CBPF-MO-001/95

PROGRAMAÇÃO EM MAPLE*

Renato Portugal†
Centro Brasileiro de Pesquisas Físicas
Departamento de Relatividade e Partículas
Rua Dr. Xavier Sigaud 150
22290-180 Rio de Janeiro – RJ, Brazil

*Versão preliminar

†Email: portugal@lca1.drp.cbpf.br

Contents

Prefácio

Essa apostila é uma compilação das notas de aula dos cursos de Maple que foram dados em 1994 no CBPF. Esses cursos tinham duas partes, que versavam sobre o uso do Maple como calculadora e sobre programação, respectivamente. Essa apostila cobre a segunda parte desses cursos. Assim, estamos supondo que o leitor já tenha familiaridade com o Maple. Para aqueles que estão iniciando, recomendamos o livro **Introduction to Maple**, citado nas Referências. Em particular, os capítulos 8 e 12 são importantes para a compreensão dessa apostila. Eles tratam de funções e estrutura de dados respectivamente.

Os assuntos tratados aqui cobrem uma grande parte da teoria de programação do Maple.

Cada capítulo tem uma grande quantidade de exercícios que servem para fixar e complementar o que foi apresentado no texto. Vários exercícios tem sugestões para ajudar os leitores que estão tendo dificuldades na resolução. As sugestões são bastante sucintas e portanto estamos prevendo que só ajudarão quem estiver perto da solução.

O uso do HELP ON LINE é quase que obrigatório para esclarecer o funcionamento de comandos que não foram suficientemente explicados no texto, para tirar dívidas mais detalhadas de algum comando ou para descobrir qual é o comando adequado para executar uma certa tarefa, o que não foi citado no texto.

O autor está engajado num projeto de tornar essa apostila autosuficiente, e a previsão para o lançamento da segunda versão é julho de 1995. Lá esperamos apresentar um trabalho mais ameno aos leitores.

As sugestões dos leitores são bem-vindas. Elas podem ser enviadas por correio eletrônico para o endereço portugal@lca.drp.cbpf.br.

Os pedidos de cópia dessa apostila podem ser enviados ao CBPF para o endereço:

Centro Brasileiro de Pesquisas Físicas - CBPF
Área de Publicações
Rua Dr. Xavier Sigaud, 150
22290-180 - Rio de Janeiro, RJ – Brasil

ou para Socorro@cbpfsu1.cat.cbpf.br

ou para o autor. Por favor, mandem o endereço completo para onde as cópias deverão ser enviadas.

10 de Março de 1995
CBPF

1 Programação Funcional

1.1 Introdução

Existem diferentes estilos de programação no Maple. A programação através de funções é útil para programas simples, que podem ser escritos com apenas um comando. As funções são geradas através do comando *unapply*( ) ou através do operador seta. Nesse capítulo, além desses comandos, vamos descrever os comandos *map*( ) e *select*( ) que estão relacionados com programação funcional, seja desenvolvimento do algoritmo do programa, seja porque utilizam funções como argumentos. Veremos também como fazer manipulações algébrica com funções, como adição, produto e composição de funções. Uma atenção especial deve ser dada à funções encaixadas, isto é, funções que aparecem dentro de outras funções.

A programação funcional também é útil quando queremos utilizar comandos que tem funções como argumento, como é o caso dos comandos *map*( ), *select*( ), *matrix*( ) e *zip*( ) entre outros. Se quisermos selecionar elementos de um conjunto ou partes de uma expressão que seguem uma certa relação funcional, podemos executar essa tarefa criando uma função adequada para ser usada dentro do comando *select*( ). O poder desse comando será ampliado de acordo com a nossa capacidade de criar essas funções. O mesmo se aplica na hora de gerar uma matriz onde os elementos seguem uma regra de formação. A regra de formação será dada pela função.

Vamos começar com um comando importantíssimo do Maple.

1.2 O Comando map

Suponha que temos a expressão

$$(x^2 - 2\ x + 1)^2 + (x^2 + 2\ x + 1)^2$$

e queremos colocá-la na forma

$$(x-1)^4 + (x+1)^4\ .$$

Uma vez que essa expressão tem dois operandos $(x^2 - 2x + 1)^2$ e $(x^2 + 2x + 1)^2$, podemos como primeira tentativa isolar cada um deles com o comando *op*( ) e fatorá-los separadamente:

```
> expr1 := (x^2 - 2*x + 1)^2 + (x^2 + 2*x + 1)^2 ;
```

$$expr1 := (x^2 - 2\ x + 1)^2 + (x^2 + 2\ x + 1)^2$$

```
> factor(op(1, expr1)) + factor(op(2, expr1)) ;
```

$$(x-1)^4 + (x+1)^4$$

Essa maneira de partir uma expressão em geral é trabalhosa e pode induzir a erros. A maneira mais simples é usar o comando *map (mapping)*, da seguinte forma:

```
> map(factor, expr1) ;
```

$$(x-1)^4 + (x+1)^4$$

O comando $map(fator,\ expr)$ aplica a função $factor(\ )$ sobre cada operando da expressão.

