para o primeiro cliente têm um valor inicial «100», para o segundo «110», etc. Existem actualmente vinte clientes, numerados contiguamente, tendo, assim, o último número de projecto a começar em «290».

Será que o método que utilizou funcionaria para projectos em que os mesmos teriam para o primeiro cliente números entre 100-124, para o segundo 125-149, etc.?

10.5. Todos os movimentos processados por uma secção de vendas a crédito vão para um ficheiro de movimentos diário do seguinte formato:

```
Registo de movimentos
        Referência do movimento          1 letra e 5 dígitos
        Data do movimento                6 dígitos (ddmnaa)
        Número de conta                  2 letras, 3 dígitos e uma letra
        Valor do movimento               5 dígitos (pénis, com sinal)
```

O movimento pode ser uma venda, quando o valor do movimento é negativo, ou um pagamento, quando o movimento é positivo.

Requer-se um programa que irá aceitar uma selecção aleatória de números de conta, bem como a abertura de balanços para cada uma delas. O programa deve então processar o ficheiro de movimentos, produzindo um mapa de números de movimentos e balanço de fecho para cada um deles. Por exemplo, o *output* poderia começar por:

CODIGO PROJECTO	NUMERO DE MOVIMENTOS	BALANÇO DE FECHO
100	12	£234.56
110	0	£23.75DB

As contas requeridas serão especificadas num ficheiro em que cada registo tem o seguinte formato:

Registo de conta

Número de conta	2 letras, 3 dígitos e 1 letra
Balanço de abertura	7 dígitos, (pénis, com sinal)

Nunca existirão mais de quarenta valores diferentes de números de conta especificados no ficheiro.

11. CARACTERÍSTICAS ADICIONAIS DO COBOL

Este capítulo propõe-se fornecer uma breve cobertura de um número de características úteis do COBOL para as quais não haverá espaço para uma abordagem mais em pormenor.

A primeira característica a ser estudada será o salto incondicional, a instrução GO TO, que fornece um mecanismo extra para a transferência de controlo dentro do programa. Os discípulos da programação estruturada pura argumentam (e podem prová-lo) que as instruções de salto incondicional são desnecessárias numa linguagem que fornece estruturas de controlo «apropriadas» (sequência, selecção e iteração). Na opinião do autor, existem situações especiais em que a utilização de um salto incondicional é de muita utilidade, e isto será analisado na primeira secção.

A tecnica para o desenvolvimento de programas advogada neste livro é a de refinamento passo a passo — a participação de um problema em subproblemas e a continuação desta acção até que os subproblemas se tornam fáceis de resolver. No entanto, todas as soluções conseguidas até agora têm sido sob a forma de um programa único. Para a resolução de um problema complexo talvez seja necessário a implementação da utilização de uma colecção de subprogramas inter-relacionados. A intercomunicação de programas dentro de um programa será apreciada na segunda secção deste capítulo. Até agora temos suposto que todos os ficheiros de *input* e *output* utilizados por um programa são sequenciais, de acesso sequencial. Muitos problemas podem ser resolvidos mais naturalmente se se utilizarem ficheiros em que um dado registo pode ser tratado sem existir referência a outros registos dentro do ficheiro; isto pode ser conseguido através de ficheiros de acesso directo. Em COBOL são fornecidas duas estruturas de ficheiros de acesso directo: relativo e indexado; eles são analisados nas duas secções seguintes deste capítulo.

11.1 — A INSTRUÇÃO «GO TO»

Neste livro tem sido dada grande ênfase à programação metódica utilizando o refinamento passo a passo. Programas bastante complexos foram construídos utilizando os princípios dos controlos da programação estruturada: if, until e for, conjuntamente com os refinamentos.

De parceria com muitas outras linguagens de programação, o COBOL fornece uma instrução de salto incondicional que tem o seguinte formato:

```
GO TO nome-parágrafo
```

O efeito desta instrução é a transferência de controlo para o parágrafo nela designado, o que significa que a instrução a ser executada é a primeira executável após o «nome-parágrafo». Ao contrário do que acontece com o «PERFORM nome-parágrafo», não existe retorno à instrução seguinte a um GO TO.

A filosofia geral da programação estruturada é evitar a utilização do GO TO, por esta instrução ser considerada desnecessária e tender a obscurecer o significado do programa se usada indiscriminadamente. O uso do GO TO deve ser reservado para circunstâncias em que é necessário definir uma saída especial de um ciclo ou de um programa.

No algoritmo para pesquisa numa tabela, dado na secção 10.4, surgia um problema quando a tabela estava quase a entrar em *overflow* (transbordar); foi então sugerido o seguinte refinamento:

```
verificar-excesso-tabela
        if pt-indice > <tamanho-tabela> then
                display «especificados demasiados artigos»
                display «maximo possivel <tamanho-tabela>»
                fecho
        endif
```

Quando o refinamento «fecho» for executado, o programa irá parar porque este refinamento inclui uma instrução de STOP RUN. No entanto, o programa seria mais claro se incluísse no refinamento acima um salto explícito para «fecho». Por exemplo:

```
verificar-excesso-tabela
        if pt-indice > <tamanho-tabela> then
                display «especificados demasiados artigos»
                display «maximo possível <tamanho-tabela>»
                go to fecho
        endif
```

Isto salienta que, em vez de existir uma execução de refinamento seguido de uma transferência de controlo de novo para a sequência principal das operações, existe um salto para o ponto de saída do programa.

Se <tamanho-tabela> tiver o valor 30, então o programa de COBOL equivalente seria:

```
VERIFICAR-EXCESSO-TABELA.
        IF PT-INDICE > 30
                DISPLAY «ESPECIFICADOS DEMASIADOS ARTIGOS»
                DISPLAY«MAXIMO POSSÍVEL E' 30»
                GO TO FECHO.
```

Os leitores que utilizem compiladores de COBOL de baixo nível poderão necessitar de utilizar instruções de GO TO para tradução de operações until e for; estas traduções foram utilizadas nos apêndices 5.3 e 5.5, respectivamente.

11.2 — COMUNICAÇÃO INTERPROGRAMAS

O tamanho de qualquer programa de COBOL isolado está fisicamente limitado pelo tamanho da memória e limitações no endereçamento do sistema de computador utilizado. Em alguns sistemas pequenos pode ser necessário considerar a hipótese de segmentar grandes programas isto envolve a recriação de certas secções do programa e está para além dos propósitos deste livro. No entanto, na maior parte dos sistemas, a segmentação não é necessária porque o limite físico no tamanho do programa excede o tamanho-limite para um programa maleável e simples de entender.

Quando se lida com programas complexos, pode não ser suficiente partir a solução do problema em refinamentos para um só programa. Pode ser necessário subdividir a tarefa utilizan-

do ou uma colecção de programas separados, comunicando através de ficheiros, ou uma estrutura de programas inter-relacionados. Esta última técnica requer um utilitário que chame outro programa para executar uma determinada subtarefa. Em qualquer interacção entre programas existe um programa chamador e um programa chamado. Quando o programa chamado acabar a execução da sua subtarefa, o controlo é devolvido ao programa chamador através de um mecanismo semelhante ao do PERFORM.

Chamada de programas

Para transferir o controlo para o programa chamado, o programa chamador utiliza uma instrução CALL. Por exemplo:

```
CALL «MODULO-ERRO»
```

iria transferir o controlo do programa corrente para o programa MODULO-ERRO. Não existe nenhuma transferência directa de dados entre estes dois programas, o que chama e o que é chamado; no entanto, podem ser transferidos indirectamente pela utilização de ficheiros.

Uma extensão do formato do CALL permite passar dados para o programa chamado e receber os seus resultados de novo para o ficheiro chamador. Por exemplo:

```
CALL «CONV-DATA» USING DATA-VENDA DATA-COMPLETA
```

chamaria o programa CONV-DATA e permitiria que os campos DATA-VENDA e DATA-COMPLETA fossem utilizados para a transferência de dados entre o programa chamador e CONV-DATA.

Neste exemplo, o programa chamado está desenhado para converter uma data abreviada, num formato número-dia a partir duma década, numa data completa. Por exemplo, o dia 1106 seria transformado em 13 de Janeiro de 1983 se os números-dia armazenados tiverem considerado como primeiro dia o dia 1 de Janeiro de 1980. Os campos DATA-VENDA e DATA-COMPLETA poderiam ser descritos no programa chamador da seguinte forma:

```
05      DATA-VENDA.                 PIC 9999.
....
05      DATA-COMPLETA.
        10      DIA-COMPLETO        PIC 99.
        10      MES-COMPLETO                PIC A(9).
        10      ANO-COMPLETO                PIC 9999.
```

É agora necessário olharmos para a estrutura do ficheiro chamado, para ver como podem utilizar-se estes campos de dados.

Programas chamados

Existem três características essenciais num programa chamado:

- A descrição de qualquer campo de dados para a transferência de dados do e para o programa que chama;
- Estabelecimento duma correspondência entre esses mesmos campos e os da instrução de CALL no programa que chama;
- Definição do ponto de saída do programa chamado.

Os campos utilizados para a comunicação interprogramas são especificados na LINKAGE-SECTION da DATA-DIVISION. Esta é a terceira secção da DATA-DIVISION, que é somente uti-

lizada nos programas chamados. O seu formato geral é similar ao da WORKING-STORAGE SECTION. Assim, no programa do exemplo CONV-DATA, poder-se-ia definir uma LINKAGE-SECTION do seguinte formato:

```
LINKAGE-SECTION.
01      DATA-INPUT.              PIC 9999.
01      DATA-OUTPUT.
        05      DIA-OUTPUT       PIC 99.
        05      MES-OUTPUT       PIC A(9).
        05      ANO-OUTPUT       PIC 9999.
```

Para se estabelecer a correspondência entre estes campos e os utilizados na instrução de CALL do programa que chama, as descrições de registos dadas na LINKAGE SECTION são fornecidas numa cláusula USING escrita junto com o cabeçalho da PROCEDURE DIVISION do programa chamado. Por exemplo:

```
PROCEDURE DIVISION USING DATA-INPUT DATA-OUTPUT.
```

Os campos listados depois do USING correspondem aos mesmos da instrução CALL, que para este caso eram:

```
CALL «CONV-DATA» USING DATA-VENDA DATA-COMPLETA
```

Assim, DATA-VENDA tem correspondência com DATA-INPUT e DATA-COMPLETA com DATA–OUTPUT.

O programa que chama deverá ter colocado um valor em DATA-VENDA antes de chamar o programa CONV-DATA; o programa chamado pode então utilizar este valor em DATA-INPUT e colocar os resultados em DATA-OUTPUT. Voltando agora ao programa que chama, os resultados estarão disponíveis em DATA-COMPLETA. Se CONV-DATA alterar o valor de DATA-INPUT, isto irá reflectir-se no programa que chama <DATA-VENDA irá ser alterado>.

Noutras linguagens de programação, DATA-INPUT e DATA-OUTPUT seriam designados «parâmetros formais» e DATA-VENDA e DATA-COMPLETA «parâmetros actuais». Tecnicamente, o método de passagem de parâmetros em COBOL é designado «chamada por referência». O programa chamado é equivalente a um subprograma sub-rotina em FORTRAN ou a um procedimento em Pascal ou Algol.

Os campos de dados listados após o USING são referenciados por posição e não por nome, permitindo assim que seja possível chamar CONV-DATA através de outros campos, por exemplo:

```
CALL «CONV-DATA» USING DATA-FACTURA DATA-EXPANDIDA
```

Os campos irão agora ser referenciados por forma a que DATA-FACTURA corresponda a DATA INPUT e DATA-EXPANDIDA a DATA-OUTPUT.

Para devolver o controlo ao programa que chama, o último parágrafo a ser executado deve conter uma só instrução de EXIT PROGRAM. Por exemplo:

```
RETORNO-AO-CHAMADOR.
        EXIT PROGRAM.
```

No caso do primeiro exemplo para a instrução CALL dado acima:

```
CALL «MODULO-ERRO»
```

não existe transferência directa de dados. Assim, a única diferença entre o programa MODULO-ERRO e um normal seria a utilização de EXIT PROGRAM em vez de STOP RUN.

Um programa chamado pode também chamar outros programas desde que directa ou indirectamente não vá chamar o programa que o chamou, isto é, não são permitidos recursos em COBOL.

11.3 — BIBLIOTECA DE «SOURCE» [FONTE] E DIRECTIVAS DE CÓPIA

É possível manter uma bibioteca de fragmentos de programas COBOL, que podem ser copiados para dentro de um programa antes da compilação. Isto poderia permitir, por exemplo, que uma instalação particular tivesse uma CONFIGURATION SECTION *standard* que seria copiada da biblioteca para um programa de COBOL que estivesse a ser escrito.

Iria permitir também que um grupo de programadores que estivessem a trabalhar em programas inter-relacionados armazenassem descrições de ficheiros dos ficheiros mais utilizados e os copiassem para cada um dos seus programas. Desta forma, qualquer alteração relevante ao formato de um ficheiro iria automaticamente ser inserida em todos os programas.

O verbo COPY, utilizado para copiar texto da biblioteca para o programa, é chamado directiva do compilador: isto é, afecta muito mais a compilação do que a execução de um programa. Uma directiva de COPY pode aparecer em qualquer ponto do programa; não se encontra restrita a uma divisão ou contexto.

