PARTE A

PROGRAMAÇÃO DE COMPUTADORES

CAPÍTULO I

PROGRAMAÇÃO DE COMPUTADORES E INDUÇÃO MATEMÁTICA

1. Introdução

A programação de computadores é uma atividade que pode ser vista de várias maneiras. Por um lado, ela é praticada hoje em dia em escala industrial, e os problemas encontrados na confecção de grandes sistemas de programação – como sistemas operacionais ou de reservas de passagens aéreas – são comparáveis aos problemas encontrados em grandes projetos de engenharia. Por outro lado, a programação ainda é uma atividade quase que artesanal e a metodologia para desenvolvimento de programas de grande porte, e que sejam confiáveis e eficientes, é muito rudimentar. Este estado de coisas deve-se, em parte, ao fato da programação de computadores ser uma atividade muito recente, e que sofre transformações rápidas resultantes da expansão do seu campo de aplicação, e dos progressos tecnológicos na construção dos próprios computadores. Uma boa parte da pesquisa nesta área está dedicada à identificação dos princípios básicos que podem ser aplicados à programação, e do seu uso sistemático. Este tipo de pesquisa teve uma influência muito grande sobre projetos de novas linguagens de programação, e sobre técnicas de programação utilizadas.

O objetivo deste capítulo é apresentar alguns dos princípios em que se baseia a programação de computadores. Em primeiro lugar, é discutido na Seção 2 o conceito de procedimento efetivo que deve corresponder à noção intuitiva daquilo que pode ser calculado por meios mecânicos. Na Seção 3 apresentamos o conceito de linguagem de programação e o de programa, que é uma representação formal de procedimento. Na Seção 4, discutimos a relação entre os mecanismos de repetição existentes nos programas e o princípio de indução matemática. Finalmente, na Seção 5, apresentamos vários exemplos que ilustram o uso do princípio de indução para desenvolver, analisar e demonstrar propriedades de programas.

2. Procedimentos e algoritmos

Um *procedimento* é uma seqüência finita de instruções que podem ser executadas por um agente computacional, seja ele humano ou não. Este conceito corresponde, portanto, às noções intuitivas de "receita", "roteiro", "método", etc. Um exemplo clássico de procedimento é o chamado "Algoritmo de Euclides" que especifica como calcular o máximo divisor comum (*mdc*) de dois inteiros positivos m e n (o significado da palavra "algoritmo" será explicado mais adiante — por enquanto ele faz parte do nome do procedimento):

Passo 1: Adote como valores iniciais de x e y os valores m e n, respectivamente.

Passo 2: Adote como valor de r o resto da divisão do valor de x pelo valor de y.

Passo 3: Adote como o novo valor de x o valor de y, e como novo valor de y o valor de r.

Passo 4: Se o valor de r é nulo, então o valor de x é o *mdc* procurado, e o cálculo termina; caso contrário volte a executar as instruções do procedimento a partir do passo 2.

Este exemplo ilustra algumas propriedades que vamos exigir de um procedimento:

(i) A descrição do procedimento deve ser finita. No exemplo citado utilizamos uma seqüência finita de palavras e símbolos para descrever o procedimento.

(ii) Todo procedimento parte de um certo número de dados pertencentes a conjuntos especificados de objetos (como m e n que são inteiros positivos), e espera-se que produza um certo número de resultados (como o valor final de x) que mantêm uma relação específica com os dados.

(iii) Supõe-se que exista um agente computacional — humano, mecânico, eletrônico, etc. — que execute as instruções do procedimento. Este agente deverá ter uma maneira de guardar e recuperar informações durante a execução do procedimento. No exemplo anterior, estas informações seriam constituídas pelos valores de x, y e r, bem como a indicação do próximo passo a ser executado.

(iv) Cada instrução especificada pelo procedimento deve estar bem definida, não deixando dúvidas quanto ao seu resultado. Assim, no exemplo acima, supõe-se que dados os valores inteiros positivos de x e y, o agente computacional sabe calcular o resto da divisão de x

por y. A mesma instrução não estaria bem definida se x e y pudessem ter valores inteiros quaisquer: quais são os restos da divisão de 10 por 0 ou de -20 por -7? Evidentemente poderíamos estender de maneira conveniente a definição do resto da divisão para inteiros negativos, e neste caso o resultado para -20 e -7 estaria bem definido.

(v) As instruções do procedimento devem ser efetivas, isto é, devem ser tão simples que poderiam ser executadas, em princípio, por uma pessoa usando lápis e papel, num espaço de tempo finito. As instruções do Algoritmo de Euclides satisfazem este critério quando os valores envolvidos são inteiros positivos. Entretanto, elas deixariam de ser efetivas se os valores de x e de y pudessem ser números reais quaisquer em representação decimal, possivelmente de comprimento infinito. Um outro exemplo de instrução não efetiva seria: "adote para s o valor zero se existir um inteiro k maior do que 2, e inteiros positivos x, y e z tais que $x^k + y^k = z^k$". Para poder executar esta instrução, teríamos que saber se o chamado Último Teorema de Fermat, que afirma que tais inteiros não existem, é verdadeiro ou não. Este problema está em aberto desde que foi proposto no século XVII.

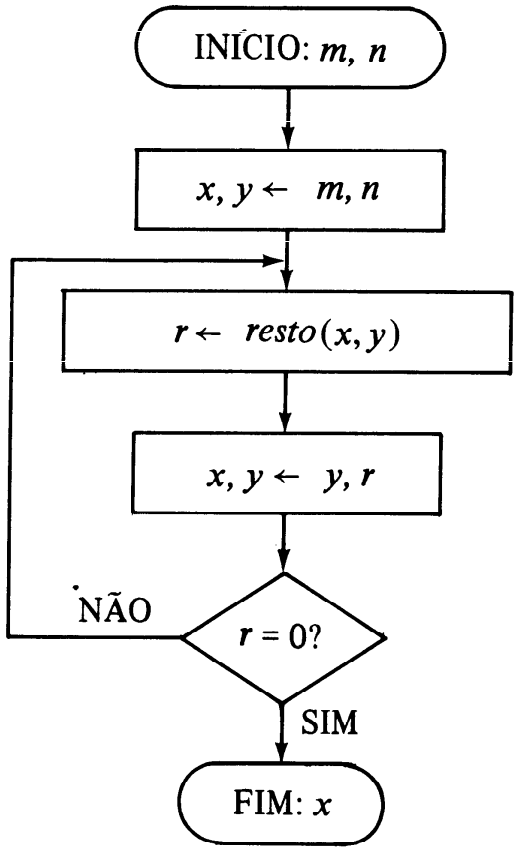

Figura 1

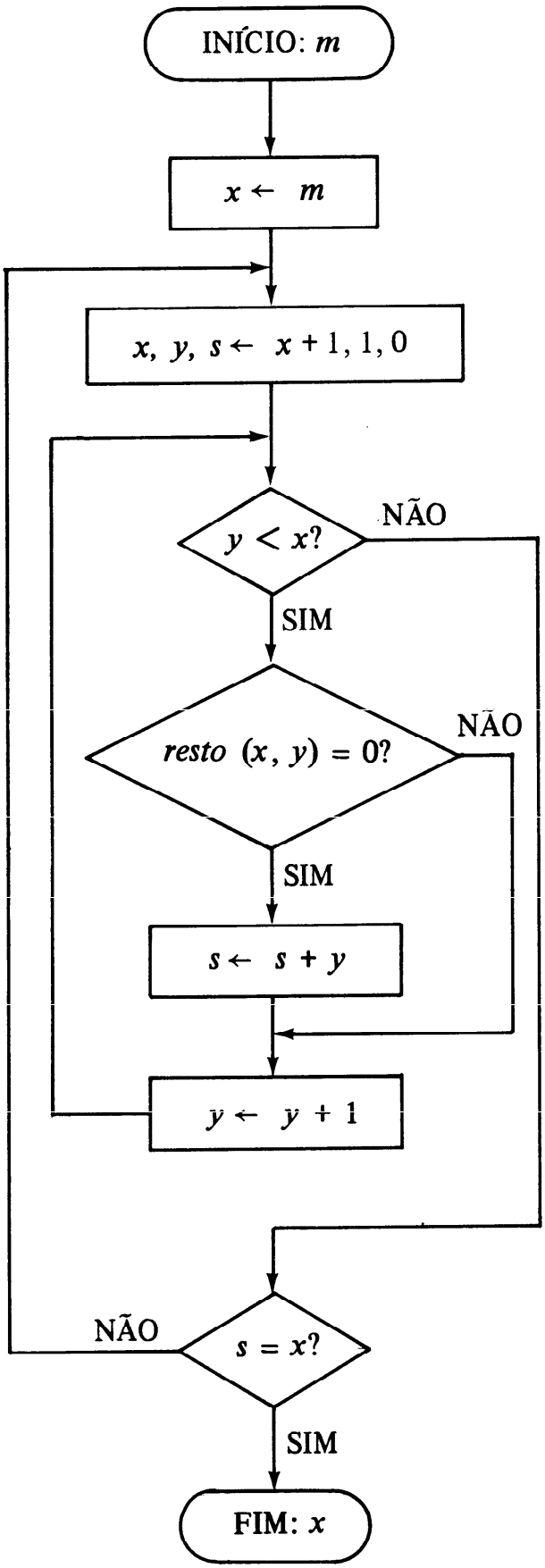
INÍCIO: m
x ← m
x, y, s ← x + 1, 1, 0
y < x?
NÃO
SIM
resto (x, y) = 0?
NÃO
SIM
s ← s + y
y ← y + 1
s = x?
NÃO
SIM
FIM: x

Figura 2

Note-se que não chegamos a definir precisamente o conceito de procedimento, e nem o faremos. Este é um conceito primitivo, correspondente à abstração das noções exemplificadas acima, independentemente da sua representação. Um procedimento que satisfaz as propriedades enumeradas acima é chamado também *procedimento efetivo*. Continuaremos, entretanto, a utilizar o termo procedimento para designar o mesmo conceito.