Vamos supor agora que temos uma lista de funções na variável x e queremos construir a lista das derivadas de cada função em relação a x:

$$> \ L := [sin(x),\ a * \exp(-x^2),\ GAMMA(x),\ f(x) * g(x),\ a,\ x]\ ;$$

$$L := [sin(x),\ a\ e^{x^2},\ \Gamma(x),\ g(x)\ f(x),\ a,\ x]$$

Como primeira tentativa podemos diferenciar cada função, por exemplo, através de um loop:

```
> for i to nops (L) do
>   diff(L[i],x)
> od ;
```

$$\cos(x)$$

$$-2\ a\ x\ E^{-x^2}$$

$$\Psi(x)\ \Gamma(x)$$

$$(\frac{\partial}{\partial x}\ f(x))\ g(x) + f(x)\ (\frac{\partial}{\partial x}\ g(x))$$

$$0$$

$$1$$

Observe que o resultado desse comando não está na forma de uma lista das derivadas. Para tal, temos que fazer da seguinte forma:

```
> diffL : = [ ];        #iniciação da variavel diffL
```

$$diffL := [\ ]$$

```
>  for i to nops (L) do
>       diffL := [op(diffL),  diff(L[i], x)]
> od:
> diffL  ;
```

$$\left[\cos(x), -2axE^{-x^2}, \Psi(x)\Gamma(x), \left(\frac{\partial}{\partial x}f(x)\right)g(x) + f(x)\left(\frac{\partial}{\partial x}\ g(x)\right), 0, 1\right]$$

A maneira mais simples de se obter o mesmo resultado é usar o comando $map(\ )$, porém aqui temos uma novidade

```
> map(diff, L);
Error, wrong number (or type) of parameters in function diff
```

O Maple reclama dos parâmetros dados para a função *diff( )*. Eles podem ter o tipo errado ou número errado. Aqui o número de parâmetros está errado pois a função *diff( )* precisa pelo menos dois parâmetros. Falta a variável de diferenciação. Essa variável é repassada como terceiro argumento do comando $map(\ )$, da seguinte forma:

$> map(diff, L, x);$

$$\left[\cos(x), -2axe^{-x^2}, \Psi(x)\Gamma(x), \left(\frac{\partial}{\partial x}f(x)\right)g(x) + f(x)\left(\frac{\partial}{\partial x}\ g(x)\right), 0, 1\right]$$

Esse último comando é equivalente à:

$$[diff(op(1, L), x),\ diff(op(2, L), x), \cdots,\ diff(op(6, L), x)]\ .$$

A sintaxe do comando $map(\ )$ é:

$$map(\text{ função , expressão , argumentos extras da função })$$

O comando $map(\ )$ faz com que a função seja aplicada sobre cada operando da expressão e o resultado é uma expressão do mesmo tipo. Ou seja, se a expressão for uma lista, o resultado será uma lista, se a expressão for do tipo algébrico, o resultado será uma expressão algébrica. Caso a função tenha mais de um argumento, eles são especificados depois da expressão.

Qual é a saída do comando $map(f, a/b)$?. Para prever qual é o resultado do comando $map(\ )$ sobre uma expressão, é necessário saber quais são os operandos desta expressão. Para isso usa-se o comando *op* (operands). Por exemplo, quais são os operandos das expressão a/b?

$> op(a/b);$

$$a, 1/b$$

$> whattype(expr)\ ;$

$$*$$

Isso explica porque $map(f, a/b)$ não retorna $f(a)/f(b)$, já que os operandos não são a e b mas sim a e $1/b$. O comando $whattype(\ )$ fornece o operador de superfície que no caso da expressão a/b é ' $*$ '.

1.3 Unapply

Queremos definir uma função matemática, como por exemplo:

$$f(x, y) = x^2 - a\ y^2\ .$$

Existem várias maneiras de construir novas funções no Maple[1]. Vamos começar usando o comando $unapply(\ )$:

$> f := unapply(x\hat{}2 - y\hat{}2, x, y)\ ;$

$$f := (x, y) \mathrel{-\!\!>} x^2 - a\ y^2$$

O resultado quer dizer que x, y são as variáveis, f é o nome da função e $x^2 - a\ y^2$ é a expressão da função uma vez dado x e y. Assim:

[1] A lista das funções matemáticas já conhecidas pelo Maple é obtida com o comando ?inifcns.

> $f(2,3)$;

$$4 - 9\ a$$

> $f(u,v)$;

$$u^2 - 9\ v^2$$

Na expressão de uma função, as variáveis tem um papel diferente dos outros termos. Elas podem ser substituidas por qualquer valor númerico ou algébrico de forma simples:

> $f(valor1, valor2)$;

$$valor1^2 - a\ valor2^2$$

Por sua vez, o termo a que aparece na expressão de f só pode ter seu valor modificado com o comando

> $subs(a = novo\_valor,\ f(x,y))$;

$$x^2 - novo\_valor\ y^2$$

ou através de uma atribuição:

> $a\ :=\ novo\_valor$:
> $f(x,y)$;

$$x^2 - novo\_valor\ y^2$$

A sintaxe do comando *unapply*( ) é

unapply(expressão, sequência de variáveis)

e o resultado é

(sequência de variáveis avaliada) –> expressão avaliada

Veja outro exemplo:

> $b := Pi$:
> $unapply(sin(n * b), n)$;

$$n \text{ -> } sin(n\ Pi)$$

Nesse caso temos uma função anônima, pois nenhum nome foi dado a ela. As funções anônimas são muito úteis, como veremos adiante. Observe que b foi substituído por Pi, pois a expressão $sin(n * b)$ é avaliada antes de se criar a função.