O formato mais simples da directiva COPY é:

```
COPY  nome-membro-biblioteca.
```

onde nome-membro-biblioteca é o nome único de um membro existente na biblioteca de *sources* do COBOL. Por exemplo:

```
. . . . .
ENVIRONMENT  DIVISION.
COPY  CONFIG-SECTION.
INPUT-OUTPUT  SECTION.
. . . . .
```

A directiva «COPY CONFIG-SECTION.» será substituída pelo pedaço de texto COBOL com o nome de CONFIG-SECTION que se encontra na bilioteca de *sources*.

Suponha que o membro CONFIG-SECTIO contém o seguinte:

```
CONFIGURATION  SECTION.
SOURCE-COMPUTER.  MM1.
OBJECT-COMPUTER.  MM1.
SPECIAL-NAMES.
        MM-CONTROLE  IS  OPERATORS-CONSOLE.
```

Então, este bocado de texto seria copiado para o programa antes de o mesmo ser compilado. Assim, a forma expandida do programa seria:

```
. . . . .
ENVIRONMENT  DIVISION.
CONFIGURATION  SECTION.
SOURCE-COMPUTER.  MM1.
OBJECT-COMPUTER.  MM1
SPECIAL-NAMES.
        MM-CONTROLE  IS  OPERATORS-CONSOLE.
INPUT-OUTPUT  SECTION.
. . . . .
```

Uma forma avançada da directiva COPY permite que sejam copiados textos da biblioteca sistematicamente, alterando os seus conteúdos por novos pedaços de texto. Isto iria permitir, por exemplo, que se armazenasse na biblioteca um algoritmo geral que, copiado para um programa, substituísse o nome de alguns campos por nomes apropriados para o corrente programa.

Para ilustrar o formato da directiva COPY com a opção de REPLACING, suponha que se requer um programa com uma CONFIGURATION SECTION modificada, baseada no membro CONFIG-SECTION da biblioteca. Seria então possível escrever:

```
COPY  CONFIG-SECTION
        REPLACING MM1 BY XYZ-82
                  MM-CONSOLE  BY  XYZ-TYPEWRITER.
```

A forma expandida do programa irá agora conter:

```
. . . . .
ENVIRONMENT  DIVISION.
CONFIGURATION  SECTION.
SOURCE-COMPUTER.  XYZ-82.
OBJECT-COMPUTER.  XYZ-82.
SPECIAL-NAMES.
        XYZ-TYPEWRITER  IS  OPERATORS-CONSOLE.
INPUT-OUTPUT  SECTION.
. . . . .
```

A directiva COPY pode especificar qualquer número de alterações a serem efectuadas quando um membro da biblioteca é copiado para dentro do programa.

A forma como os textos são adicionados às bibliotecas e o formato do ficheiro ou ficheiros de biblioteca é tarefa dos implementadores de sistemas, pelo que eles irão variar de sistema para sistema.

11.4 — FICHEIROS RELATIVOS

Um ficheiro relativo pode ser considerado uma extensão da ideia de tabela a uma dimensão, em que os seus elementos são registos. Um ficheiro relativo pode ser imaginado como uma sequência de áreas de dados, em que cada uma tem o tamanho suficiente para comportar um registo do ficheiro.

Uma área é unicamente identificada por um número relativo de registo, que pode ser utilizado para o acesso a essa área e não outra diferente. A primeira área do ficheiro tem sempre o número relativo um, estando as seguintes numeradas em sequência: dois, três, etc. Da mesma forma, o número relativo funciona como um índice de uma tabela.

Não é obrigatório que todas as áreas do ficheiro contenham dados; assim, pode ser escrito um registo dentro de uma área com o número relativo cinquenta, independentemente do facto de as quarenta e nove primeiras conterem dados. Por princípio, todas as áreas de um ficheiro relativo devem ser consideradas vazias, tendo este estado especial de ser previsto nas instruções de *input/output* de acesso ao ficheiro.

Para o uso de ficheiros relativos interessa aumentar o repertório de instruções de *input/output,* pelo que se fornecem as seguintes:

READ — ler um registo de uma área especificada;
WRITE — escrever um registo em determinada área;
REWRITE — escrever um registo em determinada área, por cima do registo já lá existente;
DELETE — apagar os conteúdos de determinada área, deixando-a «vazia».

Em qualquer dos casos, a área deve ser identificada pelo seu número relativo de registo. Todas as instruções acima devem ter um qualificador de INVALID KEY (chave inválida) para especificar o que acontece se ocorrer uma tentativa de execução de uma operação inválida. No caso do READ, REWRITE e DELETE, uma condição de chave inválida ocorre se a área especificada estiver vazia, enquanto para o WRITE a condição ocorre se a respectiva área contiver um registo.

Para ilustrar o uso de ficheiros relativos, será delineada uma solução de um programa simples. Os clientes da Erewhon Building Society têm registos de contas poupança com números de conta com dez dígitos; os primeiros três dígitos identificam o tipo de conta, os próximos três o ramo ao qual a conta é entregue e os últimos quatro dígitos constituem o número do cliente nesse ramo. Cada ramo requer um ficheiro de contas de clientes que podem ser actualizadas directamente por movimentos desencadeados a partir das instituições do ramo. Isto pode ser conseguido através da utilização de um ficheiro relativo que utiliza os últimos quatro dígitos do número de conta do cliente como número relativo de registo.

Para se descrever um ficheiro relativo, é necessário utilizar um formato modificado da cláusula SELECT. Por exemplo:

```
ENVIRONMENT  DIVISION.
. . . . .
INPUT-OUTPUT  SELECTION.
FILE-CONTROL.
        SELECT  FICH-CLIENTE  ASSIGN  TO  <nome-externo-ficheiro>
        ORGANIZATION  RELATIVE
        ACCESS  MODE  RANDOM
        RELATIVE  KEY  FICH-CLIENTE-CHAVE.
```

Repara que não existe ponto final após <nome-externo-ficheiro> porque as linhas que se seguem são parte integrante da declaração SELECT. O campo FICH-CLIENTE-CHAVE não deve ser parte da descrição do registo associado à descrição do ficheiro de FICH-CLIENTE.

A cláusula ORGANIZATION especifica como os registos estão armazenados no ficheiro. Na declaração SELECT acima, ORGANIZATION RELATIVE indica que os registos em FICH-CLIENTE estão armazenados de tal forma que só é possível identificá-los unicamente por número relativo de registo. A cláusula ACCESS MODE especifica a forma como o ficheiro deve ser utilizado por um programa particular. No exemplo acima, ACCESS MOD RANDOM especifica que o programa irá aceder aos registos «aleatoriamente», isto é, utilizando o número relativo de registo para seleccionar registos individuais para manipulação.

Na DATA-DIVISION, a descrição de ficheiro para FICH-CLIENTE erá parecer semelhante a outras descrições feitas já neste livro; os pormenores de organização de ficheiro estão contidos na cláusula SELECT dada acima.

Por exemplo:

```
DATA  DIVISION.
FILE  SECTION.
FD      FICH-CLIENTE
        LABEL  RECORDS  STANDARD.
01      REG-CLIENTE.
        05      CLIENTE-NUM-CONTA              PIC 9(10).
        . . . . .
```

Para que o campo CLIENTE-NUM-CONTA possa conter o número relativo de registo, terá de ser descrito na WORKING-STORAGE SECTION, possivelmente como parte do VECTOR-ESTADO. Por exemplo:

```
WORKING-STORAGE  SECTION.
01      VECTOR-ESTADO.
        05      FICH-CLIENTE-CHAVE              PIC 9999.
        .....
```

Tal como um campo de índice, qualquer campo que seja utilizado para conter um número relativo de registo terá de ser um inteiro de tipo numérico e não deverá ser editado. O tamanho deste campo irá definir o limite absoluto do ficheiro, neste caso 9999 registos.

Avançando para a PROCEDURE DIVISION, o ficheiro terá de ser aberto antes da sua utilização. Terá de ser aberto para *input* e *output* simultaneamente, por forma a fazer uso total da sua condição de ficheiro relativo. Por exemplo:

```
OPEN I-O FICH-CLIENTE . . . . .
```

A opção «I-O» da declaração OPEN irá permitir a um ficheiro ser utilizado como *input* e *output*.

Para obter o registo relacionado com determinado cliente, por exemplo para imprimir o balanço actual, é necessário especificar o número relativo de registo do referido registo e depois desencadear uma instrução de READ. Para o exemplo, isto poderia ser escrito assim:

```
ENCONTRAR-REG.
        MOVE  NUM-CONTA  TO  FICH-CLIENTE-CHAVE
        READ  FICH-CLIENTE
                INVALID  KEY  PERFORM  REG-INEXISTENTE.
```

Se NUM-CONTA for um número de conta completo com dez dígitos, a instrução de MOVE terá de truncar os dígitos mais significativos, deixando os quatro registos menos significativos em FICH–CLIENTE-CHAVE. A instrução de READ irá colocar o registo de cliente apropriado na área de *input* do programa para o ficheiro (que será também a área de *output*, pois o ficheiro foi previamente aberto para I-O). Se não existir nenhum registo com o número relativo de registo fornecido, então o READ falha e o procedimento REG-INEXISTENTE é executado.

Para actualizar um registo de cliente, por exemplo emendar o balanço corrente, poderia ser utilizado o seguinte procedimento:

```
ACTUALIZAR-REG.
        MOVE  NUM-CONTA  TO  FICH-CLIENTE-CHAVE
        READ  FICH-CLIENTE
                INVALID  KEY  PERFORM  REG-INEXISTENTE.
        PERFORM  EMENDAR-BAL
        REWRITE  REG-CLIENTE
                INVALID  KEY  PERFORM  ACTUAL-FALHADA.
```

A primeira parte deste procedimento é igual a ENCONTRAR-REG, definido acima. Se um registo for lido com êxito no ficheiro, então o procedimento EMENDAR-BAL é executado para modificar o registo e o mesmo já emendado e de novo escrito no ficheiro, utilizando a instrução REWRITE. A menos que o programador faça qualquer coisa que altere o valor de FICH-CLIENTE-CHAVE entre o READ e o REWRITE, o procedimento ACTUAL-FALHADA nunca será executado.

Pode ser criado um novo registo de cliente através da utilização da instrução WRITE. Por exemplo:

```
ADICIONA-CLIENTE.
        PERFORM  CRIAR-NOVO-CLIENTE
        MOVE  NUN-CONTA-NOVO  TO  FICH-CLIENTE-CHAVE
        WRITE  REG-CLIENTE  FROM  REG-NOVO-CLIENTE
                INVALID  KEY  PERFORM  CRIACAO-FALHADA.
```

É aceite que o procedimento CRIAR-NOVO-CLIENTE reconstrói um novo registo de cliente em REG-NOVO-CLIENTE. A instrução MOVE coloca então o número relativo de registo em FICH-CLIENTE-CHAVE pronto para a instrução WRITE. Os conteúdos de REG-NOVO-CLIENTE são nessa altura escritos em FICH-CLIENTE na posição especificada, a não ser que a área já esteja ocupada, desencadeando-se a seguir o procedimento CRIACAO-FALHADA.

Um registo de cliente pode ser apagado por um procedimento do seguinte formato:

```
APAGA-CLIENTE.
        MOVE  NUM-CONTA-APAGA  TO  FICH-CLIENTE-CHAVE
        DELETE  FICH-CLIENTE
                INVALID  KEY  PERFORM  APAGA-FALHADO.
```

Tendo o número relativo de registo sido definido, a área correspondente é marcada como vazia, ou, se já se encontrar vazia, o procedimento APAGA-FALHADO é executado.

Apesar de o exemplo dado utilizar o facto de existir uma correspondência linear entre o número de conta e o número relativo de registo, isto pode não ser verdade. Quaisquer chaves podem ser utilizadas desde que seja possível transformá-las em números relativos de registo algoritmicamente. Isto é similar à transformação de chaves em índices de tabelas, como foi discutido na secção 10.4.

É possível ler um ficheiro relativo sequencialmente, onde os registos são lidos por ordem de número relativo de registo, sendo ignoradas quaisquer áreas que se encontrem vazias. Suponha que o ramo Erewhon Building Society precisa de uma listagem de todas as contas activas de clientes. Poderia então ser escrito um programa que tratasse o ficheiro FICH-CLIENTE como sequencial, que incluiria uma cláusula SELECT com o seguinte formato:

```
SELECT  FICH-CLIENTE  ASSIGN  TO  <nome-externo-ficheiro>
        ORGANIZATION  RELATIVE
        ACCESS  MODE  SEQUENTIAL.
```

O ficheiro FICH-CLIENTE pode agora ser aberto como *input* e tratado exactamente como um ficheiro sequencial, utilizando uma instrução READ com um qualificador AT END.

A ORGANIZATION do ficheiro continua a ser definida como RELATIVE, o que significa que os registos mantêm a sua identificação através de número relativo de registo. No entanto ACCESS MODE SEQUENTIAL indica que os registos serão lidos ou escritos na ordem segundo a qual aparecem no ficheiro, diferente agora da anterior forma aleatória.

11.5 — **FICHEIROS INDEXADOS**

Um ficheiro indexado é um ficheiro no qual cada registo pode unicamente ser identificado por um valor de chave contido no próprio registo. O sistema mantém um índice ou uma hierarquia de índices que irá permitir a um programa aceder aos registos através de um dado valor de chave. O programador não tem de se preocupar com o modo como esta estrutura é mantida.