A descrição de um procedimento pode assumir várias formas distintas, e mais ou menos formalizadas. Uma maneira comum de se representar um procedimento é através do chamado *diagrama de blocos*. O Algoritmo de Euclides poderia ser representado pelo diagrama da Figura 1. O significado do diagrama deve ficar óbvio ao compará-lo com a descrição do procedimento dada anteriormente.

Um outro exemplo de procedimento está representado na Figura 2. A sua função é determinar o menor número perfeito maior do que um inteiro positivo m dado (um número k é perfeito se for igual à soma de todos os seus divisores exceto o próprio k). Mais um exemplo de procedimento está indicado na Figura 3.

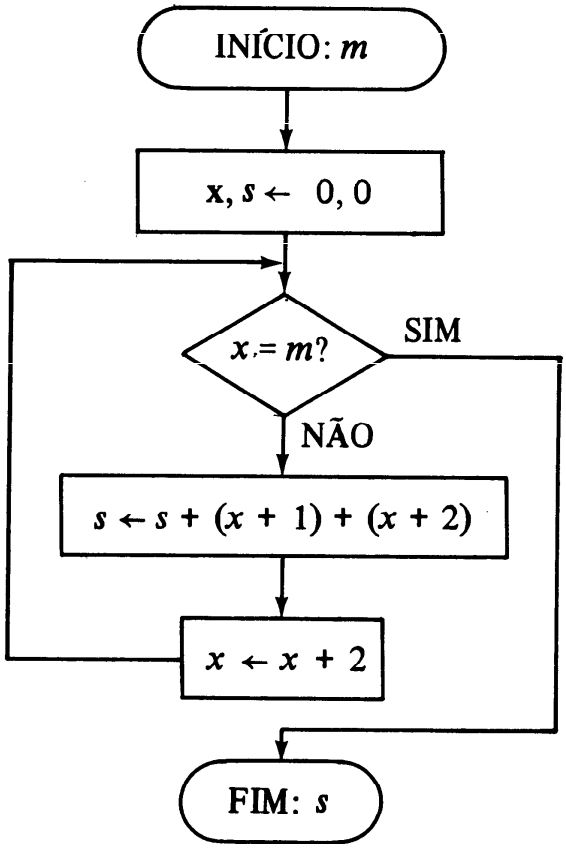

Figura 3

Uma pergunta que surge naturalmente é se um procedimento, partindo de certos dados iniciais, executa uma seqüência finita de cálculos, produzindo resultados finais, ou se então essa seqüência de cálculos nunca termina. No caso do Algoritmo de Euclides podemos mostrar que a seqüência de cálculos é finita provando a seguinte proposição: se no passo 2 do procedimento os valores de x e y são inteiros e positivos, então os passos 2, 3 e 4 serão executados apenas um número finito de vezes, com os cálculos terminando no passo 4. A demonstração é por indução sobre o valor de y. Se $y = 1$, então teremos, pela execução do passo 2, $r = 0$. Conseqüentemente, os passos 2, 3 e 4 são executados uma única vez, e os cálculos terminam no passo 4. Suponhamos agora que a proposição é verdadeira para qualquer $x > 0$ e qualquer y, com $1 \leq y < k$, e demonstraremos que ela é verdadeira para $y = k$. Por definição do resto da divisão de inteiros positivos, teremos, após a execução do passo 2, $0 \leq r < k$. Se $r = 0$, então a execução termina, como anteriormente, numa única vez. Se $r > 0$, então, com a execução dos passos 3 e 4, teremos $x = k > 0$ e $y = r$ com $0 < r < k$, e a execução volta ao passo 2. Por hipótese de indução, os passos 2, 3 e 4 serão executados um número finito p de vezes, com os cálculos terminando no passo 4. Ao todo teremos, então, $p + 1$ execuções para $y = k$. Notemos ainda que os valores iniciais $x=m$ e $y=n$ resultantes da execução do passo 1 satisfazem as condições da proposição acima. Podemos concluir, portanto, que a execução do Algoritmo de Euclides termina para quaisquer inteiros positivos m e n.

No caso do segundo procedimento, sabemos que o cálculo termina para certos valores de m. Por exemplo, se $m = 4$ então obteremos o resultado $x = 6$ pois $5 \neq 1$ e $6 = 1 + 2 + 3$. Entretanto, no caso geral a resposta não é conhecida, pois a existência ou não de um número infinito de números perfeitos é um problema em aberto. Se existirem infinitos números perfeitos, então a execução do procedimento termina para qualquer m; caso contrário, se K é o maior número perfeito, então o procedimento executa uma seqüência infinita de cálculos para todo $m \geq K$. Finalmente, no caso do terceiro exemplo podemos ver que a execução do procedimento termina com $s = \sum_{0}^{m} i$ para valores pares de m, pois os valores consecutivos de x são 0, 2, 4, 6, ... Para valores ímpares de m, a igualdade $x = m$ nunca será satisfeita, e a execução do procedimento não termina.

Entre todos os procedimentos, teremos um interesse especial naqueles cuja execução termina para quaisquer valores dos dados. Estes procedimentos serão chamados *algoritmos*(1).

Em conseqüência da discussão anterior, conclui-se que o procedimento que chamamos Algoritmo de Euclides é realmente um algoritmo. Quanto ao segundo procedimento, a resposta não é conhecida, e o terceiro exemplo representa um procedimento que não é um algoritmo.

Neste ponto fica claro que o problema de decidir se um dado procedimento é ou não um algoritmo deve ser difícil. Caso contrário, já saberíamos as respostas sobre várias conjeturas, tais como a existência de um número infinito de números perfeitos, a veracidade do Último Teorema de Fermat, e outros. Este problema será discutido de maneira mais completa no Capítulo B.II.

Note-se que todo procedimento é um método para o cálculo de alguma função, eventualmente não definida para certos argumentos. Assim, o terceiro diagrama de blocos corresponde à função h que pode ser descrita por:

$$h(m) = \begin{cases} \sum_{0}^{m} i & \text{se } m \text{ é par} \\ \text{não definida} & \text{se } m \text{ é ímpar.} \end{cases}$$

Por outro lado, uma mesma função pode ser calculada por vários procedimentos distintos. O procedimento da Figura 4 calcula a mesma função h definida acima.

3. Programas e linguagens de programação

Vimos na seção anterior que um procedimento pode ser especificado por uma mistura de palavras e símbolos, como foi feito com a primeira versão do Algoritmo de Euclides. Esta maneira é, em geral, bastante adequada quando se trata de descrever procedimentos que serão interpretados por pessoas. Entretanto, para que um procedimento possa ser executado por uma máquina, é necessário que a sua descrição seja feita numa linguagem que não tenha as imprecisões nem a varia-

(1) O termo "algoritmo" vem do nome Muhammad ibn-Musa al-Khwarizmi, autor do famoso texto matemático *Kitab al-jabr wa al-muqabalah* escrito em árabe no Século IX, e que deu origem ao termo "álgebra". Khwarezm, a cidade de origem do autor, é hoje a pequena cidade de Khiva na república soviética de Uzbequistão.

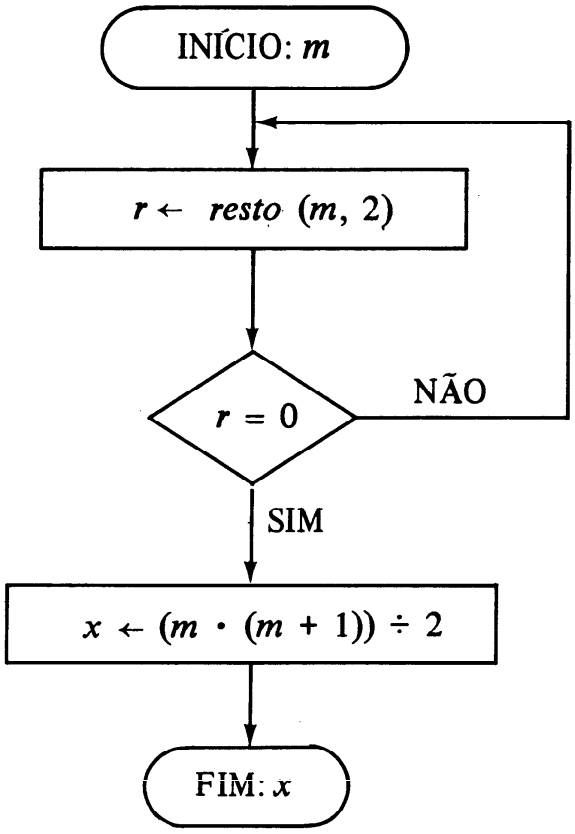

Figura 4

bilidade de uma língua natural. Uma outra vantagem da formalização é que ela permite maior rigor nas definições e demonstrações sobre procedimentos.

O fato de que a maneira mais conveniente de transmitir informação aos computadores modernos é através de seqüências de caracteres exclui, em geral, o uso de diagramas de blocos, se bem que estes podem ser definidos de maneira rigorosa. O método mais comum, portanto, para se especificar um procedimento de maneira formal é através de uma *linguagem de programação*.

Uma linguagem de programação é definida por um conjunto de símbolos, chamado *alfabeto*, que podem ser usados na representação de procedimentos, e por um conjunto de regras que especificam como compor estas representações e quais são as ações associadas a estas representações. Uma seqüência de símbolos de uma linguagem de programação que representa um ou mais procedimentos será chamada *programa*.