O principal uso do comando *unapply*( ) é transformar uma expressão previamente calculada em uma função: Vamos supor que após uma série de comandos no Maple obtemos uma expressão como resultado que contém as variáveis x e y. Queremos agora transformar essa expressão em uma função nas variáveis x e y sem digitá-la novamente. Podemos atribuir à expressão a variável *expr* e então usar o comando *unapply( )* da seguinte forma

$\vdots$

$expr := expressão;$

$nome\ da\ função := unapply(expr, x, y)$

O resultado será

$$nome\ da\ função :=\ (x, y) \text{ -> } valor\ da\ expressão$$

Podemos também usar as aspas duplas da forma nome da função :=unapply(",x,y). As aspas duplas fornecem o último resultado calculado. Neste caso não é necessário atribuir a expressão a uma variável.

Em alguns casos temos que tomar cuidado com a avaliação da expressão. Vamos supor que queremos definir a função *soma* que tem uma lista de números como argumento e que retorna a soma do primeiro elemento com o último. Por exemplo:

$$>\ soma([10, 8, -3])\ ;$$

$$7$$

Se definirmos essa função da forma

$$>\ soma :=\ unapply(lista[1] +\ lista[nops(lista)],\ lista)\ :$$

não obteremos o resultado correto, de fato:

$$>\ soma([10, 8, -3])\ ;$$

$$20$$

Se verificarmos o corpo da função

$$> print(soma)\ ;$$

$$lista \text{ -> } 2\ lista[1]$$

podemos ver que a expressão $lista[1] +\ lista[nops(lista)]$ foi avaliada em $2\ lista[1]$ pois $nops(lista)$ é 1. Se *lista* tivesse um certo valor previamente atribuído a ela, o resultado poderia ser mais diferente ainda do que queríamos. Para obter a função correta, temos que inibir a avaliação da expressão:

$$>\ soma :=\ unapply('lista[1] +\ lista[nops(lista)]',\ 'lista')\ ;$$

$$soma :=\ lista \text{ -> } lista[1] +\ lista[nops(lista)]$$

Nesse último exemplo é mais conveniente usar o operator -> para definir essa função, como veremos na seção seguinte.

Um erro comum é tentar definir uma função, por exemplo $g(x, y) = ax^2 + bx + c$, da seguinte forma

$$>\ g(x)\ :=\ a * x\hat{}2 + b * x + 2\ ;$$

$$g(x) \;:=\; a\ x^2 + b\ x + c$$

Observe que $g(0)$ e $g(t)$ não retornam o valor esperado:

$>\ g(0), g(t), g(x)$

$$g(0),\ g(t),\ a\ x^2 + b\ x + c$$

Quando se faz uma atribuição dessa forma, o termo 'x' não é uma variável da função. Vamos ver melhor o que está acontecendo. Para saber quem é g, use o comando *op* (ou *eval*, ou *print*):

$>\ op(g)\ ;$

$$proc(\ )\ options\ remember;\ 'procname(args)'\quad end$$

Devido a atribuição $g(x) := a * x\hat{}2 + b * x + 2$, g passou a ser um procedimento que tem como saída o resultado do comando $procname(args)$. $Procname$ se refere ao nome do procedimento que aqui é g, e $args$ se refere aos argumentos de g. Por isso o comando $g(0)$ retorna $g(0)$. Ou seja esse procedimento não faz nada a menos de um detalhe. A opção *remember table* faz com que g passe a ter uma *remember table* associada. A *remember table* é o quarto operando de um procedimento. Assim:

$>\ op\ (4, op(g))\ ;$

$$table([$$
$$x = ax^2 + bx + c$$
$$t = g(t)$$
$$0 = g(0)$$
$$)]$$

A atribuição $g(x) := a * x\hat{}2 + b * x + c$ fez com que a expressão $ax^2 + bx + c$ fosse colocada na *remember table* de g. Quando se pede o valor de $g(x)$, a *remember table* é consultada antes de qualquer verificação de quem seja a função. Cada nova atribuição terá seu valor acrescentado na *remember table* assim como cada nova chamada de g.