As declarações de *input/output* que podem ser utilizadas com um ficheiro de *input* são as mesmas estudadas para o ficheiro relativo: READ, WRITE, REWRITE e DELETE. O qualificador de INVALID KEY necessita de novo de ser especificado para lidar com as tentativas de executar operações ilegais no ficheiro. Para o caso do WRITE, REWRITE e DELETE, ocorre uma condição de chave inválida quando não existe nenhum registo no ficheiro com a chave definida, enquanto para o WRITE uma chave inválida ocorre se o registo com uma chave igual à da especificada já existe no ficheiro.

Para ilustrar a utilização do uso de um ficheiro indexado, é dada uma referência ao problema simples colocado na secção 11.3. Isto irá permitir uma comparação directa das utilizações de ficheiros indexados e relativos.

A cláusula SELECT para o ficheiro terá agora de especificar que a organização do ficheiro é INDEXED (indexada). Por exemplo:

```
ENVIRONMENT  DIVISION.
. . . . .
INPUT-OUTPUT  SECTION.
FILE-CONTROL.
        SELECT  FICH-CLIENTE  ASSIGN  TO  <nome-externo-ficheiro>
                ORGANIZATION  INDEXED

                ACCESS  MODE  RANDOM
                RECORD  KEY  CLIENTE-NUM-CONTA.
```

Note que o campo especificado como RECORD KEY deve fazer parte da descrição de registo associada a este ficheiro.

A descrição de ficheiro para FICH-CLIENTE pode ser exactamente a mesma utilizada para o caso do ficheiro relativo, que era:

```
DATA  DIVISION.
FILE  SECTION.
FD      FICH-CLIENTE
        LABEL  RECORDS  STANDARD.
01      REG-CLIENTE.
        05      CLIENTE-NUM-CONTA              PIC 9(10).
```

Na PROCEDURE DIVISION, o ficheiro terá de ser aberto para *input* e *output:*

```
OPEN I-0 FICH-CLIENTE . . . . .
```

A maior diferença entre indexados e relativos ocorre na inicialização de campos antes da execução de uma instrução de *input* ou de *output*. Para a organização de ficheiros indexada, é o valor da chave do registo, CLIENTE-NUM-CONTA, que controla o acesso ao ficheiro. Assim, para se obter um registo relacionado com um cliente particular, poder-se-ia utilizar o seguinte procedimento:

```
ENCONTRAR-REG.
        MOVE  NUM-CONTA  CLIENTE-NUM-CONTA
        READ  FICH-CLIENTE
                INVALID  KEY  PERFORM  REG-INEXISTENTE.
```

A instrução de READ irá encontrar um registo em FICH-CLIENTE com o valor da chave correspondente ao valor transferido de NUM-CONTA para CLIENTE-NUM-CONTA. Se a instrução de READ se efectuar com êxito, então os conteúdos irão ser copiados para a área de *input;* se ocorrer o contrário, será executado o procedimento REG-INEXISTENTE.

Para actualizar um registo existente, o procedimento é o mesmo do ficheiro relativo, exceptuando a inicialização da chave do registo.

```
ACTUALIZAR-REG.
        MOVE  NUM-CONTA  TO  NUM-CONTA-CLIENTE
        READ  FICH-CLIENTE
                INVALID  KEY  PERFORM  REG-INEXISTENTE.
        PERFORM  EMENDAR-BAL
        REWRITE  REG-CLIENTE
                INVALID  KEY  PERFORM  ACTUAL-FALHADA.
```

A primeira parte deste procedimento é igual a ENCONTRAR-REG, definido acima. Se um registo for lido com êxito, então o procedimento EMENDAR-BAL é usado para modificar o registo, sendo o registo emendado reescrito no ficheiro através da instrução REWRITE. A menos que o programador altere o valor de NUM-CONTA-CLIENTE, na área de *input*, entre as instruções de READ e REWRITE, o procedimento ACTUAL-FALHADA nunca será executado.

Pode ser então criado um novo registo de cliente através da instrução WRITE. Por exemplo:

```
ADICIONA-CLIENTE.
        PERFORM  CRIAR-NOVO-CLIENTE
        WRITE  REG-CLIENTE  FROM  REG-NOVO-CLIENTE
                INVALID  KEY  PERFORM  CRIACAO-FALHADA.
```

Convenciona-se que o procedimento CRIAR-NOVO-CLIENTE cria um novo registo de cliente em REG-NOVO-CLIENTE. Este novo registo, em REG-NOVO-CLIENTE, deveria conter um número de conta com o mesmo formato de NUM-CONTA-CLIENTE, de modo que a instrução WRITE . . . FROM irá implicitamente inicializar a chave do registo na área de *output*. Se já existir um registo com esta chave, então a instrução de WRITE irá falhar, sendo executado o procedimento CRIACAO-FALHADA.

Um registo de cliente pode ser apagado por um procedimento do seguinte formato:

```
APAGA-CLIENTE.
        MOVE  NUM-CONTA-APAGA  TO  NUM-CONTA-CLIENTE
        DELETE  FICH-CLIENTE
                INVALID  KEY  PERFORM  APAGA-FALHADO.
```

Se não existir nenhum registo no ficheiro com uma chave correspondente ao valor transferido de NUM-CONTA-APAGA, então o procedimento APAGA-FALHADO será executado.

É possível ler um ficheiro indexado sequencialmente, em que os registos são lidos por valor ascendente de chave. Suponha que o ramo Erewhon Bulding Society precisa de uma listagem de todos as contas activas de clientes. Poderia então ser escrito um programa que tratasse o ficheiro FICH-CLIENTE como sequencial, que incluiria uma cláusula SELECT com o seguinte formato:

```
SELECT  FICH-CLIENTE  ASSIGN  TO  <nome-externo-ficheiro>
        ORGANIZATION  INDEXED
        ACCESS  MODE  SEQUENTIAL
        RECORDS  KEY  NUM-CONTA-CLIENTE.
```

Note que a chave do registo continua a ser especificada para o acesso sequencial, pois a chave define a ordenação do ficheiro. O ficheiro FICH-CLIENTE pode agora ser aberto como *input* e tratado exactamente como um ficheiro sequencial, utilizando uma instrução READ com um qualificador AT END.

Do ponto de vista do programador, os ficheiros indexados são mais fáceis de utilizar do que os realtivos, porque o tratamento das chaves e as suas correspondências com os registos é feita pelo sistema e não dentro do programa. A outra razão por que os ficheiros indexados são mais utilizados é porque as chaves dos registos são de qualquer tipo e não precisam de se encontrar entre limites contíguos de valores. Assim, um ficheiro pode ser indexado através de uma chave alfabética, alfanumérica ou uma colecção aleatória de chaves numéricas, as quais são difíceis de transformar em número relativo de registo para um ficheiro relativo.

Seria possível utilizar um nome como chave para um ficheiro de fornecedores, por exemplo, desde que cada nome identificasse só um registo. O problema que se põe acerca dos nomes é que não pode garantir-se, para o caso particular de nomes pessoais, que este seja chave única no ficheiro. Assim, os ficheiros são acedidos na generalidade através de chaves construídas, tais como números de conta de cliente, números de contribuinte, números que garantidamente só identificam uma conta ou uma pessoa, respectivamente.

No entanto, os ficheiros relativos não devem ser esquecidos. Um ficheiro indexado tem uma sobrecarga inerente em necessidades de espaço e tempo gasto no acesso a um registo particular por causa das estruturas do índice e da necessidade de sistematicamente se pesquisar o mesmo. Assim, os ficheiros relativos são utilizados quando se necessita de um acesso rápido e a conversão das chaves em números relativos de registo podem ser eficientemente efectuadas dentro do programa.

Esta conversão pode ser conseguida através da utilização dos métodos discutidos na secção 10.4. Também podem utilizar-se «algoritmos de mistura» conjuntamente com uma técnica de tratamento de colisões entre sinónimos, onde o limite de chaves é muito grande. A maior parte dos livros acerca de estruturas de dados incluem secções acerca da codificação misturada e os problemas das colisões entre sinónimos.

Apêndice 1

O ESQUELETO DE PROGRAMA COBOL

Os exemplos e exercícios nos primeiros oito capítulos deste livro baseiam-se no esqueleto de programa *standard*. Este esqueleto foi criado de forma a esconder algumas particularidades do COBOL irrelevantes para o principiante.

Existe um alicerce predeterminado para todos os programas de COBOL dados no apêndice 4. O esqueleto de programa dado neste apêndice vai para além dessas regras e impõe uma estrutura adicional no programa do principiante. Assim, deve ser salientado que este esqueleto de programa não é uma estrutura para todos os programas de COBOL, mas somente uma conveniente ajuda de ensino neste livro. Programas mais avançados de COBOL, que explorem todas as suas potencialidades, terão estruturas diferentes dentro dos parâmetros-base dum programa.

O esqueleto fornecido abaixo é baseado num programa simples de COBOL com um ficheiro de *input* sequencial e um ficheiro de *output* também sequencial, cada um com um só registo associado.

Os objectos entre parênteses angulares, «<» e «>», são para ser substituídos por nomes apropriados aquando da geração de um programa particular. No fim deste apêndice são fornecidas notas para a escolha destas substituições.

```
IDENTIFICATION  DIVISION.
PROGRAM-ID.  <nome-programa>.

ENVIRONMENT  DIVISION.
CONFIGURATION  SECTION.
SOURCE-COMPUTER.  <nome-computador>.
OBJECT-COMPUTER.  <nome-computador>.
*       <nome-computador> deve ser substituído pelo nome
*       do computador usado para correr programas.

INPUT-OUTPUT  SECTION.
FILE-CONTROL.
        SELECT  <nome-ficheiro-input>  ASSIGN  TO  <sistema-input>.
        SELECT  FICHOUT  ASSIGN  TO  <sistema-output>.
*       <sistema-input> e <sistema-output> devem estar conformes
*       com as regras do sistema utilizado.

DATA  DIVISION.
FILE  SECTION.
FD      <nome-ficheiro-input>
        LABEL  RECORDS  OMITTED.
*       Neste espaço deve ser inserida
*       a descrição do registo de input.
FD      FICHOUT
```

```
        LABEL  RECORDS  OMITTED.
01      LINHA-OUT                                  PIC X(120).

WORKING-STORAGE  SECTION.
01      VECTOR-ESTADO.
        05      FIM-<nome-ficheiro-input>          PIC X.
        88      FIM-DE-<nome-ficheiro-input>                VALUE «E».
*       Devem ser aqui inseridas descrições do registo
*       ou registos de output, cabeçalhos ou
*       campos temporários.

PROCEDURE  DIVISION.
PROGRAMA-PRINCIPAL.
        PERFORM  ESTADO-INIC
        PERFORM  <processamento-principal>
        PERFORM  FECHO.
ESTADO-INIC.
        MOVE  SPACES  TO  FIM-<nome-ficheiro-input>
        OPEN  INPUT  <nome-ficheiro-input>  OUTPUT  FICHOUT
        MOVE  SPACES  TO  <nome-registo>.
FECHO.
        CLOSE  <nome-ficheiro-input>  FICHOUT
        STOP  RUN.
LER-PROXIMO-<nome-ficheiro-input>.
        READ  <nome-ficheiro-input>
                AT END MOVE «E» TO FIM-<nome-ficheiro-input>.
ESCREVER-<nome-registo>.
        WRITE  LINHA-OUT  FROM  <nome-registo>
        MOVE  SPACES  TO  <nome-registo>.

<processamento  principal>.
*       Desenvolvimento do algoritmo principal para
*       o programa a partir deste ponto.
```

Notas

1. O <nome-programa> identifica o programa *source* (fonte) e é um nome definido pelo utilizador (ver 3.2). No entanto, o sistema de computador que está a ser utilizado pode colocar algumas restrições ao formato desse mesmo nome.
2. O <nome-computador> identifica o sistema de computador em uso para a compilação e execução do programa. O seu formato irá obviamente variar de sistema para sistema.
3. O <sistema-input> e o <sistema-output> são também particulares do sistema de computador. Eles serão referências aos ficheiros actuais utilizados para *input* e *output* pelo programa.
4. As referências a <nome-ficheiro-input> devem ser todas substituídas pelo nome escolhido pelo programador para o ficheiro de *input*.
5. As referências a <nome-registo> devem todas ser substituídas pelo nome escolhido para o registo de *output* no programa.
6. Se o programa utilizar registos de *output* múltiplos que contenham dados variáveis, então cada tipo de registo terá de ter um parágrafo ESCREVER-<nome-registo> para cada um, no formato fornecido acima. O <nome-registo> deve ser também adicionado ao fim de ESTADO-INIC por uma linha do seguinte formato:

```
MOVE  SPACES  TO  <nome-registo-1>  <nome-registo-2>
        . . .  <nome-registo-n>.
```

A utilização de registos múltiplos de *output* foi analisada na secção 5.5.

7. Para se fazer o *output* de uma linha de constantes, tal como um cabeçalho, ou uma mistura de dados constantes e variáveis, o parágrafo ESCREVER-<nome-registo> é simplificado para:

```
ESCREVER-<nome-registo>.
        WRITE  LINHA-OUT  FROM  <nome-registo>.
```

porque o conteúdo de <nome-registo> não precisa de ser inicializado para uso posterior. Um registo que contenha dados constantes não deve ser inicializado em ESTADO-INIC porque isso irá destruir o seu conteúdo.