As várias linguagens de programação têm características bastante diferentes, conforme a sua finalidade. Existem linguagens muito simples, que incluem um número muito pequeno de operações primitivas, mas

que têm um grande interesse para a teoria da computação, como por exemplo a linguagem de Turing, a ser vista no Capítulo B.I. Tais linguagens dificilmente são usadas para programar procedimentos utilizados na prática. As chamadas *linguagens de máquina*, que são "compreendidas" diretamente pelos computadores, variam bastante de um computador ao outro, e a programação nestas linguagens pode ser muito trabalhosa. A tendência moderna é a utilização das chamadas *linguagens de alto nível*, mais adequadas para a representação de procedimentos. Um programa escrito em linguagem de alto nível é traduzido para a linguagem de máquina de um dado computador, para que possa ser executado. Essa tradução, por sua vez, é feita pelo próprio computador, executando um programa especial chamado *compilador*. Conseqüentemente, tudo se passa como se o computador "compreendesse" a linguagem de alto nível.

É importante saber, em geral, se numa dada linguagem de programação podem-se representar todos os procedimentos efetivos, isto é, se a linguagem é universal. Uma discussão pormenorizada deste problema está fora do alcance deste texto, mas devemos considerar os seguintes fatos. Em primeiro lugar, é aceita universalmente a chamada *Tese de Church*, segundo a qual qualquer procedimento pode ser representado em linguagem de Turing. Evidentemente, esta tese não pode ser demonstrada, pois não temos uma definição de procedimento. Entretanto, o volume de evidência empírica, que abrange todos os procedimentos já formulados, faz com que se aceite esta tese. Em segundo lugar, pode-se provar que os programas escritos em linguagem de Turing podem ser "traduzidos" em programas equivalentes em outras linguagens de programação, contanto que estas contenham certos conjuntos mínimos de operações primitivas, como por exemplo soma, subtração, teste de zero, e um mecanismo para indicar cálculos repetitivos. Em geral, as linguagens de programação incluem um número de operações maior do que o citado acima, tornando o trabalho de programação mais conveniente, e resultando numa execução mais rápida de certas operações que poderiam ser programadas através das operações básicas. A Parte B do texto inclui uma discussão deste aspecto de linguagens.

Neste texto utilizaremos uma linguagem de programação bastante simples, mas que se assemelha às linguagens de alto nível usadas na prática, e em particular à linguagem chamada Algol 60. Antes de descrever esta linguagem, que denominaremos *LP*, mostramos nas Figuras

5 e 6 programas que representam os procedimentos das Figuras 1 e 2, respectivamente.

Um programa em *LP* é constituído por uma seqüência de símbolos, sendo que o espacejamento entre os mesmos, as mudanças de linha e, em geral, a disposição física são irrelevantes, e têm por finalidade

```
procedimento Euclides(m, n):
    início
        x, y ← m, n;
        repita
            r ← resto(x, y);
            x, y ← y, r
        até que r = 0;
        devolva x
    fim
```

Figura 5

```
procedimento número perfeito(m):
    início
        x ← m;
        repita
            x, y, s ← x + 1, 1, 0;
            enquanto y < x faça
                início
                    se resto(x, y) = 0 então s ← s + y
                                       senão nada;
                    y ← y + 1
                fim;
        até que s = x;
        devolva x
    fim
```

Figura 6

aumentar a legibilidade e compreensão pelo leitor humano. A estrutura de um programa é indicada pelos símbolos de pontuação tais como vírgula, dois-pontos, ponto-e-vírgula, seta e outros, bem como por símbolos compostos de letras impressas em negrito. Neste texto, estes símbolos são palavras como **início, fim, se, então, senão** e outras, e o seu significado ajuda na compreensão dos programas. Os nomes dos programas (como *Euclides* e *número perfeito*), dos argumentos (como

m e n) e das variáveis (como x, y e s) podem ser escolhidos de maneira arbitrária.

Passaremos a descrever agora de maneira mais precisa, se bem que informal, a linguagem *LP*. Uma indicação de como tal definição poderia ser formalizada será dada no final desta seção.

Em primeiro lugar, deveríamos especificar os objetos que podem ser manipulados nesta linguagem e quais as operações primitivas disponíveis para manipulá-los. Como já foi mencionado, bastaria adotar os números naturais e um mínimo de operações primitivas para conseguir representar, segundo a Tese de Church, qualquer procedimento. Entretanto, esta decisão não permitiria a manipulação direta de outros objetos tais como caracteres e símbolos, seqüências de números, vetores, matrizes, etc. Todos estes objetos teriam que ser representados por meio de uma codificação conveniente através dos números naturais. Assim, os dados para os programas teriam que ser codificados, os resultados teriam que ser decodificados, e ao programar teríamos que estar sempre atentos ao fato de utilizarmos números para representar objetos de outra natureza. Por conveniência, incluímos então em *LP* a manipulação de vários tipos de objetos, além dos números naturais. A especificação precisa destes tipos de objetos e das operações primitivas correspondentes será feita ao apresentarmos os exemplos particulares de programas que os utilizam. Suporemos apenas que existe uma maneira de denotar as constantes como 0 (inteiro zero), $\langle 2, 7, 10 \rangle$ (seqüência dos três inteiros 2, 7 e 10), "*aspectos teóricos*" (cadeia de caracteres), etc. As *expressões* da linguagem serão formadas utilizando-se estas constantes, nomes de argumentos e variáveis, operações primitivas, ou então nomes de procedimentos representados no programa. Neste último caso, devem ser calculados os valores dos argumentos, e executado o procedimento correspondente, que deverá devolver os valores calculados.

Um programa em *LP* será constituído da representação de um procedimento, seguido eventualmente da representação dos procedimentos auxiliares.

A representação de um procedimento tem a forma geral:

procedimento $f(m_1, m_2, \ldots, m_n)$: S

com $n \geq 0$, onde f será o nome do procedimento, os m_i são os nomes dos argumentos (dados), e S é um *comando* que indica como devem ser calculados os resultados da execução de f, para os valores $m_1, \ldots, m_n$ dos dados.

Os comandos em *LP* podem assumir várias formas. O *comando de atribuição de valor* tem a forma

$$x_1, \ldots, x_n \leftarrow E_1, \ldots, E_n$$

com $n \geq 1$, onde os x_i são nomes de variáveis, e os E_i são expressões que envolvem variáveis, constantes, argumentos, as operações primitivas e, eventualmente, nomes de procedimentos. A ação que corresponde à execução deste comando consiste em se calcular, em primeiro lugar, os valores e_i das expressões E_i, e em seguida adotá-los como os novos valores das variáveis x_i ($i = 1, \ldots, n$), respectivamente. Uma outra forma possível para o comando de atribuição é

$$x_1, \ldots, x_n \leftarrow f(E_1, \ldots, E_p)$$

com $n \geq 2$ e $p \geq 0$, onde f é o nome de um procedimento definido no programa, cuja execução deve devolver n resultados. (O caso $n = 1$ está incluído na primeira forma do comando.)

O *comando condicional* tem a forma

se E **então** S_1 **senão** S_2

onde E é uma expressão, e S_1 e S_2 são comandos quaisquer. Durante a execução deste comando, é calculado em primeiro lugar o valor e da expressão E, que deve ser verdadeiro ou falso. Se e for verdadeiro então será executado em seguida o comando S_1; caso contrário será executado o comando S_2. Quando se utilizam diagramas de blocos, o comando condicional corresponde à construção indicada na Figura 7.

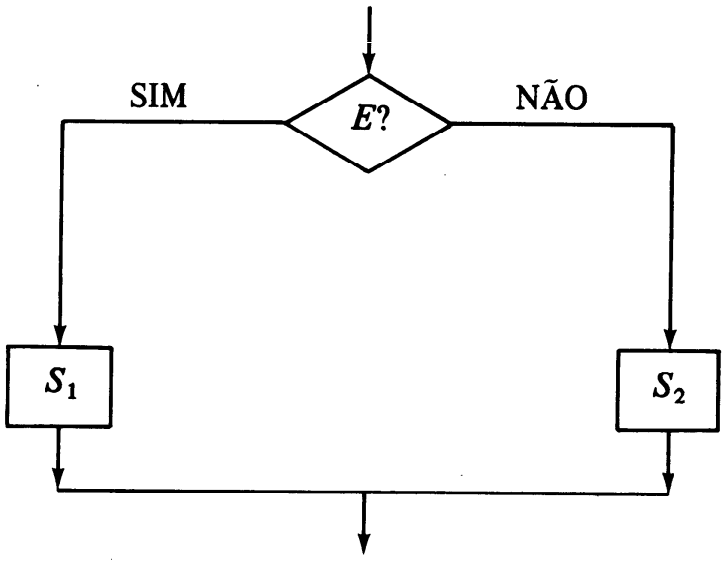

Figura 7

O comando **nada**, que não gera ação alguma, será usado, às vezes, como uma das alternativas S_1 ou S_2.

O *comando repetitivo* tem a forma

enquanto E **faça** S

onde E é uma expressão cujo valor deve ser verdadeiro ou falso, e S é um comando qualquer. Durante a execução, se o valor de E é falso, então a execução do comando repetitivo termina; caso contrário é executado o comando S, e o comando inteiro "**enquanto** E **faça** S" volta a ser executado. A Figura 8 indica o diagrama de blocos correspondente.

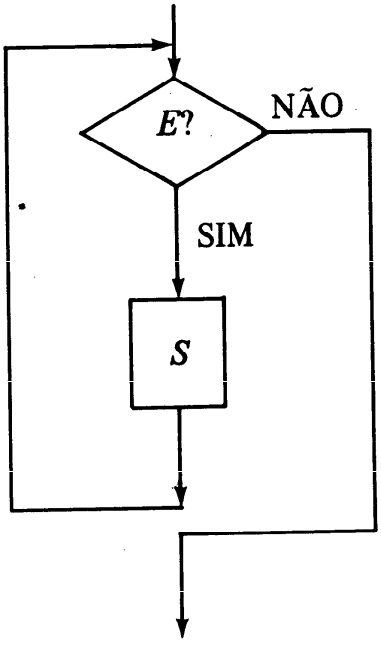

Figura 8

Um outro comando repetitivo tem a forma:

repita $S_1; S_2; \ldots; S_n$ **até que** E

com $n \geq 1$, onde os S_i são comandos quaisquer, e E é uma expressão cujo valor deve ser verdadeiro ou falso. O significado deste comando está indicado através do diagrama de blocos da Figura 9.