1.4 O Operador Seta

Podemos criar funções matemática com o operador seta, da seguinte forma

$>\ g := (x, y) \text{-}\!> sin(x + y)\ ;$

$$g := (x, y) \text{-}\!> sin(x + y)$$

Como antes, podemos fazer:

$>\ g(Pi/2, Pi/2)\ ;$

> $g(a, Pi/2)$;

$$cos(a)$$

A sintaxe do operador –> é:

(sequência de variáveis) –> *local* sequência de strings; expressão

O comando *local* sequência de strings raramente é necessário no uso do operador seta. Vamos dar um exemplo onde é necessário este comando. Suponha que queremos criar a função *somatorio(n)* que calcula:

$$\sum_{i=0}^{5} n^i$$

Essa função pode ser definida como

> $f := n\text{-}> sum(n\hat{}i, i = 0..5)$;

$$f := n \text{ -> } \sum_{i=0}^{5} n^i$$

Por exemplo:

> $somatorio(10)$;

111111

Suponha agora que devido a algum cálculo anterior a variável i sofreu atribuição de um número. Veja o que ocorre:

> $i := 1$;

$$i := 1$$

> $somatorio(10)$;
Error, (in sum) summation variable previously assigned, second argument evaluates to, $1 = 0..5$

Uma vez que o comando *sum( )* avalia seus argumentos, a variável i, que é o indice da soma, foi substituido por 1, por isso a temos a mensagem de erro acima.

Uma possível maneira de resolver esse problema é:

> $somatorio := x\text{-}> \ local\ i;\ sum(x\hat{}i, i = 0..5)$;

$$somatorio := x\text{-}> \ local\ i;\ \sum_{i=0}^{5} n^i$$

O comando *local i* faz com que a variável i seja *local* da função *somatório* e nesse caso ele não tem nenhuma relação com a variável *global i* de mesmo nome.

Note que a expressão colocada após a seta não é avaliada, diferente do comando *unapply( )*. Segue um exemplo que não funciona como esperaríamos:

> $formula := x\hat{}2 - y\hat{}2$;

$$formula := x^2 - y^2$$

> $h := (x, y) \text{->} \ formula$;

$$h := (x, y) \text{->} \ formula$$

Podemos ver que $formula$ não foi avaliada de forma que não criamos a função $h(x, y) = x^2 - y^2$. O que fizemos funciona em parte:

> $h(x, y)$;

$$x^2 - y^2$$

Porém

> $h(a, b)$;

$$x^2 - y^2$$

Se modificarmos o valor de $formula$, passaremos a ter um novo resultado com a função h.

Este último exemplo é um caso típico onde devemos usar o comando *unapply*( ) no lugar do operador seta. Toda vez que em que for necessário a avaliação da expressão ou das variáveis devemos usar o comando *unapply*( ). Caso contrário, se vamos usar algum comando dentro da função no qual temos que inibir a sua atuação, é mais indicado usar o operador –> pois nesse último nem a expressão nem as variáveis são avaliadas.

1.5 Select

Vamos supor que temos um conjunto de números naturais e queremos selecionar os números primos desse conjunto. Por exemplo:

> $S1 := \{3, 5, 6, 7, 8, 11, 15\}$;

$$S1 := \{3, 5, 6, 7, 8, 11, 15\}$$

> $select(isprime, \ S1)$;

$$\{3, 5, 7, 11\}$$

A função $isprime(n)$ retorna $true$ se n for primo e $false$ caso contrário. O comando *select*( ) aplicou, nesse caso, a função *isprime*( ) a cada elemento do conjunto $S1$ e reteve os números que retornam $true$, descartando os números que retornam $false$. A saída tem a mesma forma da expressão de entrada. Aqui a entrada é um conjunto, consequentemente a saída também é.

A função *isprime*( ) é um exemplo de uma função booleana. As funções booleanas são funções que podem ter os mais diversos tipos de entrada porém a saída sempre é $true$ ou $false$.

Vamos definir a função booleana *F(n)* que retorna $true$ se a variável n for maior que 30 e $false$ caso contrário:

$> F := n -> if\ n > 30\ then\ true\ else\ false\ fi;$

$$F := n -> if\ n > 30\ then\ true\ else\ false\ fi$$

$> F(31);$

$$true$$

$> F(29);$

$$false$$

Outra forma é equivalente de definir essa função é através de comando *evalb*( ), que significa *evaluate boolean expression*

$> G := x -> evalb(x > 30);$

$$G := x -> evalb(x > 30)$$

Assim, se temos um conjunto de números por exemplo:

$> S2 := \{n!\$n = 1..10\};$

$$\{1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800\}$$

Para selecionar os números maiores que 30:

$> select(G, S2);$

$$\{120, 720, 5040, 40320, 362880, 3628800\}$$

A sintaxe do comando *select*( ) é

select(função booleana, expressão, argumentos extras da função booleana)

No caso da função booleana ter mais de uma variável, as variáveis extras são repassadas depois da expressão, similar ao comando *map*( ). O comando *select*( ), tal como o *map*( ), retorna uma expressão do mesmo tipo da expressão de entrada.

Como último exemplo, suponha que temos uma expressão algébrica do tipo '+' que envolve números ou raízes de números, por exemplo:

$> expr := sqrt(2) + sqrt(3) + 5 + sqrt(7) + 11\hat{}(1/3);$

$$expr := 2^{1/2} + 3^{1/2} + 5 + 7^{1/2} + 11^{1/3}$$

Queremos selecionar as raízes quadradas de forma que a saída continue sendo do tipo '+':

$> select(type, expr, sqrt);$

$$2^{1/2} + 3^{1/2} + 7^{1/2}$$

1.6 Funções Encaixadas

Quando se têm funções dentro de funções, atenção especial deve ser dada às variáveis comuns às funções de diferentes níveis.