O uso de linhas de constantes é comentado na secção 5.6.

Apêndice 2

FORMATO DO PROGRAMA COBOL

O formato segundo o qual as declarações COBOL devem ser preparadas para serem enviadas para o compilador está definido rigidamente. Pode considerar-se que qualquer linha consiste em quatro partes, que estão sumariadas no diagrama abaixo.

As letras no topo do diagrama definem as «margens» onde «E», margem esquerda, e «D», margem direita, definem os limites da linha.

Os primeiros seis caracteres da linha estão reservados para o número de sequência. O sétimo carácter é chamado «área do indicador». Dos caracteres 8 até ao 11 temos a área A, sendo o espaço do carácter 12 até à margem direita chamado «área b».

Convenções sobre margens

1. Cabeçalhos de divisão e secção devem estar em área A.
2. Os nomes de parágrafo devem também começar em área A. A primeira instrução do parágrafo deve continuar na mesma linha ou na seguinte, mas em área B. Todas as instruções subsequentes devem aparecer em área B. Essas linhas podem ser indentadas da margem B para mostrar a estrutura do programa.
3. O FD duma descrição deve começar em área A, seguido de pelo menos um espaço em branco, o nome do ficheiro e a restante descrição em área B.
4. Numa descrição de registo, o número de nível 01 deve começar em área A e ser seguido de, pelo menos, um espaço em branco, estando a restante descrição em área B. Todos os outros itens de grupo ou elementares com número de nível superior a 01 devem ter, pelo menos, um espaço a seguir ao número de nível e devem ser indentados para mostrar a estrutura do registo.

Separadores

O separador principal usado entre os elementos da linguagem COBOL é o espaço. Neste contexto, o elemento é uma unidade indivisível do programa. Em qualquer zona do programa onde seja permitido um espaço é permitido um número qualquer de espaços.

Por exemplo, considere:

```
IF PAG-TRIBUTADO > 500
```

A palavra reservada IF, o nome de campo PAG-TRIBUTADO, o comparador «>» e o literal numérico 500 são todos elementos da linguagem e devem estar separados por um ou mais espaços. Exceptuando o caso dos literais não numéricos (ver «Linhas de continuação abaixo), uma interrupção numa declaração por motivo de fim de linha é considerada como espaço separador.

O exemplo acima poderia então ser reescrito da seguinte forma:

```
IF      PAG-TRIBUTADO      >      500
```

ou mesmo:

```
IF PAG-TRIBUTADO
      > 500
```

Não seria legal escrever:

```
IF PAG-TRIBUTADO>500
```

porque não existe separador entre PAG-TRIBUTADO e «>», ou entre «>» e 500. Reescrevendo o exemplo acima como:

```
IF PAG
-TRIBUTADO > 500
```

seria também ilegal, pois o campo PAG-TRIBUTADO foi separado por uma linha em branco.

As marcas de pontuação, «.» e «,», podem também ser utilizadas como separadores, mas cada uma delas tem de ser seguida de, pelo menos, um espaço antes do próximo elemento. Note que esta referência ao carácter «.» é para quando este é utilizado na pontuação e não como ponto decimal numa cláusula PIC ou literal.

Os únicos outros separadores que terão sido encontrados neste livro são os parênteses esquerdo e direito, «(» e «)», que podem aparecer em expressões aritméticas, condições e nomes subscritados. Sem um envolvimento muito grande em pormenores, regras seguras a seguir são:

a) O parênteses esquerdo deve ser precedido de um ou mais espaços, mas não precisa de ser seguido de nenhum;

b) O parênteses direito não necessita de ser precedido de nenhum espaço, mas deve ser seguido de um ou mais espaços, a não ser que seja um ponto final.

Linhas de continuação

Qualquer elemento da linguagem COBOL pode ser continuado em uma ou mais linhas, que devem ser todas em área B, independentemente do posicionamento desta.

Um hífen (-) na área do indicador indica que o primeiro carácter diferente de espaço da área B da linha de continuação se segue imediatamente após o último carácter diferente de espaço da linha precedente, excluindo linhas de comentário intervenientes. A única excepção a esta regra são as constantes não numéricas, que são partidas em duas ou mais linhas, utilizando as seguintes regras:

1. Qualquer linha que seja para continuar não deve estar entre aspas e quaisquer espaços no fim da linha são considerados como partes do literal.

2. A linha de continuação deve ter um hifen na área do indicador, a área A deve estar a espaços e o primeiro carácter diferente de espaço da área B deve ser aspas.
3. É convencionado que o primeiro carácter após a aspa de abertura da linha de continuação se segue imediatamente após o último carácter da linha precedente, excluindo linhas de comentário intervenientes.

Como o número de espaços usados como separador não é significante, a continuação de constantes não numéricas é o único caso em que um indicador de continuação é absolutamente necessário.

Comentários

Qualquer linha que tenha um asterisco na área do indicador é tratada como comentário, sendo ignorada para tudo menos listagens do programa.

Apêndice 3

PALAVRAS RESERVADAS

A seguinte lista fornece as palavras reservadas definidas para o *ANS COBOL '74.* Posteriores palavras reservadas só serão adicionadas para uma dada implementação do COBOL; dessa maneira terá de ser consultado o manual apropriado.

ACCEPT
ACCESS
ADD
ADVANCING
AFTER
ALL
ALPHABETIC
ALSO
ALTER
ALTERNATE
AND
ARE
AREA
AREAS
ASCENDIG
ASSIGN
AT
AUTHOR

BEFORE
BLANK
BLOCK
BOTTOM
BY

CALL
CANCEL
CD
CF
CH
CHARACTER
CHARACTERS
CLOCK-UNITS
CLOSE
COBOL
CODE
CODE-SET
COLLATING
COLUMN
COMMA
COMMUNICATION
COMP
COMPUTATIONAL
COMPUTE
CONFIGURATION
CONTAINS
CONTROL
CONTROLS
COPY
CORR
CORRESPONDING
COUNT
CURRENCY

DATA
DATE
DATE-COMPILED
DATE-WRITTEN
DAY
DE
DEBUG-CONTENTS
DEBUG-ITEM
DEBUG-LINE
DEBUG-NAME
DEBUG-SUB-1
DEBUG-SUB-2
DEBUG-SUB-3
DEBUGGING
DECIMAL-POINT
DECLARATIVES
DELETE
DELIMITED
DELIMITER
DEPENDING
DESCENDING
DESTINATION
DETAIL
DISABLE
DISPLAY
DIVIDE
DIVISION
DOWN
DUPLICATES
DYNAMIC

EGI
ELSE
EMI
ENABLE
END
END-OF-PAGE
ENTER
ENVIRONMENT
EOP
EQUAL
ERROR
ESI
EVERY
EXCEPTION
EXIT
EXTEND

FD
FILE
FILE-CONTROL

FILLER
FINAL
FIRST
FOOTING
FOR
FROM

GENERATE
GIVING
GO
GREATER
GROUP

HEADING
HIGH-VALUE
HIGH-VALUES

I-O
I-O-CONTROL
IDENTIFICATION
IF
IN
INDEX
INDEXED
INDICATE
INITIAL
INITIATE
INPUT
INPUT-OUTPUT
INSPECT
INSTALLATION
INTO
INVALID
IS

JUST
JUSTIFIED

KEY

LABEL
LAST
LEADING
LEFT
LENGHT
LESS
LIMIT
LIMITS
LINAGE
LINAGE-COUNTER
LINE
LINE-COUNTER
LINES
LINKAGE
LOCK

LOW-VALUE
LOW-VALUES

MEMORY
MERGE
MESSAGE
MODE
MODULES
MOVE
MULTIPLE
MULTIPLY

NATIVE
NEXT
NO
NOT
NUMBER
NUMERIC

OBJECT-COMPUTER
OCCURS
OF
OFF
OMITTED
ON
OPEN
OPTIONAL
OR
ORGANIZATION
OUTPUT
OVERFLOW

PAGE
PAGE-COUNTER
PERFORM
PF
PH
PIC
PICTURE
PLUS
POINTER
POSITION
POSITIVE
PRINTING
PROCEDURE
PROCEDURES
PROCEED
PROGRAM
PROGRAM-ID

QUEUE
QUOTE
QUOTES
RANDOM
RD

READ
RECEIVE
RECORD
RECORDS
REDEFINES
REEL
REFERENCES
RELATIVE
REMAINDER
REMOVAL
RENAMES
REPLACING
REPORT
REPORTING
RERUN
RESERVE
RESET
RETURN
RESERVED
REWIND
REWRITE
RF
RN
RIGHT
ROUNDED

SAME
SD
SEARCH
SECTION
SECURITY
SEGMENT
SEGMENT-LIMIT
SELECT
SEND
SENTENCE
SEPARATE
SEQUENCE
SEQUENTIAL
SET
SIGN
SIZE
SORT
SORT-MERGE
SOURCE
SOURCE-COMPUTER
SPACE
SPACES
SPECIAL-NAMES
STANDARD
STANDARD-1
START
STATUS
STOP
STRING

SUB-QUEUE-1
SUB-QUEUE-2
SUB-QUEUE-3
SUBTRACT
SUPPRESS
SYMBOLIC
SYNC
SYNCHRONIZED

TABLE
TALLYNG
TAPE
TERMINAL
TERMINATE
TEXT
THAN
THROUGH
THRU
TIME
TIMES
TO
TOP
TRAILING
TYPE

UNIT
UNSTRING
UNTIL
UP
UPON
USAGE
USE
USING

VALUE
VALUES

WHEN
WITH
WORDS
WORKING-STORAGE
WRITE

ZERO
ZEROES
ZEROS

Apêndice 4

SUMÁRIO DA SINTAXE DE COBOL

O objectivo deste apêndice é juntar todas as construções de COBOL usadas neste livro, sumariando a sua sintaxe. Nalguns casos, especialmente *pictures,* é difícil executar um sumário conciso, e aqui deve o leitor referir-se às secções relevantes do texto principal. Em todos os casos serão dadas referências cruzadas à discussão principal de uma característica no texto principal. Deve salientar-se que o que aqui se apresenta é uma pequena subdivisão do COBOL, suficiente só para escrever os programas fornecidos neste livro.

A notação utilizada neste sumário de sintaxe é a utilizada nos documentos ANS COBOL *standards* e também na maioria dos manuais de COBOL. As palavras maiúsculas sublinhadas são palavras-chave e devem aparecer se uma construção particular for utilizada. As outras palavras também em maiúsculas mas não sublinhadas são opcionais e podem ser omitidas sem alterar o significado da construção. Palavras em minúsculas indicam referências a outras definições.

As alternativas são indicadas por uma chaveta de objectos, dos quais se deve escolher um. Parênteses rectos indicam que o conteúdo que encerram é opcional, mas se for incluído irá alternar o significado da construção. As reticências, «. . .», indicam que o objecto precedente pode ser repetido tantas vezes quantas as requeridas. Os parênteses curvos são utilizados para encerrar um item obrigatório, usualmente uma escolha de coisas ou um objecto composto que pode ser repetido.

Nome definido por utilizador

Um nome definido por utilizador é uma sequência de não mais de trinta caracteres escolhidos de A a Z, 0 a 9 e «-» (hífen), não começando ou acabando em hífen. Não pode ser também uma palavra reservada de COBOL (ver apêndice 3) e cada nome definido por utilizador deve unicamente estar associado a uma só entidade do programa.

Ver 3.2.

Literal

Um literal é uma constante utilizada num programa e pode ser de três tipos:

a) Literal numérico, um inteiro ou valor decimal, possivelmente precedido de um sinal;
b) Literal não número; qualquer sequência de caracteres entre aspas (« »);
c) Literal figurativo, SPACE[S], HIGH-VALUE[S]. LOW-VALUE[S].

Ver 4.2 (Constantes).

Divisão de identificação

```
IDENTIFICATION DIVISION
PROGRAM-ID. nome-programa.
[AUTHOR. Entrada comentario]
[INSTALLATION. entrada-comentario]
[DATE-WRITTEN. entrada-comentario]
[DATE-COMPILED. entrada-comentario]
[SECURITY. entrada-comentario]
```

onde nome-programa é um nome definido pelo utilizador, possivelmente com as restrições próprias de uma implementação particular, e entrada-comentário é qualquer cadeia de caracteres de grupo permitido pelo COBOL.

Ver 81.