Em certos casos aparece a necessidade de se transformar uma seqüência de comandos num único comando como, por exemplo, quando esta seqüência deve constituir uma das alternativas de um comando condicional. Neste caso será usado o *comando composto* da forma:

início S_1 ; S_2 ; ... ; S_n **fim**

com $n \geq 1$. Note-se que os símbolos **início** e **fim** funcionam apenas como parênteses, indicando um agrupamento de comandos quando necessário. Os comandos S_1, S_2, ..., S_n serão executados um após o outro, nesta ordem.

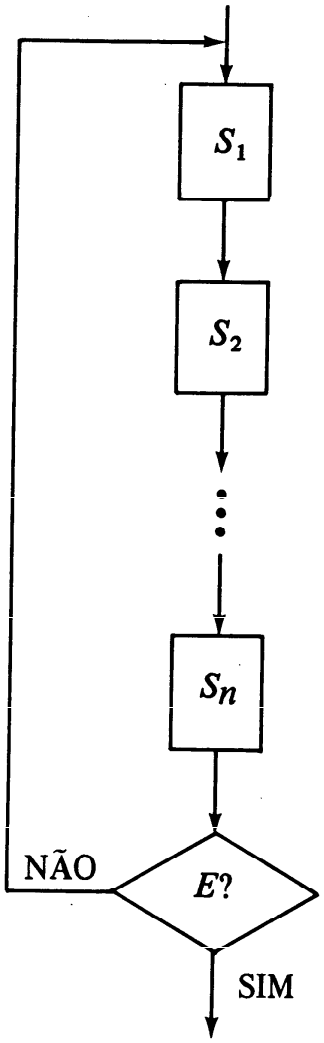

Figura 9

Finalmente, os resultados a serem devolvidos por um procedimento são indicados pelo *comando de retorno* da forma:

devolva $E_1, E_2, \ldots, E_n$

com $n \geq 1$. Os valores e_i das expressões E_i são devolvidos como resultados do procedimento dentro do qual o comando de retorno ocorre, encerrando a sua execução.

Uma convenção importante que adotaremos para a linguagem *LP* é a de que o *escopo* dos nomes dos argumentos e das variáveis é constituído pelo próprio texto da definição do procedimento no qual eles ocorrem. Em outras palavras, se num programa temos duas definições de procedimentos, ambas fazendo referência à variável x, na realidade trata-se de duas variáveis distintas.

Ocasionalmente, haverá observações no meio do programa visando facilitar a compreensão do mesmo. Tais observações, contidas sempre entre as chaves { e }, serão ignoradas durante a execução do programa.

O leitor atento deve ter notado que a definição da forma de um comando em *LP* é uma definição indutiva, e poderia ser formulada de maneira mais concisa. Se V é o conjunto de nomes de argumentos e de variáveis, F é o conjunto de nomes de procedimentos, e X é o conjunto de seqüências de símbolos que representam expressões, então o conjunto C de seqüências de símbolos que representam comandos de *LP* pode ser definido por

C é o menor conjunto tal que:

(i) a seqüência "**nada**" está em C;

(ii) se $x_1, \ldots, x_n$ estão em V, e $E_1, \ldots, E_n$ em X $(n \geq 1)$, então a seqüência "$x_1, \ldots, x_n \leftarrow E_1, \ldots, E_n$" está em C;

(iii) se $x_1, \ldots, x_n$ estão em V, f em F, e $E_1, \ldots, E_p$ em X $(n \geq 2, p \geq 0)$, então a seqüência "$x_1, \ldots, x_n \leftarrow f(E_1, \ldots, E_p)$" está em C;

(iv) se $E_1, \ldots, E_n$ estão em X $(n \geq 1)$, então a seqüência "**devolva** $E_1, E_2, \ldots, E_n$" está em C;

(v) se E está em X, e S_1 e S_2 em C, então a seqüência "**se** E **então** S_1 **senão** S_2" está em C;

(vi) se E está em X e S em C, então a seqüência "**enquanto** E **faça** S" está em C;

(vii) se E está em X, e $S_1, \ldots, S_n$ em C $(n \geq 1)$, então a seqüência "**repita** S_1 ; ...; S_n **até que** E" está em C;

(viii) se $S_1, \ldots, S_n$ estão em C $(n \geq 1)$, então a seqüência "**início** S_1 ; ...; S_n **fim**" está em C.

Na prática, estas definições são abreviadas usando-se a notação chamada FNB[2]. Nesta notação, os conjuntos de seqüências de sím-

(2) A abreviação vem de "Forma Normal de Backus". Essa notação foi usada pela primeira vez por J. Backus para descrever a linguagem Algol 60 num relatório editado por P. Naur [87]. Em inglês utiliza-se a abreviação BNF que indica tanto "Backus Normal Form" como "Backus Naur Form".

bolos recebem nomes mnemônicos colocados entre ⟨ e ⟩. Assim, a definição acima ficaria:

⟨comando⟩ :: =

nada
| ⟨variável⟩ [, ⟨variável⟩] ← ⟨expressão⟩ [, ⟨expressão⟩]
| **devolva** ⟨expressão⟩ [, ⟨expressão⟩]
| **se** ⟨expressão⟩ **então** ⟨comando⟩ **senão** ⟨comando⟩
| **enquanto** ⟨expressão⟩ **faça** ⟨comando⟩
| **repita** ⟨comando⟩ [; ⟨comando⟩] **até que** ⟨expressão⟩
| **início** ⟨comando⟩ [; ⟨comando⟩] **fim**

A notação [α] indica a repetição da seqüência α zero ou mais vezes. Note-se que os casos (ii) e (iii) da definição foram reunidos num único caso, por motivo de simplicidade. A rigor, isto não seria correto, pois permitiria comandos como "$x, y, z \leftarrow 0, 1$". Utilizando-se a mesma notação, a definição de *LP* pode ser completada com:

⟨programa⟩ :: =
⟨procedimento⟩ [⟨procedimento⟩]
⟨procedimento⟩ :: =
procedimento ⟨nome⟩ ( ): ⟨comando⟩
| **procedimento** ⟨nome⟩ (⟨nome⟩ [, ⟨nome⟩]): ⟨comando⟩

Note-se que estas definições determinam apenas a forma de um programa em *LP*, isto é, a sua *sintaxe*. Para que a definição fosse completa, deveríamos especificar também o significado, ou seja, a *semântica* de cada construção. Os métodos para a especificação da semântica de linguagens de programação são bastante complicados, e não serão discutidos neste texto.

Nas Partes *B* e *C*, a linguagem *LP* será utilizada para representar vários algoritmos. A fim de aumentar a clareza, serão introduzidas certas construções não incluídas na nossa definição, mas cujo significado deve ser óbvio dada a notação usual que será empregada. Uma convenção comum será a eliminação da seqüência "**senão nada**", sempre que possível. Esta eliminação não poderá ser feita nas construções da forma "**se** E_1 **então se** E_2 **então** S_1 **senão nada senão** S_2" e "**se** E_1 **então se** E_2 **então** S_1 **senão** S_2 **senão nada**", pois obteríamos a mesma forma "**se** E_1 **então se** E_2 **então** S_1 **senão** S_2" que seria, portanto, ambígua.

4. Iteração e recursão

Um conceito fundamental em programação é o de execução repetitiva de um comando, ou de uma série de comandos. Como vimos, a descrição de um procedimento é sempre finita. Na ausência de mecanismos repetitivos, o número de cálculos executados por um programa será limitado, então, por uma constante cujo valor depende do próprio programa. Assim, por exemplo, o Algoritmo de Euclides não poderia ser descrito por um programa desta espécie, uma vez que dado um inteiro K, é sempre possível escolher valores m e n tais que o número de divisões necessárias será maior do que K.

Todas as linguagens de programação de aplicação geral incluem mecanismos que causam execução repetitiva de comandos. Um mecanismo explícito existente em *LP* são os comandos repetitivos, chamados também de *comandos iterativos*. Por exemplo, no programa que representa o Algoritmo de Euclides, a seqüência de comandos "$r \leftarrow resto\ (x, y)$; $x, y \leftarrow y, r$" será executada repetitivamente até que o valor de r seja zero. Assim, se os valores dados são $m = 119$ e $n = 544$, os valores consecutivos das variáveis x, y e r durante o cálculo repetitivo serão:

x	y	r
119	544	119
544	119	68
119	68	51
68	51	17
51	17	0
17	0	

Analisemos agora o significado dos comandos iterativos, escolhendo para isto a forma "**enquanto** E **faça** S". Suponhamos que $m_1, \ldots, m_n$ são os argumentos e $x_1, \ldots, x_p$ são as variáveis de um programa no qual ocorre este comando iterativo. O comando S, ao ser executado, calcula, a partir dos valores $m_1, \ldots, m_n$ e $x_1, \ldots, x_p$, os novos valores atribuídos às variáveis $x_1, \ldots, x_p$ (alguns podem permanecer inalterados). Podemos supor, portanto, a existência das funções $g_i(u_1, \ldots, u_n, v_1, \ldots, v_p)$ $(i = 1, \ldots, p)$ que representam este cálculo, e reescrever o comando iterativo sob a forma:

enquanto E **faça**

$$x_1, x_2, \ldots, x_p \leftarrow g_1(m_1, \ldots, m_n, x_1, \ldots, x_p), g_2(m_1, \ldots, m_n, x_1, \ldots, x_p), \ldots, g_p(m_1, \ldots, m_n, x_1, \ldots, x_p).$$

Abreviando a notação para as seqüências de variáveis e funções, e lembrando que o valor de E também é função de $m_1, \ldots, m_n$ e $x_1, \ldots, x_p$, podemos escrever:

enquanto $E(m, x)$ **faça** $x \leftarrow g(m, x)$.