Vamos construir a função *maior* para exemplificar esse fato. *Ela* seleciona os números de um conjunto que são maiores que um dado número. Por exemplo:

$> \ maior(3, \{-2, 7, 8, 0, 5, 2\});$

$$\{5, 7, 8\}$$

A primeira tentativa poderia ser:

$> \ maior \ := (n, S) \ -> select \ (x -> x > n, S) :$

Essa função não passa no teste:

$> \ maior(3, \{-2, 7, 8, 0, 5, 2\});$
$Error, \ (in \ unknown) \ cannot \ evaluate \ boolean$

Por que *maior* não funciona?

Aqui temos um exemplo de uma função dentro de outra. Devemos nos perguntar se a variável 'n' que aparece na função interna $x -> x > n$ é o mesmo 'n' argumento da função *maior*. A resposta é não. O argumento n é um parâmetro local do procedimento *maior* e não tem nenhuma relação nem com as variáveis locais n de outros procedimentos, nem com a variável global n. Aqui a variável n da função interna $x -> x > n$ se refere a variável global n , de fato isso explica o seguinte resultado

$> \ n := 3 :$
$> \ maior(100, \{-2, 7, 8, 0, 5, 2\});$

$$\{7, 8, 5\}$$

Agora podemos entender a mensagem de erro do comando anterior. Lá a variável global n não tinha nenhuma atribuição numérica. A função interna $x -> x > n$ tentava comparar números com o *string* n, por isso a mensagem "*cannot evaluate boolean*". A mensagem "$(in \ unknown)$" se refere ao procedimento onde foi detectado o erro. Uma vez que $x -> x > n$ é um procedimento anônimo, na mensagem aparece *unknown*, caso contrário apareceria o nome do procedimento.

Existem várias formas de corrigir o procedimento *maior*. A primeira forma é

$> \ maior \ := (n, S) -> select \ ((x, n) -> x > n, S, n);$

$$maior \ := (n, S) -> select \ \ ((x, n) -> x > n, S, n)$$

Aqui, n é um argumento da função $(x, n) -> x > n$ e ele é repassado através do terceiro argumento do comando *select*( ). A segunda forma é usar o comando *unapply*( ) no lugar do operador seta mais interno, pois o *unapply*( ) avalia seus argumentos antes de criar a função. Assim:

$> \ maior \ := \ (n, S) -> select \ (unapply \ (n < x, x), S);$

$$maior \;:=\; (n, S) \text{-}\!> select\;\; (unapply\;(n < x, x), S)$$

Quando damos o comando $maior(3, \{0, 5, 7, -2\})$, o comando $unapply(n < x, x)$ cria a função $x \text{-}\!> (3 < x)$ com n avaliado, só então que essa função é aplicada em S.

Existe uma terceira forma de corrigir a função maior usando o comando *subs*( ), de forma a introduzir o argumento n dentro da função mais interna.

1.7 Composição de Funções (@ e @@)

Podemos somar, multiplicar e compor duas ou mais funções. Por exemplo, se temos uma equação e queremos subtrair o lado direito do lado esquerdo podemos usar a soma das funções lhs e rhs:

$>\;\; equacao \;:=\; 2 * y * x + x - 1 = 2 * x - 5;$

$$equacao \;:=\; 2\;y\;x + x - 1 = 2\;x - 5$$

$>\;\; (lhs - rhs)(''');$

$$2\;y\;x - x + 4$$

Observe que os seguintes comandos falam por si só:

$>\;\; (f1 * f2)(x, y);$

$$f1(x, y)\;\; f2(x, y)$$

$>\;\; (g1@g2)(x);$

$$g1(g2(x))$$

$> (g3@g3 - g3@@2)(x);$

$$0$$

$>\;\; ((x\text{-}\!> a\hat{}\,x)@@5)(x);$

$$a^{a^{a^{a^x}}}$$

1.8 Exercícios

1) Tente prever o resultado de cada um dos seguintes comandos e verifique o resultado com o do Maple:

> $map(f, \{a, b, c\}, x)$;
> $map(g, a * b * c, d)$;
> $map(\cos, linalg[matrix](2, 2, [0, Pi, Pi/2, 0]))$;
> $map(a, b/c, d, e)$;
> $map(unapply(x\hat{}2 - y\hat{}2, x, y), [1, 2, 3], 3)$;
> $map(\ln, a * (b + c))$;
> $map(f, [[a], [b]])$;
> $map(abs, a\hat{}2 - b\hat{}2)$;
> $map(sqrt, a - 4)$;
> $map(x \text{-}\!\!> x\hat{}2, \{\$1..5\})$;

2) a) Usando a função $rand(\ )$, construa uma lista de 100 números randômicos menores que 100.
b) Elimine as repetições.
c) Selecione os números maiores que 10 menores que 80.
d) Selecione da lista de c) os números que são ou divisíveis por 3 ou por 5.

3) Como se seleciona os números maiores que 3 de um conjunto que admite números reais, por exemplo:

$$\{1,\ Pi,\ E,\ 5/3,\ 10 * gamma\}$$

4) Explique porque os seguintes comandos não funcionam como esperado

> $f := x, y \text{-}\!\!> cos(x + y)$:
> $f(0, 0)$;

$$x(0, 0), cos(x)$$

Essa atribuição a f é sintaticamente correta ou não?