Divisão do meio envolvente («Environment»)

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. nome-computador-fonte.
OBJECT-COMPUTER. nome-computador-objecto.
[SPECIAL-NAMES.
        nome-implementador IS nome-mnemónica.]
INPUT-OUTPUT SECTION.
FILE-CONTROL.
        entrada-controlo-ficheiro . . .
```

onde nome-computador-fonte, nome-computador-objecto e nome-implementador devem ser substituídos por uma cadeia de caracteres definidos pelo implementador ou pelo sistema em uso; nome-mnemónica é um nome de utilizador.

Ver 8.2.

Entrada de controlo de ficheiros

```
SELECT nome-ficheiro ASSIGN TO nome-externo-ficheiro.
```

onde nome-ficheiro é um nome de utilizador e o formato de nome-externo-ficheiro é definição do implementador.

Ver: 8.2 e 9.1.

Divisão de dados

```
DATA DIVISION.
[FILE SECTION.
        entrada-descrição-ficheiro . . .]
[WORKING-STORAGE SECTION.
        entrada-descrição-registo . . .]
[LINKAGE SECTION.
        entrada-descrição-registo...]
```

Ver 8.3 e 11.2.

Entrada de descrição de ficheiro

```
FD  nome-ficheiro
                        { STANDARD }
LABEL  RECORDS          { OMITTED. }
        entrada-descrição-registo...
```

onde nome-ficheiro é um nome de utilizador.
Ver: 8,3 e 9.1.

Entrada de descrição de registo

```
01      nome-dados
        [PIC  cadeia-moldura].
```

onde nome-dados é um nome de utilizador e cadeia-moldura é um formato correcto de caracteres que descrevem um campo de dados.
As *pictures* (molduras) são analisadas em 3.4, 5.1, 5.2 e 5.3.
Ver 3.3.

Entrada de descrição de dados

```
                  { nome-dados }
numero-nivel      { FILLER     }
[REDEFINES  nome-dados
[PIC  cadeia-moldural].
[OCCURS  numero inteiro  TIMES]
[VALUE  literal].
```

onde numero-nivel é um número inteiro entre 02 e 49, nome-dados é um nome de utilizador e cadeia-moldura é uma cadeia correcta de caracteres que descrevem um campo de dados. A cláusula VALUE só pode ser associada a uma entrada de descrição de dados na WORKING-STORAGE SECTION.
As *pictures* (molduras) são discutidas em 3.4, 5.1, 5.2 e 5.3.
Ver 3.3, 5,4 (VALUE), 6.2 (REDEFINES) e 10.1 (OCCURS).

Nomes-condição

```
88      nome-condição VALUE { literal [THRU literal] } . . . .
```

onde nome-condição é um nome de utilizador; o nome-condição é associado ao item elementar imediatamente precedente e cada literal deve ser do mesmo tipo do item elementar.
Ver 6.3.

Índices

```
                  { literal     }     [,  { literal     }  ]... )
nome-tabela (     { nome-dados  }         { nome-dados  }
```

onde nome-tabela é o nome de um item de grupo ou item elementar que é sujeito a uma ou mais cláusulas OCCURS. Qualquer literal deve ser um número inteiro e nome-dados deve estar associado a um campo de tipo inteiro não editado nem subscritado.

Ver 10.1 (e seguintes secções do capítulo 10).

Identificador

Um identificador é um nome utilizado somente para referenciar um único campo de dados. Se o campo faz parte de uma tabela, então o nome-dados necessitará de um índice. Em geral, um nome-dados pode também ser indexado ou qualificado, mas esses conceitos estão para além dos propósitos deste livro.

Condição

```
identificador [NOT]  { >  }  literal
                     { =  }  identificador
                     { <  }

[NOT] nome-condição

identificador [NOT]  { NUMERIC    }
                     { ALPHABETIC }
```

Condições simples completas, como as dadas acima, podem ser combinadas em condições compostas utilizando AND e OR juntamente com parênteses, se necessário.

Ver 4.3 (Condições simples), 6.2 (Condições de classe), 6.3 (Nomes-condição) e 6.5 (Condições compostas).

Divisão de procedimento

```
PROCEDURE DIVISION [USING identificador . . . ] .
        (nome-paragrafo. frase . . . ) . . .
```

onde nome-parágrafo é um nome de utilizador e frase é uma sequência de instruções COBOL terminadas com um ponto final.

Sequência imperativa

É uma sequência de instruções que podem ser executadas sem que nenhuma futura estrutura condicional seja introduzida. Deste modo, a declaração IF não pode ser usada como parte de uma sequência imperativa, assim como a declaração READ com um qualificador AT END (ou um qualificador INVALID KEY). Esta regra é imposta para evitar a ambiguidade numa sequência de declarações.

Uma sequência imperativa é normalmente terminada por um ponto final, mas pode terminar com um ELSE se colocado dentro duma declaração IF. Na prática, a maioria das sequências imperativas consiste em declarações únicas, tais como o MOVE ou o PERFORM.

Declaração «Call» (chamar)

CALL nome-programa [USING identificador . . .]

onde nome-programa é um literal não numérico.
Ver 11.2.

Declaração «Close» (fechar)

CLOSE nome-ficheiro . . .

onde nome-ficheiro é um nome de utilizador que aparece nas entradas de controlo de ficheiros e descrição de ficheiros.
Ver 8.4 e 9.1.

Declaração «Compute» (calcular)

COMPUTE identificador [ROUNDED] = expressão-aritmetica

onde identificador está associado com um campo de dados de tipo numérico.
A formulação de expressões-aritméticas é analisada na secção 2.1.
Ver 4.2 (Operação de cálculo) e 5.1 (Arredondamento).

Declaração «Display» (mostrar)

DISPLAY { literal / identificador } . . . [UPON nome-mnemonica]

onde nome-mnemonica é um nome de utilizador que deve previamente ter sido especificado no parágrado SPECIAL-NAMES.
Ver 7.2.

Declaração de saída de programa

EXIT PROGRAM.

onde esta declaração deve ser única no parágrafo.
Ver 11.2.

Declaração «GO TO» (ir para)

GO TO nome-paragrafo

onde nome-paragrafo é um nome de utilizador que define um parágrafo na PROCEDURE DIVISION.
Ver 11.1.

Declaração «If» (se)

IF sequencia condição-declaração [ELSE condição-declaração]

onde condição-declaração é qualquer sequência de uma ou mais declarações COBOL.
Ver 4.3 (Selecção) e 6.4.

Declaração «Move» (mover)

MOVE { literal / identificador } TO identificador . . .

Ver 4.2 (Operações de transferência).

Declaração «Open» (abrir)

OPEN INPUT nome-ficheiro . . . OUTPUT nome-ficheiro . . .

onde nome-ficheiro é um nome de utilizador que aparece nas entradas de controlo e definição de ficheiros.
Ver 8.4 e 9.1.

Declaração «Perform» (execute)

PERFORM nome-paragrafo

onde nome-paragrafo é um nome de utilizador que identifica um parágrafo na PROCEDURE DIVISION.
Ver 4.2 (Refinamento).

Declaração «Perform-until» (execute até)

PERFORM nome-paragrafo UNTIL condição

onde nome-paragrafo é um nome de utilizador que identifica um parágrafo na PROCEDURE DIVISION.
Ver 4.3 (Repetição).

Declaração «Perform-varying» (execute variando)

PERFORM nome-paragrafo VARYING identificador

FROM { literal / identificador } BY { literal / identificador } UNTIL condição

onde nome-paragrafo é um nome de utilizador que identifica um parágrafo na PROCEDURE DIVISION. Quaisquer literais devem ser de tipo numérico e os identificadores devem estar associados com campos de dados de tipo numérico, não editados.

(Esta declaração é utilizada mais vulgarmente conjuntamente com tabelas em que os literais e identificadores são restritos a valores inteiros.)

Ver 10.1 (Ciclo for).

Declaração «Read» (ler)

READ nome-ficheiro [INTO identificador] AT END sequencia-imperativa

onde nome-ficheiro é um nome de utilizador que aparece nas entradas de controlo e definição de ficheiros. O identificador deve especificar um campo que é independente da descrição do ficheiro (normalmente, uma descrição de registo da WORKING-STORAGE SECTION).

Ver 8.4, 8.6 e 9.1.

Declaração «Stop» (parar)

STOP { literal / RUN }

Ver 8.4.

Declaração «Write» (escrever)

WRITE nome-registo [FROM identificador]

onde nome-registo é um nome de uma descrição de registo que aparece na descrição de ficheiro e identificador especifica um campo que é independente da descrição do ficheiro (normalmente, uma descrição de registo da WORKING-STORAGE SECTION).

Ver 8.4 e 9.1.

Declaração «Write» (impressora)

WRITE nome-registo [FROM identificador]

{ BEFORE / AFTER } ADVANCING { inteiro LINES / PAGE }

onde nome-registo é um nome de uma descrição de registo que aparece na descrição de ficheiro e identificador especifica um campo que é independente da descrição do ficheiro (normalmente, uma descrição de registo da WORKING-STORAGE SECTION).

Para um ficheiro de *output* particular, é de aconselhar o uso ou do BEFORE ou do AFTER isoladamente em vez de os misturar.

Ver 8.5.

Ficheiros de acesso directo

O uso de estruturas de ficheiros de acesso directo, indexado ou relativo, requer um formato alternativo da entrada de controlo de ficheiros e várias declarações de *input/output*. Estas são dadas separadamente para evitar a inclusão de demasiadas alternativas na sintaxe para os formatos básicos das declarações fornecidas acima.

Entrada controlo de ficheiro (ficheiros relativos)

```
SELECT nome-ficheiro ASSIGN TO nome-externo-ficheiro

ORGANIZATION RELATIVE
                      { RANDOM     }
ACCESS MODE           { SEQUENTIAL }
[RELATIVE KEY nome-dados].
```

onde nome-ficheiro e nome-dados são nomes de utilizador e o formato de nome-externo-ficheiro é definição do implementador. A cláusula RELATIVE KEY é requerida para ACCESS MODE RANDOM. O nome-dados deve estar associado com um campo de dados de tipo numérico inteiro que não deve estar contido dentro da descrição de registo associada ao nome-ficheiro.

Ver 11.4.

Entrada controlo de ficheiro (ficheiros indexados)

```
SELECT nome-ficheiro ASSIGN TO nome-externo-ficheiro

ORGANIZATION INDEXED
                      { RANDOM     }
ACCESS MODE           { SEQUENTIAL }
[RECORD KEY nome-dados].
```

onde nome-ficheiro e nome-dados são nomes de utilizador e o formato de nome-externo-ficheiro é definição do implementador. Note que a cláusula RECORD KEY é requerida em ambos os casos. O nome-dados deve estar associado a um campo de dados descrito como parte de uma descrição de registo ao nome-ficheiro.

Ver 11.5.

Declaração «Delete» (apagar)

```
DELETE nome-ficheiro INVALID KEY sequencia-imperativa
```

onde nome-ficheiro é um nome de utilizador que aparece nas entradas de controlo e descrição de ficheiros.

Ver 11.4 e 11.5.

Declaração «Open» (abrir) (acesso directo)

```
                 { I-O    nome-ficheiro . . . }
OPEN             { OUTPUT nome-ficheiro . . . } . . .
                 { INPUT  nome-ficheiro . . . }
```

onde nome-ficheiro é um nome de utilizador que aparece nas entradas de controlo e descrição de ficheiros.

Ver 11.4 e 11.5.

Declaração «Read» (ler) (acesso directo)

READ nome-ficheiro [INTO identificador]
INVALID KEY sequencia-imperativa

onde nome-ficheiro é um nome de utilizador que aparece nas entradas de controlo e definição de ficheiros. O identificador deve especificar um campo que é independente da descrição do ficheiro (normalmente, uma descrição de registo da WORKING-STORAGE SECTION). Este formato só é válido quando a entrada de controlo de ficheiros especifica ACCESS MODE RANDOM.

Ver 11.4 e 11.5.

Declaração «Rewrite» (reescrever)

REWRITE nome-registo [FROM identificador]
INVALID KEY sequencia-imperativa

onde nome-registo é um nome de uma descrição de registo que aparece na descrição de ficheiros e o identificador deve especificar um campo que é independente da descrição do ficheiro (normalmente, uma entrada de descrição de registo na WORKING-STORAGE SECTION).

Ver 11.4 e 11.5.

Declaração «Write» (acesso directo)

WRITE nome-registo [FROM identificador]
INVALID KEY sequencia-imperativa

onde nome-registo é um nome de uma descrição de registo que aparece na descrição de ficheiros e o identificador deve especificar um campo que é independente da descrição do ficheiro (normalmente, uma entrada de descrição de registo da WORKING-STORAGE SECTION). Este formato só é válido quando a entrada de controlo de ficheiros especifica ACCESS MODE RANDOM.

Ver 11.4 e 11.5.

Declaração «Copy» (copiar)

COPY entrada-nome-biblioteca
[REPLACING { literal / identificador } BY { literal / identificador } ...].

onde entrada-nome-biblioteca é um nome de utilizador que identifica um pedaço de texto numa biblioteca de *sources* de COBOL.

Ver 11.3.

Apêndice 5

NOTAS PARA UTILIZADORES DE ANS COBOL DE BAIXO NÍVEL

O American National Standard para o COBOL está estruturado por forma a que uma variedade de compiladores COBOL de baixo nível possa ser produzida dependendo do tamanho do computador a ser usado. Um ANS COBOL mínimo, por exEmplo para um microcomputador, não irá implementar todas as características descritas neste livro. Assim, este apêndice é desenhado para ajudar esses leitores que possam utilizar um tal sistema da baixo nível.

A5.1 — ARITMÉTICA

A declaração COMPUTE pode não estar implementada em todos os compiladores ANS COBOL. Neste caso, o programador irá usar as declarações aritméticas simples: ADD, SUBTRACT, MULTIPLY e DIVIDE. A sintaxe dessas declarações é dada abaixo:

```
                 { literal      }   literal
ADD              { identificador }   identificador   . . .   TO identificador [ROUNDED]

                 { literal      }  { literal      }  [ literal      ]
ADD              { identificador }  { identificador }  [ identificador ]. . .
                 GIVING identificador [ROUNDED]

                       { literal      }  { literal      }
SUBTRACT               { identificador }  { identificador }. . . FROM identificador
                 [ROUNDED]