Sejam $a^0 = (a_1^0, \ldots, a_p^0)$ os valores iniciais das variáveis $x_1, \ldots, x_p$, antes de começar a execução do comando iterativo. Enquanto prosseguir a repetição de S, teremos uma seqüência de valores de x: a^0, a^1, a^2, ..., onde

$$a^i = g(m, a^{i-1}) \qquad (i \geq 1).$$

Esta seqüência será finita ou não conforme exista ou não um $k \geq 0$ tal que $E(m, a^k)$ seja falso. Caso a seqüência seja finita, o seu último valor a^k fornecerá os valores finais das variáveis x.

Estas considerações sugerem a seguinte regra para a demonstração de proposições que relacionam os valores de m e de x, baseada no princípio de indução matemática:

Se (i) a proposição $P(m, x)$ é verdadeira antes da execução de "**enquanto** E **faça** S" .

e (ii) a veracidade de $P(m, x)$ e de $E(m, x)$ implica a veracidade de $P(m, x)$ após a execução de S,

então

ou (iii) a execução do comando "**enquanto** E **faça** S" não termina,

ou (iv) após o seu término a proposição $P(m, x)$ é verdadeira e $E(m, x)$ é falsa.

Esquematicamente podemos representar esta regra num diagrama de blocos como indicado na Figura 10.

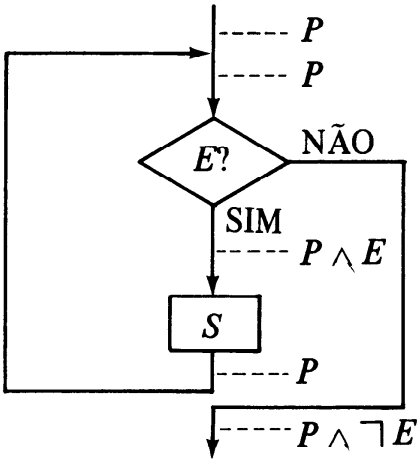

Figura 10

No texto de um programa apresentaremos o uso desta regra colocando as proposições como comentários:

$\{P\}$ **enquanto** $\{P\}\ E$
faça $\{P \wedge E\}\ S\ \{P\}$;
$\{P \wedge \hat{\neg}\ E\}$

Note-se que a indução está sendo feita sobre o número de vezes que o comando S é executado. A premissa (i) é a base da indução, e a premissa (ii) é o passo indutivo.

Considerações análogas levam à formulação de uma regra para o comando "**repita** S_1; ...; S_r **até que** E" que pode ser enunciada por:

$\{P\}$ **repita**
$\{P\}\ S_1; \ldots; S_r\ \{Q\}$
até que E;
$\{Q \wedge E\}$

onde $Q \wedge \neg E$ implica P. A Figura 11 indica esta regra num diagrama de blocos.

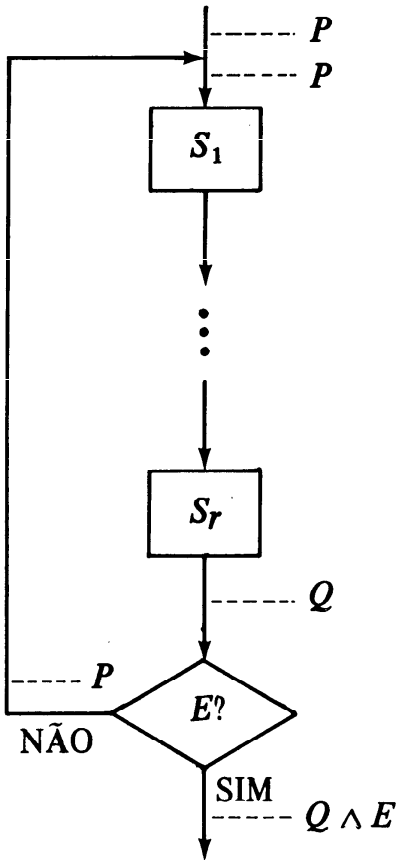

Figura 11

Um outro mecanismo existente em *LP*, e em várias linguagens de programação usadas na prática, é a *recursão*. Esta consiste em utilizar, direta ou indiretamente, um procedimento dentro do mesmo procedimento que o define. Consideremos a seguinte definição indutiva da função fatorial:

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n \cdot (n-1)! & \text{se } n > 0. \end{cases}$$

Esta definição pode ser transformada no seguinte programa em *LP*:

```
procedimento fat(n):
    se n = 0 então devolva 1
             senão devolva n · fat(n − 1)
```

Para compreender melhor o mecanismo de recursão, simulemos a execução deste programa para $n = 4$. Inicialmente, obtém-se:

$fat(4) = 4 \cdot fat(3)$

ou seja neste ponto deve-se suspender temporariamente o cálculo de $fat(4)$, "lembrando" para o uso posterior o valor $n = 4$ e passar a usar o mesmo programa para $n = 3$, e assim por diante:

$fat(3) = 3 \cdot fat(2)$
$fat(2) = 2 \cdot fat(1)$
$fat(1) = 1 \cdot fat(0)$
$fat(0) = 1$

Neste ponto devemos ir "relembrando" os valores anteriores de n:

$fat(1) = 1 \cdot 1 = 1$
$fat(2) = 2 \cdot 1 = 2$
$fat(3) = 3 \cdot 2 = 6$
$fat(4) = 4 \cdot 6 = 24$

Note que o programa será usado $n + 1$ vezes para calcular $n!$. Não é difícil, por outro lado, escrever uma versão iterativa do programa que calcula $n!$:

```
procedimento fat(n):
    início
        s, x ← 1, 1;
        enquanto x ≤ n faça s, x ← s · x, x + 1;
        devolva s
    fim
```

Simulando a execução deste programa para $n = 4$, obtêm-se os seguintes valores consecutivos de x e s:

x	s
1	1
2	1
3	2
4	6
5	24

Um outro exemplo de programa recursivo é o que calcula os números de Fibonacci. A seqüência de Fibonacci é definida indutivamente por:

$$F_n = \begin{cases} 0 & \text{se } n = 0 \\ 1 & \text{se } n = 1 \\ F_{n-1} + F_{n-2} & \text{se } n > 1 \end{cases}$$

e esta definição pode ser diretamente transformada num programa em *LP*:

```
procedimento fib(n):
    se n = 0 então devolva 0
             senão
    se n = 1 então devolva 1
             senão
    devolva fib(n − 1) + fib(n − 2)
```

A Figura 12 indica como os cálculos seriam executados para $n = 4$:

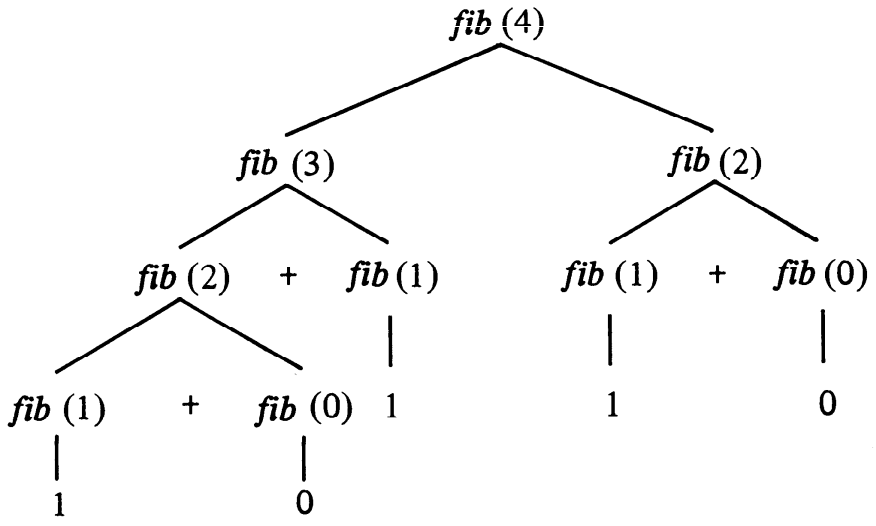

Figura 12

Devido à maneira como foi escrito o programa, vários valores são calculados mais do que uma vez.

Um conceito importante em programação é o de *eficiência* de programas, e que será discutido na Parte B deste texto. Adiantar-nos-emos um pouco, entretanto, e tentaremos avaliar a eficiência do programa acima. Uma medida aceitável de eficiência poderia ser o número total de operações primitivas (comparações, somas e subtrações) executadas para calcular F_n. Este número depende de n, e observando o programa, pode-se ver que será dado pela função:

$$c(n) = \begin{cases} 1 & \text{se } n = 0 \\ 2 & \text{se } n = 1 \\ 5 + c(n-1) + c(n-2) & \text{se } n > 1. \end{cases}$$

Esta é uma outra definição indutiva, cuja solução pode ser escrita em termos de números de Fibonacci:

$$c(n) = F_{n+2} + 5 \cdot (F_{n+1} - 1) \qquad (n \geq 0).$$

Pode-se obter uma fórmula fechada para $c(n)$ lembrando que

$$F_k = (\alpha^k - \beta^k)/\sqrt{5} \qquad (k \geq 0)$$

onde

$$\alpha = (1 + \sqrt{5})/2 \qquad \text{e} \qquad \beta = (1 - \sqrt{5})/2.$$

Usando o fato de que $|\beta| < 1$, para os valores crescentes de n obtém-se a aproximação:

$$c(n) \simeq 2{,}96 \cdot (1{,}62)^{n+1}.$$

Alguns exemplos dos valores de $c(n)$ são:

n	$c(n)$
25	803.378 (exato)
50	13×10^{10}
100	38×10^{20}

Para se ter uma idéia do valor de $c(100)$, basta dizer que um computador que realizasse um milhão de operações básicas por segundo, levaria cerca de $1{,}2 \times 10^8$ anos para calcular F_{100}! Na realidade, a memória deste computador teria sido esgotada muito antes. Por outro lado, o programa iterativo a seguir calcula F_n executando apenas $3 \cdot n + 1$ operações (isto é 301 operações para calcular F_{100}):

procedimento *fib*(n):
  início
    $x, y, k \leftarrow 0, 1, 0$;
    enquanto $k < n$ **faça** $x, y, k \leftarrow y, x + y, k + 1$;
    devolva x
  fim

Uma conclusão óbvia é que uma mesma função pode ser calculada por procedimentos de eficiência muito diferente (veja também o Exercício 7). Abordaremos novamente este problema na Parte B.