5) a) Resolva a equação recursiva de Fibonacci $f(n) = f(n-1) + f(n-2)$, $f(0) = 0,\ f(1) = 1$.
b) A partir da solução defina uma função de nome *Fibonacci* que retorna os números de Fibonacci na forma já simplificada. Por exemplo:

> $Fibonacci(20)$;

$$6765 \qquad \#\ já\ simplificado$$

6) Faça uma função de nome *reverso* que retorna a lista inversa de uma dada lista. Por exemplo:

> $reverso\ ([a, b, c, d])$;

$$[d, c, b, a]$$

7) Defina uma função de nome *elementos* que retorna o número de ocorrências de um elemento em uma lista. Por exemplo:

$> elementos(b,\ [1, b, a, b, 2, b]);$

$$3$$

$> elementos(5,\ [1, 2, a, 3, b]);$

$$0$$

8) A partir da função *elementos* (ex. 7) crie uma função booleana de nome $Member$ que verifica se um elemento pertence a uma lista ou não. Por exemplo:

$> Member(a,\ [a, b, c]);$

$$true$$

$> Member(a,\ [1, 2, 3]);$

$$false$$

9) A partir da função *elementos*, faça a função *split* que separa os elementos repetidos dos elementos não repetidos de uma lista. Por exemplo:

$> split([i, k, j, i, j, \ell]);$

$$[\{i, j\}, \{k, \ell\}]$$

10) Faça uma função de nome *remove* tal que $remove(x, L)$ elimina a primeira ocorrência de x na lista L. Caso x não pertença a L, a função deve retornar *FAIL*. Por exemplo:

$> remove(b, [a, b, a, b, c, c]);$

$$[a, a, b, c, c]$$

$> remove(2, [a, b, c]);$

$$FAIL$$

(Monagan).

11) Defina a função *sem_repetição* tal que dado um conjunto de listas, essa função seleciona as listas que não tem elementos repetidos. Por exemplo:

$> sem_repeti$ção$(\{[a, b, c], [a, a], [1, 2, 3, 4], [a, 2, 2]\});$

$$\{[a, b, c], [1, 2, 3, 4]\}$$

12) A função *soma_coeff* tem uma expressão algébrica e uma variável como argumentos. Ela retorna a soma do coeficiente de maior grau na variável com o coeficiente de grau zero na variável. Por exemplo:

$> soma\_coeff((a + b * x)\hat{}3 + 4, x);$

$$a^3 + b^3 + 4$$

Defina essa função de forma que a expansão da expressão algébrica seja feita apenas uma vez.

Nos próximos exercícios 13), 14) e 15), faça duas versões de cada função. A primeira usando o comando *unapply*( ) e a segunda usando o operador seta.

13) SL é uma função que tem 2 argumentos que podem ser listas ou conjuntos de números. Ela retorna a soma de todos os elementos dos argumentos. Por exemplo:

$> SL([4, 3/2, -5], \{1/2, 0, -Pi, 1\});$

$$2 - Pi$$

14) Suponha que temos a matriz

$$A = \begin{bmatrix} x + y & \sin(x * y) \\ x^2 y & ax + by \end{bmatrix}$$

já definida previamente. Usando a matriz A defina a função $A\_fun(x, y)$ tal que, por exemplo:

$> A\_fun(0, 2);$

$$\begin{bmatrix} 2 & 0 \\ 0 & 2\ b \end{bmatrix}$$

e

$> D[1](A\_fun)(0, 2);$

$$\begin{bmatrix} 1 & 2 \\ 0 & a \end{bmatrix}$$

15) Seja $\vec{v}$ e $\vec{w}$ vetores. Defina a função

$$T(\vec{v}, \vec{w}) = \vec{v} \ - \ 2(\vec{v}.\vec{w})\vec{w}$$

Por exemplo:

```
> v := linalg[vector]([1/a, 1]) :
> T(v, v);
```

$$\left[-\frac{a^2+2}{a^3}, -\frac{a^2+2}{a^2}\right] \quad \#\ já\ simplificado$$

16) Faça uma função que encontra a norma euclideana de um polinômio de uma variável com coeficientes numéricos. A norma euclideana de $P(x) = \sum_{i=0}^{n} a_1 x^i$ é dada por $\sqrt{\sum_{i=0}^{n} a_i^2}$. Por exemplo:

```
> NormaEuclideana(2 * x^2 + 3 * x + 1);
```

$$14^{1/2}$$

2 Programação através de Procedimentos

2.1 Forma Geral dos Procedimentos

Vimos que podemos construir funções matemáticas tanto com o comando *unapply*( ) quanto com o operador seta. Uma terceira forma possível é usar o comando *proc*( ), que serve para construir procedimentos (subrotinas).

Por exemplo, a função $f(x,y) = x^2 - y^2$ é construída da seguinte forma:

$> \ f := proc(x,y)\ x\hat{}2 - y\hat{}2\ end;$

$$f := \ proc(x,y)\ x\hat{}2 - y\hat{}2\ end$$

Esse é um exemplo de um procedimento com apenas um comando. A forma geral dos procedimentos é:

proc(sequência de argumentos)
local sequência de nomes ;
global sequência de nomes ;[2]
options opções ;
comando 1 ;
comando 2 ;
⋮
comando N
end;

O comando *local* declara as variáveis locais do procedimento. Essas variáveis não tem nenhuma relação com as variáveis do mesmo nome usadas fora do procedimento ou dentro de qualquer outro procedimento.