                       { literal      }  [ literal      ]                literal
SUBTRACT               { identificador }  [ identificador ]. . . FROM     identificador
                 GIVING identificador [ROUNDED]

                       { literal      }
MULTIPLY               { identificador }   BY identificador [ROUNDED]

                       { literal      }                 { literal      }
MULTIPLY               { identificador }   BY            { identificador }
                 GIVING identificador     [ROUNDED]
```

```
DIVIDE { literal      } INTO identificador [ROUNDED]
       { identificador}

DIVIDE { literal      } INTO { literal      }
       { identificador}      { identificador}
        GIVING identificador [ROUNDED]
```

onde todos os identificadores devem referenciar campos do tipo numérico; um identificador que se siga a um GIVING pode referir-se a um campo editado, todos os outros devem referir-se a campos não editados e todos os literais devem ser numéricos. Em MULTIPLY e DIVIDE sem GIVING, o resultado é armazenado no identificador mais à direita. A opção ROUNDED tem o mesmo efeito nos resultados que uma declaração COMPUTE (ver 5.1).

Se o cálculo envolver só um operando, ele pode ser directamente traduzido numa das declarações acima. Por exemplo:

```
calcular det-enc-cob =
        val-alcatifa + val-entrega
```

pode ser traduzido para:

```
ADD VAL-ALCATIFA VAL-ENTREGA
    GIVING DET-ENC-COB
```

Similarmente:

```
calcular val-pagar = 1.125 * val-base
```

pode ser traduzido para:

```
MULTIPLY 1.125 BY VAL-BASE GIVING VAL-PAGAR
```

Para executar cálculos mais complexos será necessário parti-los em duas ou mais operações simples e definir campos para conterem os valores intermédios. Por exemplo:

```
calcular val- base = alug-tel
        + 0.02 * chamadas efec + 0.03 * unid-oper
```

poderia ser traduzido para:

```
MULTIPLY 0.02 BY CHAMADAS-EFEC GIVING RESULT-INTER
ADD ALUG-TEL RESULT-INTER GIVING VAL-BASE
MULTIPLY 0.03 BY UNID-OPER GIVING RESULT-INTER
ADD RESULT-INTER TO VAL-BASE
```

onde RESULT-INTER é um campo temporário agregado a CAMPOS-TEMPORARIOS especificado de tal forma que pode armazenar qualquer resultado intermédio.

A5.2 — NOMES-CONDIÇÃO

Alguns compiladores COBOL de baixo nível podem não implementar nomes-condição de número de nível 88. Como um nome-condição é só uma notação curta, é sempre possível construir uma estrutura de teste equivalente utilizando condições simples de declarações IF encadeadas.

Para implementar a condição de fim-de-<nome-ficheiro> sem utilizar nomes-condição pode ser utilizada a seguinte técnica:

a) Descreva um campo chamado END-OF-<nome-ficheiro> em VECTOR-ESTADO.

```
isto é,  01      STATE-VECTOR.
             05      FIM-DE-FICH-ENCOMENDA      PIC X.
```

b) Inicialize este campo a espaços em ESTADO-INIC.

```
isto é,  ESTADO-INIC.
             MOVE SPACE TO FIM-DE-FICH-ENCOMENDA
             .....
```

c) Coloque o valor do campo em «E» no fim do ficheiro.

```
isto é,  LER-PROXIMO-FICH-ENCOMENDA.
             READ FICH-ENCOMENDA
                  AT END MOVE «E» TO FIM-DE-FICH-ENCOMENDA.
```

d) Mude qualquer condição do formato fim-de-<nome-ficheiro> para um teste explícito para FIM-DE-<nome-ficheiro> = «E» e <u>não</u> FIM-DE-<nome-ficheiro> Not = «E».

```
isto é,  PERFORM PROCESSAR-ENCOMENDA = «E»
             UNTIL FIM-DE-FICH-ENCOMENDA = «E»
         IF FIM-DE-FICH-ENCOMENDA NOT = «E»
```

Os campos boolianos podem ser implementados de maneira análoga utilizando testes explícitos para «V» e «F» no lugar de nomes-condição. Por exemplo, para implementar o campo booliano «pesquisar»:

```
01       VECTOR-ESTADO
         05      PESQUISAR              PIC X.

         mover falso para pesquisar
         MOVE «F» TO PESQUISAR

         mover verdadeiro para pesquisar
         MOVE «V» TO PESQUISAR

         until pesquisar do
         PERFORM <nome-paragrafo>
             UNTIL PESQUISAR = «V»

         if not pesquisar then
         IF PESQUISAR NOT = «V»
         IF PESQUISAR = «F»
```

Nomes-condição associados com valores múltiplos ou gamas de valores, por exemplo em validação de dados (ver 6.3), terão de ser substituídos por uma série de testes explícitos para valores. Por exemplo:

```
             05      TIPO-CUSTO                 PIC A.
             88      CLIENTE-VALIDO             VALUE «C» THRU «H», «X».
.....
         IF CLIENTE-VALIDO
             PERFORM CALCULAR-CONTA.
```

teria de ser substituído por:

```
           05      TIPO-CUSTO                    PIC A.
.....
           IF TIPO-CUSTO > «B»
                   IF TIPO-CUSTO < «I»
                           PERFORM  CALCULAR-CONTA.
                   IF TIPO-CUSTO = «X»
                           PERFORM  CALCULAR-CONTA.
```

A5.3 — «PERFORM-UNTIL» (EXECUTAR-ATÉ)

Num compilador ANS COBOL mínimo só é requerida uma implementação linear do PERFORM; assim, PERFORM . . . UNTIL pode não ser implementado. Neste caso, a operação until terá de ser traduzida usando IF e GO TO.

Uma possível implementação de:

```
until <condição> do
        <procedimento>
enduntil
```

é:

```
UNTIL-<id>-CICLO.
        IF NOT <condição>
                PERFORM  <corpo-ciclo>
                GO  TO  UNTIL-<id>-CICLO.
.....
<corpo-ciclo>.
<procedimento>.
```

onde <id> deve ser substituído por um identificador apropriado para distinguir este ciclo until de quaisquer outros existentes no programa. Por exemplo, o primeiro ciclo until encontrado poderia ter o <id>=1, o segundo <id>=2, etc.

«Perform-thru»

Por azar, a tradução delineada acima introduz um novo problema porque a inserção de um novo nome de parágrafo, UNTIL-<id>-CICLO, altera o fluxo de controlo ao longo do programa. Considere o seguinte desenho de programa de nível superior:

```
encomendas-sumar
        ler-proximo-fich-encomenda
        until  fim-de-fich-encomenda  do
                actualizar-contadores
                ler-proximo-fich-encomenda
        enduntil
        escrever-encomenda-sumario
```

Uma tentativa de tradução seria:

```
PROCEDURE DIVISION.
PROGRAMA-PRINCIPAL.
        PERFORM ESTADO-INIC
        PERFORM SUMAR-ENCOMENDA
        PERFORM FECHO.
.....

SUMAR-ENCOMENDA.
        PERFORM LER-PROXIMO-FICH-ENCOMENDA.
UNTIL-CICLO-1.
        IF NOT FIM-DE-FICH-ENCOMENDA
                PERFORM PROCESSAR-REG-ENCOMENDA
                GO TO UNTIL-CICLO-1.
        PERFORM PRINT-ENCOMENDA-SUMARIO.
PROCESSAR-REG-ENCOMENDA.
        PERFORM ACTUALIZAR-CONTADORES
        PERFORM LER-PROXIMO-FICH-ENCOMENDA.
.....
```

O problema desta tradução é que PERFORM SUMAR-ENCOMENDA só irá executar o parágrafo em questão, SUMAR-ENCOMENDA, e o novo parágrafo chamado UNTIL-CICLO-1 será ignorado. A maneira mais simples de rodear este problema é usar uma variante da declaração PERFORM que permite que uma sequência de parágrafos seja agrupada para execução.
O parágrafo PROGRAMA-PRINCIPAL poderia ser implementado como:

```
PROCEDURE DIVISION.
PROGRAMA-PRINCIPAL.
        PERFORM ESTADO-INIC
        PERFORM SUMAR-ENCOMENDA THRU UNTIL-CICLO-1.
        PERFORM FECHO.
```

A declaração:

```
PERFORM nome-primeiro-paragrafo THRU nome-ultimo-paragrafo
```

significa que as declarações desde o princípio do primeiro parágrafo até ao fim do último são consideradas como uma única unidade por esta declaração PERFORM. Recomenda-se que esta construção seja usada somente nestes casos especiais porque ela tende a obscurecer o significado do programa, a menos que seja utilizada com cuidado.

Por uma questão de estilo, sugere-se que seja introduzido um parágrafo adicional para a continuação dum parágrafo dividido. Uma convenção, utilizada pelo autor, é construir um novo parágrafo juntando «-CONT» (continuação) ao nome original do parágrafo. Assim, o desenho de programa para «sumar-encomenda» seria traduzido como:

```
PROCEDURE DIVISION.
PROGRAMA-PRINCIPAL.
        PERFORM ESTADO-INIC
        PERFORM SUMAR-ENCOMENDA THRU SUMAR-ENCOMENDA-CONT
        PERFORM FECHO.
.....
SUMAR-ENCOMENDA.
        PERFORM LER-PROXIMO-FICH-ENCOMENDA.
```

```
UNTIL-CICLO-1.
        IF  NOT  FIM-DE-FICH-ENCOMENDA
                PERFORM  PROCESSAR-REG-ENCOMENDA
                GO  TO  UNTIL-CICLO-1.
SUMAR-ENCOMENDA-CONT.
        PERFORM  PRINT-ENCOMENDA-SUMARIO.
PROCESSAR-REG-ENCOMENDA.
        PERFORM  ACTUALIZAR-CONTADORES
        PERFORM  LER-PROXIMO-FICH-ENCOMENDA.
. . . . .
```

Se não existirem operações a seguir ao enduntil, então o parágrafo de continuação está vazio e <nome-paragrafo>-CONT é ligado a um parágrafo nulo, isto é, o próximo nome parágrafo segue-se imediatamente. Um exemplo disto é fornecido no fim da secção A5.5.

A5.4 — CONDIÇÕES COMPOSTAS

Muitos compiladores ANS COBOL de nível mais baixo não implementam o AND e o OR para combinar condições simples em compostas. Assim, quaisquer condições compostas terão de ser substituídas por declarações IF encadeadas e refinamentos. AS declarações IF encadeadas são estudadas na secção 6.5. Deve ser tomado cuidado quando for implementado o else associado a uma condição composta.

Por exemplo:

```
if tipo-custa = «W» and custo-quantidade > 100 then
        rotina-desconto
else
        rotina-preço-total
endif
```

poderia ser implementado como:

```
IF TIPO-CUSTO = «W»
        IF CUSTO-QUANTIDADE > 100
                PERFORM  ROTINA-DESCONTO
        ELSE
                PERFORM  ROTINA-PRECO-TOTAL
        ELSE  PERFORM  ROTINA-PRECO-TOTAL.
```

Num sistema mínimo ANS COBOL (nível 1), as declarações IF encadeadas não estarão disponíveis e, assim, a condição composta, acima, teria de ser traduzida para:

```
IF TIPO-CUSTO = «W»
        PERFORM  VERIFI-CUSTO-QUANTIDADE
ELSE
        PERFORM  ROTINA-PRECO-TOTAL.
. . . . .
VERIFI-CUSTO-QUANTIDADE.
        IF CUSTO-QUANTIDADE > 100.
                PERFORM  ROTINA-DESCONTO
        ELSE
                PERFORM  ROTINA-PRECO-TOTAL.
```

Para implementar um ciclo until controlado por uma condição composta será necessário construir um campo extra para controlar o ciclo; isto é analisado, na secção 6.6, no contexto da estabilidade das condições.

A5.5 — «PERFORM-VARYING» (EXECUTE VARIANDO)

Esta variante da declaração PERFORM é pouco provável que esteja disponível em compiladores ANS COBOL de baixo nível. Um ciclo for do seguinte formato:

```
for <campo-controlo> = <valor-inicial> to <valor-final> do
        <procedimento>
endfor
```

pode ser implementado como:

```
        MOVE <valor-inicial> TO <campo-controle>.
FOR-CICLO-<id>.
        IF <campo-controlo> NOT > <valor-final>
                PERFORM <corpo-ciclo>
                ADD 1 TO <campo-controlo>
                GO TO FOR-CICLO-<id>.
.....
<corpo-ciclo>.
<procedimento>.
```

onde <id> deverá ser substituído por um identificador apropriado para distinguir este ciclo de todos os outros ciclos for do programa. Por exemplo, o primeiro ciclo for encontrado poderia ter <id> = 1, o segundo <id> = 2, etc.

Esta implementação utiliza apenas verbos que devem estar disponíveis numa implementação mínima de ANS COBOL.

Existe um problema com a estrutura de programa quando o novo nome de parágrafo FOR-CICLO-<id> é introduzido — isto é apreciado na secção A5.3 (ver «Perform-thru»).

Como exemplo desta tradução do ciclo for, considere o seguinte extracto simplificado do estudo de caso da secção 10.6:

```
imprimir-tabela
        imprimir-cabeçalhos
        for mc-indice = 1 to mc-limite do
                mover modelo-codigo (mc-indice) para sumario-codigo
                mover modelo-vendidos para sumario-vendidos
                escrever linha-sumario
        endfor
imprimir-cabeçalhos
        ....
```

Isto poderia ser traduzido para:

```
IMPRIMIR-TABELA.
        PERFORM IMPRIMIR-CABECALHOS
        MOVE 1 TO MC-INDICE.
FOR-CICLO-1.
        IF MC-INDICE NOT > MC-LIMITE
```

```
                PERFORM  IMPRIMIR-DETALHES
                ADD 1 TO MC-INDICE
                GO TO FOR-CICLO-1.
        IMPRIMIR-TABELA-CONT.
        IMPRIMIR-DETALHES.
                MOVE  MODELO-CODIGO  (MC-INDICE)  TO  SUMARIO-CODIGO
                MOVE  MODELO-VENDIDOS  (MC-INDICE)  TO  SUMARIO-VENDIDOS
                PERFORM  ESCREVER-LINHA-SUMARIO.
        IMPRIMIR-CABECALHOS.
                .....
```

Qualquer referência a IMPRIMIR-TABELA necessitará agora de ser alterada; por exemplo:

```
PERFORM  IMPRIMIR-TABELA
```

terá de ficar:

```
PERFORM  IMPRIMIR-TABELA  THRU  IMPRIMIR-TABELA-CONT
```

para que a estrutura correcta do programa seja mantida.

RESPOSTAS AOS EXERCÍCIOS SELECCIONADOS

Aqui são fornecidas soluções finais da maioria dos exercícios (mas não os de programação). Muitos dos exercícios têm soluções alternativas, mas igualmente satisfatórias.

Respostas 2

2.1 — Suponha que o ficheiro de *input* se chama «fich-empregado» e os nomes dos campos do registo de input são:

empregado-num	número do empregado
empregado-sal-mensal	valor ilíquido do salário
empregado-ded-ano	dedução anual contra a taxa

Um desenho de programa possível, similar ao utilizado no exemplo 2.4, é mostrado abaixo:

```
empregado-taxa
        ler-proximo-fich-empregado
        until fim-fich-empregado do
                taxa-emp-corrente
                imprimir-emp-detalhes
                ler-proximo-fich-empregado
        enduntil

taxa-emp-corrente
        calcular ded-pensão = 0.05 * empregado-sal-mensal
        calcular desc-mensal = empregado-ded-ano / 12
        calcular pag-taxa =
                empregado-sal-mensal — ded-pensão — desc-mensal
        encontrar-taxa
        calcular pag-liquido =
                empregado-sal-mensal — ded-pensão — tx-pag

imprimir-det-empregado
        mover-det-empregado-num para printer-empregado-num
        mover empregado-sal-mensal para printer-emp-sal-mensal
        mover ded-pensao para printer-ded-pensao
        mover tx-pag para printer-tx-paga
        mover pag-liquido para printer-pag-liquido
        escrever-linha-impressora
```

```
encontrar-taxa
        if pag-taxa > 0 then
                calcular tx-pag 0.2 * pag-taxa
        else
                mover 0 para tx-paga
        endif
```

2.2 — Convencione que o ficheiro de *input* se chama «fich-censo» e que o registo de *input* contém os seguintes campos:

censo-num-pessoas	número de habitantes por casa
censo-num-divisoes	número de divisões por casa

Um desenho de programa possível, similar ao utilizado no exemplo 2.5, é mostrado abaixo:

```
dados-censo
        inicializar-contadores
        ler-proximo-fich-censo
        until fim-propriedades do
                actualizar-contadores
                ler-proximo-fich-censo
        enduntil
        imprimir-contadores

inicializar-contadores
        mover 0 para num-hab-casa
        mover 0 para hab-superior-2
        mover 0 para hab-denso

actualizar-contadores
        calcular num-hab-casa = num-hab-casa + 1
        if censo-num-pessoas > 2 then
                calcular hab-superior-2 = hab-superior-2 + 1
        endif
        calcular razao-hab =
                censo-num-hab / censo-num-div
        if razao-hab > 1.5 then
                calcular hab-denso = hab-denso + 1
        endif

imprimir-contadores
        calcular sumar-perc-sup-2 =
                hab-superior-2 / num-hab-casa * 100
        calcular sumar-perc-denso =
                hab-denso / num-hab-casa * 100
        escrever-linha-sumario
```

2.3 — Admita que o ficheiro de *input* se chama «fich-elec» e que o registo de *input* contém os seguintes campos:

elec-num-cliente	número de conta do cliente
elec-tarifa	código da tarifa
elec-nome-mor	nome e morada do cliente
elec-unidades-gastas	unidades cunsumidas num quarto

Um desenho de programa posssível, baseado numa combinação dos dois desenhos anteriores e no estudo de caso da secção 2.5, é dado abaixo:

```
elec-conta
        inicializar-totais
        ler-proximo-fich-elec
        until fim-de-fich-elec do
                processar-cliente
                imprimir-det-cliente
                actualizar-contadores
                ler-proximo-fich-elec
        enduntil
        imprimir-totais

inicializar-totais
        mover 0 para tot-unidades
        mover 0 para tot-cobrar

processar-cliente
        if elec-tarifa = «D» then
                tarifa-domestica
        else
                tarifa-comercial
        endif

imprimir-det-cliente
        mover elec-num-cliente para det-num-cliente
        mover elec-tarifa para det-tarifa
        mover elec-nome-mor para det-nome-mor
        mover elec-unidades-gastas para det-unidades-gastas
        mover quantia-cobrada para det-qt-cobrada
        escrever-linha-detalhe

actualizar-contadores
        calcular tot-unidades = tot-unidades + elec-unidades-gastas
        calcular tot-cobrar = tot-cobrar + quantia-cobrada

imprimir-totais
        mover tot-unidades para sum-tot-unidades
        mover tot-cobrar para sum-tot-cobrar
        escrever-linha-sumario

tarifa-domestica
        if elec-unidades-gastas > 100 then
                calcular quantia-cobrada =
                        13 + (elec-unidades-gastas — 100) * 0.08
        else
                calcular quantia-cobrada =
                        3 + elec-unidades-gastas * 0.01
        endif

tarifa-comercial
        if elec-unidades-gastas > 200 then
                calcular quantia-cobrada =
                        30 + (elec-unidades-gastas — 200) * 0.05
```

```
else
        calcular quantia-cobrada =
                3 + elec-unidades-gastas * 0.15
endif
```

Respostas 3

3.1 — Os seguintes nomes são válidos: (*a*), (*b*), (*d*), (*e*), e (*i*). Os nomes (*d*) e (*e*) são só válidos como nomes-procedimento, enquanto os outros são válidos quer como nomes-dados quer como nomes-procedimento. Os nomes-inválidos são (*c*) e (*g*) porque contêm caracteres inválidos («.» e «%», respectivamente), (*f*) é uma palavra reservada, (*h*) começa por um hífen e (*j*) excede trinta caracteres de comprimento.

3.2 — Registos possíveis são:

```
8100123ASTEWART, ALAN BRECK          F03012      023
8016725%MCGREGOR, ROB ROY            BAA001      003
8100001*STEVENSON, ROBERT LOUIS      S—157       120
```

3.3 — A estrutura do registo seria do seguinte formato:

```
01      REG-STOCK.
        05      NUM-STOCK.
                10      NUM-STOCK-MAN.
                        15      NUM-STOCK-MAN-1     PIC AA.
                        15      NUM-STOCK-MAN-2     PIC 999.
                        15      NUM-STOCK-MAN-3     PIC X.
                10      STOCK-NUM-SERIE             PIC 9(8).
        05      STOCK-DESCRICAO                     PIC X(30).
        05      STOCK-ARMAZEM.
                10      STOCK-ARMAZEM-1             PIC AA.
                10      STOCK-ARMAZEM-2             PIC 9999.
        05      FILLER                              PIC XXXX.
        05      STOCK-NUM-REORD                     PIC 999.
```

Respostas 4

4.1 — O formato de *output*, para os dados de *input* fornecidos, irão agora consistir em duas linhas:

```
A00008 SELECCIONADO — BAIXO STOCK      0024    BAIXO STOCK
B00009 SELECCIONADO — BAIXO STOCK      0004    BAIXO STOCK
```

Isto acontece porque PERFORM ESCREVER-DETALHES faz agora parte da estrutura IF precedente, e, assim, escrever detalhes só será executado quando QUANTIDADE-STOCK > 25 for verdadeiro.

4.2 — A tradução do desenho poderia ser:

```
PERFORM  LER-PROXIMO-FICHEIRO-CONTA
PERFORM  CICLO-PROCESSAR-CONTA
        UNTIL  FIM-DE-FICHEIRO-CONTA.
```

```
CICLO-PROCESSAR-CONTA.
        PERFORM  PROCESSAR-CONTA
        PERFORM  LER-PROXIMO-FICHEIRO-CONTA.
```

Admitindo que o parágrafo PROCESSAR-CONTA não contém um PERFORM LER-PROXIMO-FICHEIRO-CONTA, a codificação incorrecta irá provocar um «ciclo infinito», que repetidamente irá processar o ficheiro FICH-CONTA.

Respostas 5

5.1 — Os valores do *input* deveriam ser escritos como se segue:

```
    23  002300
  27.5  002750
133.42  013342
2.125   212500
```

5.2 — Os valores calculados serão:

PREÇO-TOTAL	DESCONTO-DADO	PREÇO-FINAL
10.00	8.55	8.45
20.50	3.17	17.33
37.25	5.77	31.48
1500.00	32.50	1467.50

Note que no segundo e terceiro exemplo os dígitos menos significantes de DESCONTO-DADO são truncados e no último caso o dígito mais significante foi também truncado.

5.3 — Deveriam ser produzidos os seguintes resultados:

PIC	87654321	0.756	—253
£££££9.99	£54,321.00	bbbbb£0.75	£253.00
Z(5)9	654321	bbbbb0	bbb253
£**,***.99	£54.321.00	£******. 75	£***253.00
££££9.99DB	£4,321.00bb	bbbbb0.75bb	bb£253.00DB
ZZ.ZZ9+	54,321+	bbbbb0+	bbb553-

Respostas 6

6.1 — *a*) Descrições possíveis para «modelo-em-stock» e «id-part-modelo» poderiam ser:

```
05      ID-PART-MODELO                          PIC 99.
        88      MODELO-EM-STOCK     VALUE
                4 THRU 9, 12 THRU 16, 20, 24.
```

b) Um desenho equivalente possível seria:

```
if id-part-modelo > 3 and id-part-modelo < 10 then
        verificar-part-disponível
else
        if id-part-modelo >11 and id-part-modelo < 17 then
                verificar-part-disponivel
```

```
                else
                        if id-part-modelo = 20 then
                                verificar-part-disponivel
                        else
                                if id-part-modelo = 24 then
                                        verificar-part-disponivel
                                endif
                        endif
                endif
        endif
```

Existem muitas outras soluções possíveis utilizando condições simples com if's encadeados e refinamentos. A tradução acima para COBOL é perfeitamente linear.

6.2 — O pedaço de programa mostrado não inclui pontos finais, pelo que todos os IFs se encontram ligados. Quando executado, enquanto o TIPO-CONTA é igual a «W», o procedimento ROTINA–GROSSO será executado; em todos os outros casos o parágrado PROCESSAR-TIPO-CONTA não tem efeito.

Um desenho equivalente é:

```
processar-tipo-conta
        if tipo-conta =«W» then
                rotina-grosso
                if tipo-conta = «R» then
                        rotina-retalho
                        if <tipo-conta = «W»>
                                        or <tipo-conta not = «R»> then
                                tipo-conta-erro
                        endif
                endif
        endif
```

Note que a inserção de pontos finais depois de ROTINA-GROSSO e ROTINA-RETALHO, no pedaço de programa fornecido, continuará a não produzir a solução correcta. A terceira condição deve conter um AND em vez dum OR, porque doutra forma TIPO-CONT-ERRO é sempre executado.

A melhor solução é uma estrutura de if's encadeados do seguinte formato:

```
if tipo-conta = «W» then
        rotina-grosso
else
        if tipo-conta = «R» then
                rotina-retalho
        else
                tipo-conta-erro
        endif
endif
```

A tradução para COBOL é perfeitamente linear.

Respostas 7

7.1 — No exercício *a*) somente o parcelamento equivalente é relevante, mas em *b*) e *c*) ambos, parcelamento equivalente e análise do valor-fronteira, podem ser aplicados.

a) Irão ser necessários pelo menos cinco casos de teste.

média-desconto = 10	: válido
média-desconto = 15	: válido
média-desconto = 20	: válido
média-desconto numérico, mas não 10, 15 ou 20	: inválido
média-desconto não numérico	: inválido

b) Irão ser necessários pelo menos cinco casos de teste.

média-hora = 209	: inválido (fronteira)
média-hora = 210	: inválido (fronteira)
média-hora =725	: válido (fronteira)
média-hora = 726	: inválido (fronteira)
média-hora não numérico	: inválido

c) Este exercício é similar ao do exemplo 7.4. Poderia ser desenhada uma tabela da seguinte maneira.

Classe	Cod-tarifa	Unidades-consumidas	
1	B	não numérico	inválido
2	B	200	válido(front.)
3	B	201	válido(front.)
4	B	9999	válido(front.)
5	H	não numérico	inválido
6	H	100	válido (front.)
7	H	101	válido (front.)
8	H	9999	válido (máximo)
9	nem B ou H	— inválido	

7.2 — Dados de teste para o pedaço de programa poderiam ser:

TIPO-CLIENTE	CONDIC-ESPEC	QUANT-DESCONTO	resultado esperado
D	—	—	DESCONTO-DEALER
T	—	—	DESCONTO-TRANSAC
X	verdadeiro	—	DESCONTO-TRANSAC
X	falso	51	QUANT-DESCONTO
X	falso	50	nenhum

Um valor de TIPO-CLIENTE diferente de «D» ou «T» está referenciado por «X» e «—» e indica que quaisquer valores válidos podem ser utilizados para este campo. Os procedimentos que se espera sejam executados são mostrados à direita; todos os casos devem executar PRODUZIR-FACTURA.

7.3 — Os dados de teste poderiam ser do seguinte formato:

QUANTIDADE-VENDAS	VOLUME-NIVEL-DESCONTO	CLIENTE-TRANSACCAO	produto
25	50	falso	DESCONTO-10
25	50	verdadeiro	DESCONTO-5
75	50	—	nenhum

Quaisquer valores apropriados de QUANTIDADE-VENDAS e VOLUME-NIVEL-DESCONTO podem ser usados para tornar verdadeira ou falsa a condição relevante.

Utilizando os dados de teste acima referidos para o teste do desenho de programa no papel, ir-se-á verificar que estes três casos iriam produzir o seguinte resultado: nenhum, desconto-5 e desconto-10; assim, a tradução está incorrecta.

Respostas 8

8.1 — A adição de número de página no topo de cada página de *output* é perfeitamente linear. Pode ser adicionado um campo de dados extra a CABECALHOS-PAG, por exemplo:

```
05      NUM-PAG                 PICZZ9.
```

e pode ser acrescentado um contador a VECTOR-ESTADO por exemplo:

```
05      PAG-CORRENTE            PIC 999 VALUE 1.
```

De cada vez que CABECALHOS-PAG tem de ser escrito, o valor do PAG-CORRENTE é movido para NUM-PAG e PAG-CORRENTE é incrementado de uma unidade, ficando pronto para a página seguinte. Por exemplo:

```
ESCREVER-CABECALHOS-PAG
        MOVE PAG-CORRENTE PARA NUM-PAG
        WRITE LINHA-OUT FROM CABECALHOS-PAG
                AFTER ADVANCING PAGE
        COMPUTE PAG-CORRENTE = PAG-CORRENTE = 1.
```

Alternativamente, PAG-CORRENTE poderia ser inicializado com zeros em ESTADO-INIC e incrementado em uma unidade antes de se mover o seu valor para NUM-PAG.

A impressão de números de página no fundo da folha é mais difícil. Significa contar o número de linhas escritas até ao momento, avançando então o número de linhas apropriadas até ao fundo da página antes de escrever o número de página. (Infelizmente não existe nenhuma opção AFTER ADVANCING BOTTOM-OF-PAGE.) A impressão de número de página terá de ser adicionada ao fim de PROCESSAR-LOTE para evitar problemas com a numeração da última página.

8.2 — O que se requer aqui é um meio de distinguir entre uma finalização de um lote com os tipos de registo quatro e um terminados, sem se esperar quer pela leitura de um registo de tipo um para o próximo lote ou pela ocorrência de fim de ficheiro. Seria possível adicionar outro campo de dados booliano para distinguir um lote completo de um que tenha terminado sem um registo de tipo quatro.

Alternativamente, pode ser utilizada uma variável de estado com três estados. Suponha que «estado-lote» poderia ter três valores: lote em processamento representado por um espaço; registo de tipo quatro encontrado — «lote-completo», representado por «C», lote terminado sem registo de tipo quatro encontrado — «fim-lote», representado por «E».

O desenho pode então ser modificado como se mostra a seguir:

```
processar-lote
        imprimir-cabecalhos
        guardar-detalhes-conta
        mover espaço para estado-lote
        until lote-completo or fim-lote do
                ler-proximo fich-conta
                tratar-registo-corrente
        enduntil
        ordenar-lote

tratar-registo-corrente
        if tipo-reg-conta = 4 then
                verificar-fim-lote
        else
                if fim-de-fich-conta then
                        mover «E» para estado-lote
                else
                        if tipo-reg-conta = 1 then
                                mover «E» para estado-lote
                        else
                                processar-reg-det
                        endif
                endif
        endif

verificar-fim-lote
        if conta-numero = num-conta-corrente then
                mover «C» para estado-lote
        else
                erro-num-conta-inválido
        endif

ordenar-lote
        if fim-lote then
                erro-falta-tipo-4
        else
                comparar-totais
        endif
        imprimir-qt-cred

erro-falta-tipo-4
        display «não existe reg. tipo 4 para:», num-conta-corrente

comparar-totais
        if balanço-t-conta not = balanço-corrente then
                display «discrepancias de balanços para:»,
                        num-conta-corrente
                display «precedente:», balanço-t-conta.
                        «calculado:», balanço-corrente
        endif
```

onde «balanço-t-conta» é o balanço dado pelo registo precedente.

Em COBOL, este registo adicional poderia ser traduzido da seguinte maneira:

```
01      REG-PRECEDENTE.
        05      FILLER                  PIC 9.
        05      FILLER                  PIC X(6).
        05      BALANCO-T-CONTA         PIC S999V99.
```

A tradução para COBOL também envolve a descrição da variável de estado de três estados «estado-lote» da seguinte forma:

```
05      ESTADO-LOTE             PIC X.
        88      LOTE-COMPLETO                   VALUE «C».
        88      FIM-LOTE                VALUE «E».
```

A restante tradução para COBOL é bastante simples.

8.3 — O programa não irá terminar correctamente porque, quando o ficheiro estiver na condição de fim-de-ficheiro, a área de *input* para o ficheiro já não estará disponível para o programa. Assim, a declaração

```
MOVE 999999 TO ID-CLIENTE
```

irá causar a falha do programa por um erro de execução.

Respostas 9

9.1 — O primeiro passo é alterar a descrição de «num-conta-corrente» por forma que os seus primeiros dígitos sejam testados separadamente. Por exemplo:

```
num-conta-corrente
        num-conta-corrente-12     primeiros dois dígitos do número de conta
        num-conta-corrente-resto  restantes seis dígitos de número de conta
```

Em adição, pode ser descrito um campo de dados temporários para conter os primeiros dois dígitos do último número de conta.

```
campos-temporarios
        ult-num-conta-12 primeiros dois dígitos do último número de conta
```

Este campo pode ser inicializado após ser activado «num-conta-corrente» no início do programa:

```
. . . . .
encontrar-num-conta-corrente
mover num-conta-corrente-12 para ult-num conta 12
until num-conta-corrente = 999999999 do
. . . . .
```

Para imprimir os cabeçalhos de página necessários, o refinamento «iniciar-grupo» é modificado para:

```
iniciar-grupo
        if num-conta-corrente-12 not = ult-num-conta-12 then
                escrever-cabec-pag-sumar (após nova página)
                escrever-cabec-col-sumar (após saltar duas linhas)
                mover num-conta-corrente-12 para ult-num-conta-12
        endif
```

É possível alterar a primeira tentativa de algoritmo, mas o teste de alteração de dígitos deve ser feito um nível superior no desenho ou as alterações são desnecessariamente complexas.

9.3 — Requerem-se somente três alterações no algoritmo linha-balanço para que este lide com valores descendentes de chave. A leitura-proximo-<nome-ficheiro-input> deve tentar ler um registo do <nome-ficheiro-input>, e, se não houver registos disponíveis, coloque o valor da chave num valor mínimo impossível. O ciclo principal de controlo deve continuar até que este baixo valor seja encontrado, isto é:

```
linha-balanço
        leitura-proximo-movimentos
        leitura-proximo-mestre-velho
        encontrar-chave-corrente
        until chave-corrente = (low-value> do
                processar-grupo-chave-corrente
                encontrar-chave-corrente
        enduntil
```

O procedimento para encontrar a chave corrente deve ser modificado para encontrar o valor de chave mais alto; assim, a versão emendada é:

```
encontrar-chave-corrente
        if chave-movimento > chave-mestre-velho then
                mover chave-movimento para chave-corrente
        else
                mover chave-mestre-velho para chave-corrente
        endif
```

A implementação de leitura-proximo-<nome-ficheiro-input> necessita de ser modificada para :

```
LEITURA-PROXIMO-<nome-ficheiro-input>.
        READ <nome-ficheiro-input> INTO <area-nome-registo>
                AT END MOVE <low-value> TO <area-reg-chave>.
```

Se <area-reg-chave> for de tipo alfanumérico, então a constante figurativa LOW-VALUES pode ser utilizada para <low-value>.

A primeira tentativa do algoritmo pode ser alterada completando os comparadores dos primeiros dois testes no ciclo de controlo principal, como é mostrado abaixo.

```
until fim-mestre-velho ou fim-movimentos do
        if chave-movimentos <chave-mestre-velho then
                copiar-velho-novo
        else
                if chave-movimento > chave-mestre-velho then
                        possível-reg-novo
                        .....
```

Respostas 10

10.1 — O número máximo de um dado modelo que pode ser vendido é regulado pela *picture* de MODELO-VENDIDOS; o limite corrente é de 99 para qualquer tipo de modelo. O tamanho de MODELO–GROSSO e MODELO-LUCRO é também independente do número de qualquer modelo vendido porque estes são produto de MODELO-VENDIDOS com VEICULO-GROSSO e VEICULO-LUCRO, respectivamente.

Se o número de códigos de modelo em uso aumentar para 150, isto indica que «modelo-tabela» deve ser aumentado de forma a que «modelo-contador» seja repetido 150 vezes. As outras implicações desta alteração são no programa de COBOL, onde MC-INDICE e MC-LIMITE terão ambos de ser aumentados para PIC 999, para conter valores de índice até 150.

10.2 (*a*) — A solução mais simples, mas não necessariamente a mais eficiente, é:

```
for cr-indice = 1 to numero-indices do
        if cr-codigo (cr-indice) = «GDM» then
                mover 4.21 para cr-troca (cr-indice)
        endif
endfor
```

A fraqueza desta solução é não indicar se a actualizacão foi bem sucedida. Uma solução melhor poderia ser:

```
mover falso para actual-sucedida
for cr-indice = 1 to numero-indices do
        if cr-codigo (cr-indice) = «GDM» then
                mover 4.21 para cer-troca (cr-indice)
                mover verdadeiro para actual-sucedida
        endif
endfor
if não actual-sucedida then
        display «encontrar GDM na tabela falhado»
endif
```

Na prática, claro que o código a ser actualizado e a correspondente média de câmbio seriam fornecidas por campos e não por constantes.

É também possível formular uma solução alternativa que pára a pesquisa logo que o elemento requerido é encontrado.

10.2 (*b*) — A solução deste exercício é similar à solução (*a*), acima. Uma solução linear é:

```
mover falso para cr-encontrada
        for cr-indice = 1 to numero-indices do
                if cr-codigo (cr-indice) = «FFR» then
                        calcular qt-cambio =
                                quantia-requerida * cr-troca (cr-indice)
                        mover verdadeiro para cr-encontrada
                endif
        endfor
        if not cr-encontrada then
                display «encontrar FFR na tabela falhado»
        endif
```

10.2 (*c*) — Este não necessita de um ciclo for.

```
calcular numero-cr = numero-cr + 1
mover «NGU» para cr-codigo (numero-cr)
mover 3.88 para cr-troca (numero-cr)
```

Esta solução não prevê o *overflow* (transbordo) da tabela, sendo então uma solução segura a seguinte:

```
if numero-cr < 40 then
        calcular numero-cr = numero-cr + 1
        mover «NGU» para cr-codigo (numero-cr)
        mover 3.88 para cr-troca (numero-cr)
else
        display «transbordo da tabela»
        fecho
endif
```

São requeridos os seguintes campos:

```
05      CR-INDICE               PIC 99.
```

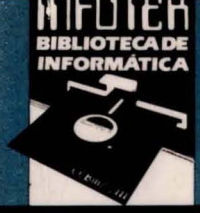
que pode ser adicionado a CAMBIO-TABELA, e

```
05      QT-CAMBIO               PIC 9(8)V99.
```

que poderia ser adicionado a CAMPOS-TEMPORARIOS ou, alternativamente, ser descrito como de edição fazendo parte de um registo de *output*. Os campos boolianos usados em (*a*) e (*b*) são:

```
05      IND-ACTUAL                      PIC X.
        88      ACTUAL-SUCEDIDA                 VALUE «V».
05      IND-ENCONTRADA                  PIC X.
        88      CR-ENCONTRADA                   VALUE «V».
```

que seriam adicionados a VECTOR-ESTADO.



Nesta obra combinam-se as ideias de programação estruturada e a linguagem COBOL na sua forma actual. A abordagem usada pelo autor foi a separação das tarefas de desenho e a codificação de um programa. A linguagem abstracta do desenho de programas foi usada para descrever algoritmos, os quais podem ser transportados para programas em COBOL através de um conjunto definido em regras de tradução e substituição de linguagem.

O livro apresenta seis estudos de casos ilustrando o desenvolvimento de programas completos em COBOL. Cada capítulo inclui problemas, que permitem ao leitor usar as técnicas descritas. No fim do livro inclui-se uma selecção de soluções resumidas.

ISBN 972-641-042-8

Biblioteca Municipal
Vila Franca de Xira
32600039901