Uma pergunta que surge naturalmente é se os dois mecanismos, iteração e recursão, são em algum sentido equivalentes. A resposta a esta pergunta depende da definição exata do conceito de equivalência. Uma definição razoável seria a seguinte: dois programas são equivalentes se, e somente se, calculam a mesma função parcial, isto é, quando partindo dos mesmos dados, ou as execuções de ambos não terminam, ou então ambas terminam, sendo devolvidos os mesmos resultados.

Adotada esta definição, pode-se provar que um programa sempre pode ser transformado num programa recursivo equivalente. Na realidade, a discussão da iteração no início desta seção sugere a maneira de substituir a iteração por recursão. Sejam $m_1, \ldots, m_n$ e $x_1, \ldots, x_p$ os nomes dos argumentos e das variáveis usados na descrição de um procedimento, e "**enquanto** E **faça** S" o comando iterativo a ser eliminado desta descrição. Deve-se definir, então, o procedimento auxiliar:

procedimento $g(m_1, \ldots, m_n, y_1, \ldots, y_p)$:
  início
    $x_1, \ldots, x_p \leftarrow y_1, \ldots, y_p$;
    se E **então**
      início
        S; **devolva** $g(m_1, \ldots, m_n, x_1, \ldots, x_p)$
      fim
    senão devolva $x_1, \ldots, x_p$
  fim

O comando "**enquanto** E **faça** S" deve ser substituído por:

$$x_1, \ldots, x_n \leftarrow g(m_1, \ldots, m_n, x_1, \ldots, x_p).$$

Repetindo-se a aplicação desta transformação a todos os comandos iterativos do programa, chega-se a uma versão sem comandos iterativos.

Na prática, várias simplificações são possíveis pois nem todas as variáveis do programa precisam ser modificadas por *S*. Note-se que nesta transformação ignoramos totalmente a natureza das operações executadas pelo programa, e dos valores manipulados, fazendo uma transformação puramente simbólica do programa dado.

Aplicando-se esta transformação à versão iterativa do programa *fat*, obtém-se a seguinte versão recursiva equivalente:

```
procedimento fat(n):
    início
        s, x ← 1, 1;
        s, x ← g(n, s, x);
        devolva s
    fim

procedimento g(n, y1, y2):
    início
        s, x ← y1, y2;
        se x ≤ n então
                    início
                        s, x ← s • x, x + 1;
                        devolva g(n, s, x)
                    fim
                senão devolva s, x
    fim
```

A transformação contrária, isto é, de um programa que usa recursão em um programa que usa apenas iterações, depende da natureza dos objetos manipulados, e da existência de certas operações primitivas. Este problema está relacionado com a Tese de Church mencionada na Seção 2. No caso particular da *LP*, supondo a manipulação de inteiros e as operações primitivas de soma, subtração e comparação com zero, esta transformação será sempre possível se bem que bastante complicada. Deve-se notar que esta transformação pode ser feita de maneira sistemática, mas os programas iterativos obtidos têm essencialmente a mesma eficiência dos programas recursivos originais. Este fato deve ser contrastado com a transformação do procedimento *fib* indicada anteriormente. Como mostra o Exercício 7, a melhoria de eficiência não se deve à eliminação da recursão, mas sim à mudança do método de cálculo, ou seja do algoritmo. Pode-se provar que para

certas funções, como por exemplo a função $S(m_1, m_2, k)$ da Seção C.V.2, este tipo de melhoria é impossível.

Por outro lado, pode-se mostrar que num certo sentido a recursão é mais poderosa do que a iteração. Consideremos o seguinte exemplo:

procedimento $PH(m)$:
  se $r(m)$ **então devolva** $f(PH(g(m)), PH(h(m)))$
  senão devolva $k(m)$

em que não foi especificado o significado dos símbolos de função f, g, h e k, e de predicado r, nem o conjunto de objetos manipulados pelos mesmos. Note-se que este exemplo representa na realidade uma coleção de programas distintos, um para cada interpretação dos símbolos f, g, h, k e r. Em outras palavras, o exemplo representa um *esquema de programas*. Pode-se demonstrar então, que não existe um esquema puramente iterativo que seja equivalente ao esquema PH para todas as interpretações possíveis dos símbolos f, g, h, k e r.

Isto não quer dizer, entretanto, que para interpretações particulares não existem transformações equivalentes. Por exemplo, se o programa deve manipular números naturais, e:

$$f(y, z) \equiv y + z$$
$$g(y) \equiv h(y) \equiv y - 1$$
$$k(y) \equiv 1$$
$$r(y) \equiv y > 0$$

então o programa calcula 2^m, e um programa equivalente seria:

procedimento $T(m)$:
  início
    $y, s \leftarrow m, k(m)$;
    enquanto $r(y)$ **faça** $y, s \leftarrow g(y), f(s, s)$;
    devolva s
  fim

Também no caso da recursão podemos estabelecer uma regra de demonstração baseada no princípio de indução matemática. Para simplificar a exposição, suponhamos um programa composto de uma única definição recursiva da forma:

procedimento $f(m_1, \ldots, m_n)$: S

onde dentro do comando S ocorrem aplicações recursivas do procedimento f sob a forma $f(e_1, \ldots, e_n)$. Suponhamos, também, que deseja-se

demonstrar a proposição $P(m_1, \ldots, m_n, f(m_1, \ldots, m_n))$ que relaciona os dados $m_1, \ldots, m_n$ com os resultados $f(m_1, \ldots, m_n)$. Para cada chamada $f(e_1, \ldots, e_n)$ pode-se adotar, então, a hipótese de indução $P(e_1, \ldots, e_n, f(e_1, \ldots, e_n))$.

Evidentemente, esta regra corresponde à indução sobre o número de vezes que o procedimento f é aplicado de maneira recursiva durante o cálculo de $f(m_1, \ldots, m_n)$. Note-se que, como é comum em provas por indução em geral, a parte mais difícil da demonstração é achar uma hipótese de indução conveniente, seja no caso da iteração ou da recursão. A verificação da hipótese é, em geral, bastante simples.

Note-se, também, que as regras de demonstração estabelecidas permitem relacionar dados com resultados, caso a computação termine. As provas de terminação de programas são freqüentemente feitas à parte, muitas vezes por indução sobre o valor dos argumentos, como foi feito no caso do Algoritmo de Euclides.

5. Exemplos de aplicação

Mostraremos, nesta seção, vários exemplos de aplicação do princípio de indução, tanto sob a forma discutida na seção anterior, como sob outras formas.

EXEMPLO 1. Neste exemplo demonstraremos que o programa iterativo *Euclides* apresentado na Seção 3 realmente calcula o *mdc* de dois inteiros positivos m e n. A demonstração será feita indicando-se a relação existente entre m, n, e os valores das variáveis do programa antes e depois de cada comando. Estas relações, dadas pelas proposições Q_0 a Q_5, estão indicadas no texto do programa da Figura 13.

Q_0 é uma proposição verdadeira por hipótese. A execução de "$x, y \leftarrow m, n$" acarreta Q_1, e Q_1 implica de maneira trivial Q_2. Q_3 resulta de Q_2, e da definição do resto da divisão de dois inteiros positivos. Para concluir Q_4, basta tomar a proposição Q_3 e substituir y por x e r por y devido à execução de "$x, y \leftarrow y, r$". Na proposição assim obtida basta considerar os casos $y = 0$ e $y > 0$, e aplicar algumas propriedades do *mdc* para obter Q_4. Finalmente, vem a verificação do passo indutivo, ao notar que Q_4 e $r \neq 0$ implicam Q_2. Q_5 é uma conseqüência trivial de Q_4 e $y = 0$.

Fica provado, então, que o valor calculado é o $mdc(m, n)$ se a execução termina, e este fato já foi provado na Seção 2. Note-se que as

procedimento *Euclides*(m, n):
  início $\{Q_0 : m > 0 \wedge n > 0\}$
    $x, y \leftarrow m, n;$
      $\{Q_1 : m > 0 \wedge n > 0 \wedge x = m \wedge y = n\}$
    repita
      $\{Q_2 : m > 0 \wedge n > 0 \wedge x > 0 \wedge y > 0 \wedge mdc(m, n) =$
        $= mdc(x, y)\}$
      $r \leftarrow resto\ (x, y);$
      $\{Q_3 : m > 0 \wedge n > 0 \wedge x > 0 \wedge y > 0 \wedge mdc(m, n) =$
        $= mdc(x, y) \wedge \exists q(q \cdot y + r = x \wedge 0 \leq r < y)\}$
      $x, y \leftarrow y, r$
      $\{Q_4 : m > 0 \wedge n > |0 \wedge x > 0 \wedge y = r \wedge [y = 0 \wedge$
      $\wedge mdc(m, n) = x \vee y > 0 \wedge mdc(m, n) = mdc(x, y)]\}$
    até que $r = 0;$
      $\{Q_5 : mdc(m, n) = x\}$
    devolva x
  fim

Figura 13

proposições Q_2 e Q_4 correspondem às proposições P e Q usadas para formular a regra de demonstração para o comando "**repita** ... **até que** ...".