O comando global declara as variáveis globais. Esse comando só está presente na versão 5.3 do maple. Nas versões anteriores, todas as variáveis que não são declaradas locais, são consideradas globais.

Vamos ver um exemplo do comando *options*. Note que o procedimento f definido acima não é mostrado como um operador $->$, apesar de ser uma função matemática. Veja agora essa nova construção.

$> \ f1 \ := \ proc(x,y)$
$> \ options\ operator,\ arrow;$
$> \ x\hat{}2 - y\hat{}2$
$> \ end;$

$$f1 \ := \ (x,y) \ -> \ x^2 - y^2$$

Podemos ver que as funções construídas com o comando *unapply*( ) ou com o operador seta são, na verdade, procedimentos que tem as opções *operator* e *arrow*.

No caso de funções temos apenas um comando. Nos procedimentos podemos ter tantos comandos quando queiramos. Qual é então o valor que o procedimento retorna? Em geral,

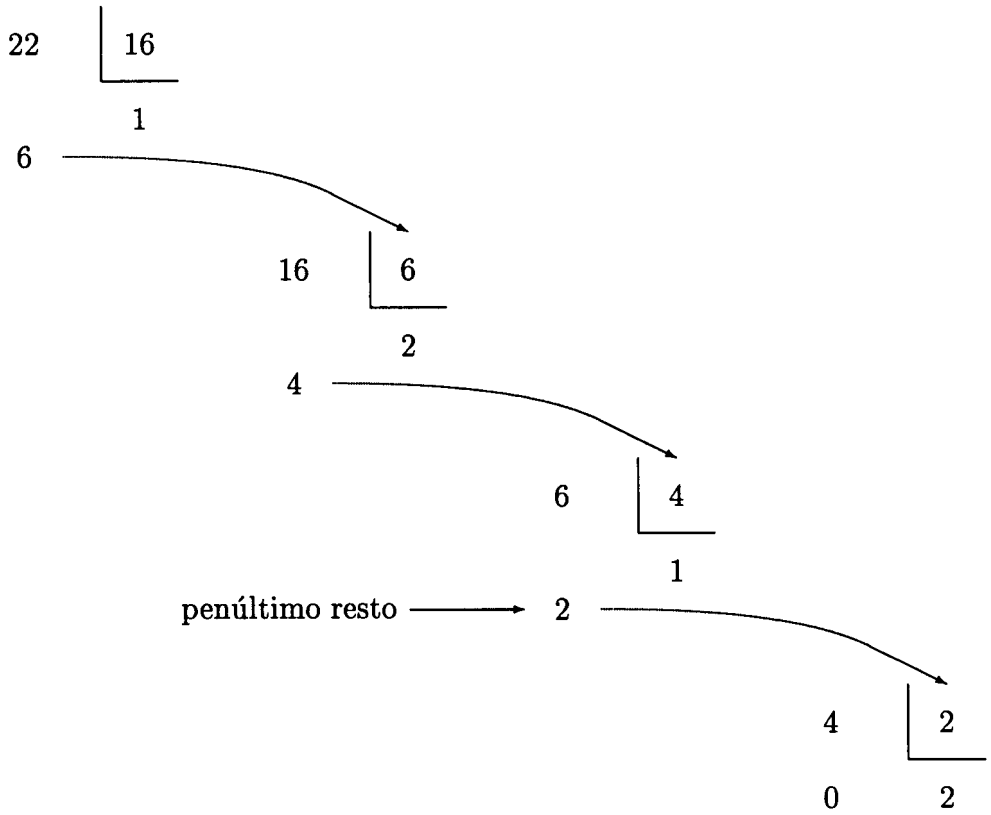
[2] Na versão 5.2 e anteriores, não existe o comando *global*.

o valor retornado é o valor do último comando. Existem outras formas de retornar um valor, por exemplo, através do comando *RETURN*, que pode estar em qualquer ponto da rotina.

Vamos ver agora um exemplo de um procedimento mais elaborado.

2.2 O Maior Divisor Comum

Se temos dois números inteiros a e b, como encontrar o maior número que divide a e b? Em 500 A.C., Euclides já tinha desenvolvido um algoritmo para resolver esse problema. O método consiste em dividir inicialmente o número maior pelo menor. Se há resto, dividimos o menor pelo resto. Continuamos a dividir o resto anterior pelo novo resto até que o último resto seja zero. O *MDC* é o penúltimo resto. Esse método pode ser visualizado no seguinte exemplo com os números 22 e 16 na figura abaixo.