EXEMPLO 2. A Figura 14 indica as proposições necessárias para demonstrar que o programa *fib*, apresentado na Seção 4, calcula F_n segundo a sua definição indutiva.

procedimento *fib*(n):
  início
      $\{n \geq 0\}$
    $x, y, k \leftarrow 0, 1, 0;$
      $\{n \geq 0 \wedge x = F_0 \wedge y = F_1 \wedge k = 0\}$
    enquanto
      $\{n \geq 0 \wedge x = F_k \wedge y = F_{k+1} \wedge 0 \leq k \leq n\}$
      $k < n$
      faça
        $x, y, k \leftarrow y, x + y, k + 1$
        $\{n \geq 0 \wedge x = F_k \wedge y = F_{k+1} \wedge 0 \leq k \leq n\};$
      $\{n \geq 0 \wedge x = F_n\}$
    devolva x
  fim

Figura 14

Para mostrar que o programa termina basta verificar que dentro do único comando repetitivo, os valores sucessivos da variável k formam a seqüência 0, 1, 2, 3, ..., e como $n \geq 0$, depois de um número finito de repetições, exatamente n, teremos $k = n$.

EXEMPLO 3. Consideremos o seguinte programa[3]:

procedimento $f(n)$:
  se $n = 1$ **então devolva** 1
    senão
  se $resto(n, 2) = 0$ **então devolva** $n \div 2$
    senão devolva $f(f((3 \cdot n + 1) \div 2))$

onde n é um inteiro positivo, e $\div$ representa a divisão de inteiros. Este exemplo é interessante, pois não se sabe se o cálculo de $f(n)$ termina para qualquer $n \geq 1$. É instrutivo calcular o valor de f para alguns argumentos; assim (omitindo os parênteses):

$$f(33) = ff50 = f25 = ff38 = f19 = ff29 = fff44 = ff22 = \\ = f11 = ff17 = fff26 = ff13 = fff20 = ff10 = f5 = \\ = ff8 = f4 = 2.$$

A proposição a ser demonstrada neste caso é:

Se $n > 1$ então ou $f(n)$ não está definido (isto é, o cálculo de $f(n)$ não termina), ou então $f(n) \leq n \div 2$.

Usaremos a regra de demonstração dada na Seção 4 para programas recursivos, sendo que a proposição acima será a hipótese de indução. Seja um inteiro $n > 1$. Se o cálculo de $f(n)$ não termina então não há nada a demonstrar. Caso contrário, consideremos as duas possibilidades:

(a) n é par: $f(n) = n \div 2 \leq n \div 2$

(b) n é ímpar: $f(n) = f(a)$ com $a = f((3 \cdot n + 1) \div 2)$. Como o cálculo de $f(n)$ termina, então terminam também os de a e de $f(a)$. Por hipótese de indução aplicada duas vezes tem-se então:

$$a \leq (3 \cdot n + 1) \div 4$$

e

$$f(a) \leq a \div 2 \leq (3 \cdot n + 1) \div 8.$$

É fácil mostrar que para todo $n \geq 1$ tem-se $(3 \cdot n + 1) \div 8 \leq n \div 2$. Portanto, $f(n) \leq n \div 2$.

---

(3) O programa foi proposto por Morris [86].

EXEMPLO 4. Consideremos o seguinte programa[4]:

procedimento $g(n)$:
se $n > 100$ **então devolva** $n - 10$
senão devolva $g(g(n + 11))$

onde n é um inteiro qualquer. Demonstraremos que a função g definida pelo procedimento é igual à função h definida por:

$$h(n) = \begin{cases} n - 10 & \text{se } n > 100 \\ 91 & \text{se } n \leq 100. \end{cases}$$

Consideremos os três casos possíveis:

(a) $n > 100$: neste caso $g(n) = n - 10 = h(n)$.

(b) $90 \leq n \leq 100$: neste caso o valor de $g(n)$ será dado por $g(g(n + 11))$, onde as duas ocorrências de g correspondem a chamadas recursivas do procedimento; usando, portanto, a hipótese de indução duas vezes, teremos:

$$g(n) = g(g(n + 11)) = g(h(n + 11)) = h(h(n + 11)).$$

Por outro lado, $n + 11 > 100$ e, portanto:

$$h(h(n + 11)) = h(n + 1) = 91 = h(n)$$

pela definição da função h. Temos finalmente: $g(n) = h(n)$.

(c) $n < 90$: ainda neste caso, aplicando a hipótese de indução duas vezes, teremos:

$$g(n) = g(g(n + 11)) = g(h(n + 11)) = h(h(n + 11)).$$

Como $n + 11 \leq 100$, temos:

$$h(h(n + 11)) = h(91) = 91 = h(n).$$

Finalmente: $g(n) = h(n)$.

O que acabamos de demonstrar é, na realidade, que para todo inteiro n ou o cálculo de $g(n)$ não termina, ou então $g(n) = h(n)$. A demonstração de que o cálculo de $g(n)$ sempre termina pode ser feita considerando os três casos acima, e fazendo indução conveniente sobre os argumentos.

---

(4) O programa foi proposto por Burstall [12].

EXEMPLO 5. Suponhamos que em *LP* podemos manipular, além de inteiros, também seqüências de inteiros, e que para isto existem as seguintes operações primitivas:

$$car(\langle s_1, s_2, \ldots, s_n\rangle) = s_1 \text{ se } n \geq 1$$
$$cdr(\langle s_1, s_2, \ldots, s_n\rangle) = \langle s_2, \ldots, s_n\rangle \text{ se } n \geq 1$$
$$s \,\&\, \langle s_1, \ldots, s_n\rangle = \langle s, s_1, \ldots, s_n\rangle$$
$$\langle s_1, \ldots, s_n\rangle \,\#\, s = \langle s_1, \ldots, s_n, s\rangle$$
$$\langle s_1, \ldots, s_n\rangle\,[i] = s_i \text{ para } i = 1, \ldots, n.$$

A seqüência de zero elementos será indicada por $\langle\ \rangle$. Algumas dessas operações serão utilizadas no exemplo seguinte.

O programa da Figura 15 produz a seqüência reversa a partir da seqüência t dada. As proposições Q_0 a Q_4 inseridas no texto do programa indicam a maneira de mostrar que o programa produz resultados corretos:

procedimento *reverso*(t):
  início
    $\{Q_0\}$
    $x, y \leftarrow \langle\ \rangle, t;$
    $\{Q_1\}$
    enquanto
      $\{Q_2\}$
      $y \neq \langle\ \rangle$
    faça
      $x, y \leftarrow car(y)\&x, cdr(y)$
      $\{Q_3\};$
    $\{Q_4\}$
    devolva x
  fim

Figura 15

$Q_0 : t = \langle s_1, \ldots, s_n\rangle \wedge n \geq 0$
$Q_1 : t = \langle s_1, \ldots, s_n\rangle \wedge n \geq 0 \wedge x = \langle\ \rangle \wedge y = t$
$Q_2 : t = \langle s_1, \ldots, s_n\rangle \wedge n \geq 0 \wedge \exists\, i[0 \leq i \leq n \wedge x =$
$= \langle s_i, s_{i-1}, \ldots, s_1\rangle \wedge y = \langle s_{i+1}, s_{i+2}, \ldots, s_n\rangle]$
$Q_3 \equiv Q_2$ (hipótese de indução)
$Q_4 : t = \langle s_1, \ldots, s_n\rangle \wedge n \geq 0 \wedge x = \langle s_n, s_{n-1}, \ldots, s_1\rangle$

EXEMPLO 6. Neste exemplo, mostraremos uma maneira indutiva de resolver um problema[5], garantindo assim que o programa obtido será correto, isto é, satisfará as especificações do problema.

Seja M o menor conjunto que contém o elemento 1, e está fechado sob as operações $g(x) = 2x + 1$ e $f(x) = 3x + 1$. Deseja-se escrever um programa que constrói a seqüência dos n menores elementos de M ($n \geq 1$), em ordem crescente. Por exemplo, para $n = 15$ teríamos:

$$\langle 1, 3, 4, 7, 9, 10, 13, 15, 19, 21, 22, 27, 28, 31, 39 \rangle.$$

Para resolver o problema, suponhamos que já conseguimos construir a seqüência parcial:

$$\langle s_1, s_2, \ldots, s_m \rangle \qquad 1 \leq m \leq n.$$

É fácil verificar que existem p e q tais que $1 \leq p, q \leq m$, e

$$\begin{aligned} &g(s_i) \leq s_m \quad \text{para todo} \quad 1 \leq i < p \\ &f(s_i) \leq s_m \quad \text{para todo} \quad 1 \leq i < q \\ &g(s_p) > s_m \\ &f(s_q) > s_m. \end{aligned}$$

Em outras palavras, o menor entre $g(s_p)$ e $f(s_q)$ será o valor de s_{m+1}, e os novos valores de p e q serão:

$$p', q' = \begin{cases} p, q + 1 & \text{se } f(s_q) < g(s_p) \\ p + 1, q & \text{se } f(s_q) > g(s_p) \\ p + 1, q + 1 & \text{se } f(s_q) = g(s_p). \end{cases}$$

Foi feito, dessa maneira, o passo indutivo, estendendo-se com mais um elemento a seqüência parcial, e calculando-se os novos valores de p e q, a partir dos anteriores. É fácil verificar que os valores $m = p = q = 1$ e a seqüência $\langle 1 \rangle$ constituem uma base conveniente para esta indução. Todas estas considerações levam diretamente à construção do programa apresentado na Figura 16.

EXEMPLO 7. Vimos na Seção 3 como definições indutivas podem ser usadas para descrever a forma dos programas numa linguagem de programação. Mostraremos neste exemplo como estas definições podem ser usadas para escrever programas que decidem se uma dada seqüência de símbolos é ou não um programa. Indicaremos apenas o procedimento que reconhece comandos em *LP*, e

(5) Este problema aparece em Wirth [130].

```
procedimento h(n):
    início
        p, q, m, t ← 1, 1, 1, ⟨1⟩;
        enquanto m < n faça
            início
            a, b ← 2 • t[p] + 1, 3 • t[q] + 1;
                m ← m + 1;
                se a < b então p, t ← p + 1, t # a
                          senão
                se a > b então q, t ← q + 1, t # b
                          senão
                                p, q, t ← p + 1, q + 1, t # a
            fim;
        devolva t
    fim
```

Figura 16

suporemos que já foi definido o procedimento que reconhece expressões. O argumento do procedimento será sempre uma seqüência de símbolos da linguagem *LP* da forma $\langle s_1, s_2, \ldots, s_m \rangle$. Suporemos que símbolo s_m será sempre um símbolo especial □ que não faz parte de nenhum comando. Desta maneira evitaremos que em alguns pontos do programa a função *car* seja aplicada a uma seqüência vazia. Se para algum valor i ($1 \leq i \leq m - 1$), a seqüência $\langle s_1, s_2, \ldots, s_i \rangle$ representa um comando, então o resultado do procedimento é a seqüência restante $\langle s_{i+1}, \ldots, s_m \rangle$ (pode-se demonstrar que tal i é único). Se não existir tal i, então o resultado será $\langle$"**erro**"$\rangle$. Suporemos que o procedimento que reconhece expressões tem funcionamento semelhante. O predicado *var*(x) indica se o símbolo x é o nome de uma variável. O procedimento está apresentado na Figura 17.

```
procedimento comando(t):
    se car(t) = "nada"
        então devolva cdr(t)
        senão
    se var(car(t))
        então início
                  v ← cdr(t);
```

Figura 17 (continua)

```
            enquanto car(v) = "," faça
               se var(car(cdr(v))) então v ← cdr(cdr(v))
                                   senão devolva ⟨"erro"⟩;
            se car(v) = "←"
               então repita
                        v ← expressão(cdr(v));
                        se v = ⟨"erro"⟩ então devolva v
                                       senão nada
                     até que car(v) ≠ ","
               senão devolva ⟨"erro"⟩;
            devolva v
         fim
   senão
se car(t) = "devolva"
   então início
            v ← t;
            repita
               v ← expressão(cdr(v));
               se v = ⟨"erro"⟩ então devolva v
                              senão nada
            até que car(v) ≠ ",";
            devolva v
         fim
   senão
se car(t) = "se"
   então início
            v ← expressão(cdr(t));
            se car(v) ≠ "então"
               então devolva ⟨"erro"⟩
               senão início
                        v ← comando(cdr(v));
                        se car(v) ≠ "senão" então devolva ⟨"erro"⟩
                                           senão devolva
                                                 comando(cdr(v))
                     fim
         fim
   senão
se car(t) = "enquanto"
```

Figura 17 (continua)

```
        então início
                  v ← expressão(cdr(t));
                  se car(v) ≠ "faça" então devolva ⟨"erro"⟩
                                    senão devolva comando(cdr(v))
              fim
        senão
    se car(t) = "repita"
        então início
                  v ← t;
                  repita
                      v ← comando(cdr(v))
                  até que car(v) ≠ ";";
                  se car(v) ≠ "até" ∨ car(cdr(v)) ≠ "que"
                      então devolva ⟨"erro"⟩
                      senão devolva expressão(cdr(cdr(v))
              fim
        senão
    se car(t) = "início"
        então início
                  v ← t;
                  repita
                      v ← comando(cdr(v))
                  até que car(v) ≠ ";";
                  se car(v) ≠ "fim" então devolva ⟨"erro"⟩
                                    senão devolva cdr(v)
              fim
        senão devolva ⟨"erro"⟩
```

Figura 17 (conclusão)

EXERCÍCIOS

1. Descreva em *LP* um procedimento que verifica se um inteiro positivo dado é ou não primo. O seu procedimento é um algoritmo?
2. Descreva em *LP* um procedimento que calcula o número de divisores distintos de um inteiro positivo.
3. Descreva um procedimento cuja execução termina se, e somente se, o Último Teorema de Fermat é falso. Se a execução terminar, o procedimento deve devolver como resultados valores inteiros positivos x, y e z e valor inteiro $k > 2$ tais que $x^k + y^k = z^k$.

4. A construção indicada através do diagrama de blocos da Figura 18 não tem um equivalente direto em *LP*. Indique como tal construção poderia ser representada em *LP*, sem introduzir um comando repetitivo novo.

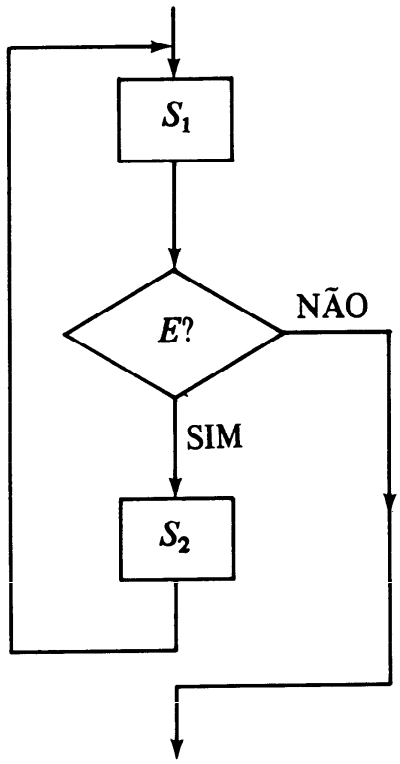

Figura 18

5. Indique as funções calculadas pelos seguintes programas:

(a) **procedimento** $f(n)$:
início
$k, s \leftarrow 0, 1$;
enquanto $k < n$ **faça** $s, k \leftarrow 2 \cdot s, k + 1$;
devolva s
fim

(b) **procedimento** $c(m, n)$:
início
$k, s \leftarrow m, 1$;
repita
$s, k \leftarrow s \cdot k, k + 1$
até que $k > n$;
devolva s
fim

(c) **procedimento** *sr*(*s*, *t*):
início
$n \leftarrow s[1] \cdot t[2] + s[2] \cdot t[1]$;
$d \leftarrow s[2] \cdot t[2]$;
$k \leftarrow$ *Euclides*(n, d);
devolva $\langle n \div k, d \div k \rangle$
fim

Neste último programa, s e t são pares de números inteiros positivos; *Euclides* é o nome do procedimento definido na Seção 3.

6. Mostre que dado um inteiro K, é sempre possível escolher valores m e n tais que o número de divisões executadas pelo Algoritmo de Euclides é maior que K.

7. Escreva um programa para calcular o n-ésimo número de Fibonacci, sem usar comandos iterativos, e executando apenas $3 \cdot n + 1$ operações primitivas para calcular F_n.

8. Determine a função $f(m, n)$ que indica o número de vezes que o procedimento *fib* da Figura 11 é chamado com argumento m durante o cálculo de *fib*(n), $n \geq m \geq 0$.

9. Ache interpretações dos símbolos f, g, h, k e r para que o programa *PH* da Seção 4 calcule as funções
(a) $2 \cdot m$
(b) F_m
(c) $m \div 2$.

10. Escreva um programa não recursivo que calcula a mesma função que o procedimento f do Exemplo 3 da Seção 5.

11. Mostre que o procedimento apresentado a seguir devolve o máximo divisor comum e o mínimo múltiplo comum de inteiros positivos m e n:

procedimento *mdcmmc*(m, n):
início
$x, y, v, w \leftarrow m, n, n, m$;
enquanto $x \neq y$ **faça**
se $x > y$ **então** $x, w \leftarrow x - y, v + w$
senão $y, v \leftarrow y - x, v + w$;
devolva $x, (v + w) \div 2$
fim

Mostre, também, que a execução do procedimento termina para quaisquer valores inteiros positivos m e n.

12. Complete o programa da Figura 17, incluindo a verificação de que os comandos de atribuição são da forma indicada na Seção 3:

$$x_1, \ldots, x_n \leftarrow E_1, \ldots, E_n \qquad (n \geq 1)$$

ou

$$x_1, \ldots, x_n \leftarrow f(E_1, \ldots, E_p) \quad (n \geq 2, p \geq 0)$$

13. Usando a notação FNB, defina o conjunto de seqüências de expressões formadas por nomes de variáveis, símbolos de operações $+$, $-$, $\cdot$ e $\div$, e parênteses. Escreva um procedimento que reconhece expressões análogo ao do Exemplo 7 da Seção 5.

NOTAS BIBLIOGRÁFICAS

Uma discussão em nível introdutório dos conceitos de procedimento e algoritmo pode ser encontrada em Knuth [60]. Davis [18] e Rogers [101] são dois textos mais avançados, e clássicos, sobre a teoria da computabilidade.

Existem inúmeras publicações sobre linguagens de programação e sua utilização. Recomendamos, em particular, o texto de Wirth [130]. A notação FNB foi usada pela primeira vez em Naur [87] para descrever a linguagem Algol 60.

A literatura sobre a demonstração de propriedades de programas é bastante vasta, mas não muito consolidada. O texto de Wirth mencionado acima introduz estas noções. Outras publicações recomendadas são Elspas et al. [27], Manna [77] e Manna et al. [78]. O problema da relação entre a iteração e a recursão é discutido de uma maneira mais profunda em Paterson e Hewitt [93] e Walker e Strong [125].