Queremos agora construir um procedimento que acha o *MDC* entre dois inteiros. Vamos precisar da função *irem* (*integer remaider*) do Maple que encontra o resto da divisão de dois números inteiros. Faremos um *loop* para ir dividindo os sucessivos restos. Porém, primeiro temos que iniciar as variáveis que vão representar aos restos ($r1$ e $r2$). O procedimento fica da seguinte forma:

$MDC := proc\,(a, b)$
$local\ r1, r2, resto;$
$r1 := a;$

$r2 := b;$
$while\ r2 > 0\ do\ resto :=\ irem(r1, r2);$
$\qquad\qquad\qquad\qquad r1\ :=\ r2;$
$\qquad\qquad\qquad\qquad r2\ :=\ tmp$
$od;$
$r1$
$end :$

Escolhemos o *loop while do od* primeiro porque não sabemos de antemão o número de voltas do *loop* e segundo porque não precisamos de um contador de voltas. O *loop* termina quando o último resto for zero e o resultado deve ser o penúltimo resto. Por isso o último comando do procedimento é $r1$. Assim:

$>\ MDC\ (22, 16);$

$$2$$

O valor das variáveis globais $r1$, $r2$ e $resto$ não foram alterados como pode ser visto no seguinte comando:

$>\ r1,\ r2,\ resto\ ;$

$$r1,\ r2,\ resto$$

As variáveis globais a e b também não se alteraram, apesar dos mesmos nomes a e b terem sido usados como argumentos do procedimento *MDC*. Note que não há nenhuma atribuição aos argumentos a e b.

2.3 Sequências

Vamos supor que queremos construir uma sequência que segue uma regra de formação dada por uma certa função, que vamos chamar de *termo*. Assim, o i-ésimo termo é dado por $termo(i)$. Para isso usamos o comando *seq (sequence constructor)*:

$$seq\ (termo(i), i = a..b)$$

onde a e b são números inteiros ou fracionários. Se a for maior que b, o resultado é $NULL$. Aqui, o índice i cresce de 1 em 1, de forma que o resultado desse comando é:

$$termo(a),\ termo(a+1),\ termo(a+2), \cdots\ termo(a+n)$$

A sequência para quando $a + n + 1$ for maior que b. Se quisermos que i cresça de k em k desde a até b, temos que redefinir o índice da seguinte forma:

$$seq(termo\ (a + i * k), i = 0..(b - a)/k)$$

A função $termo(\ )$ pode ser uma função bastante complexa, definida através de programação funcional ou através de procedimentos.

Uma sequência também pode ser gerada através de um *loop for do od*, da seguinte forma:

$seq1 := NULL :$
$for\ i\ from\ a\ to\ b\ do$
$\ \ seq1 := seq1,\ termo(i)$
$od :$

O comando $seq1 := NULL$ é um comando de iniciação da variável $seq1$. A iniciação sempre é necessária, caso contrário a atribuição $seq1 := seq1,\ termo(i)$ induziria a uma recursão infinita. A primeira vez que essa atribuição é feita, $seq1$ já tem que ter um valor definido que aqui é $NULL$.

Existe uma diferença essencial entre esses dois métodos de se criar sequências no que se refere a eficiência. O primeiro método é mais eficiente. O segundo método é mais lento porque a cada volta do *loop* é feita uma atribuição que consome tempo de computação. Veja os resultados do seguinte loop:

$> seq1 := NULL ;$

$$seq1 :=$$

$> \ for\ i\ to\ 5\ do$
$> \qquad seq1 = seq1\ , a\hat{}i$
$> \ od;$

$$seq1 := a$$
$$seq1 := a, a^2$$
$$seq1 := a, a^2, a^3$$
$$seq1 := a, a^2, a^3, a^4$$
$$seq1 := a, a^2, a^3, a^4, a^5$$

Observe que foram feitas 4 atribuições intermediárias antes que o resultado $seq1 := a, a^2, a^3, a^4, a^5$ fosse dado.

Sempre que possível, deve-se evitar a construção de sequências através de *loops*, e no lugar usar o comando $seq(\ )$, junto com técnicas de programação funcional ou por procedimentos para se gerar a função $termo(\ )$.

A eficiência conta a favor do comando $seq(\ )$, porém o *loop* $for\ do\ od$ tem mais possibilidades de construção de sequências porque podemos colocar tantos comandos quanto quisermos dentro do *loop*.

Existe uma outra forma de usar o comando seq, onde a variável que faz o papel de contador na versão $seq(termo(i),\ i = a..b)$ não assume valores numéricos, mais sim operandos de uma expressão. A notação é:

$$seq(f(i), i = L)$$

onde L pode ser uma lista, um conjunto, uma expressão algébrica, etc. O resultado desse comando é:

$$f(op(1, L)),\ f(op(2, L)), \cdots,\ f(op(\text{nops}(L), L))$$

Assim, f é aplicado nos operandos de L. Veja os seguintes exemplos:

> $seq(diff(i,x), i = [sen(x), x^3, \ln(x), g(x)])$;

$$cos(x),\ 3x^2,\ 1/x,\ \frac{\partial g(x)}{\partial x}$$

> $seq(f(i), i = a + b + c)$;

$$f(a),\ f(b),\ f(c)$$

Observe a similaridade dessa forma do comando *seq*( ) com o comando *map*( ). A diferença entre eles é que enquanto o comando *map*( ) gera uma saída do mesmo tipo da entrada, o comando $seq(f(i), i = L)$ sempre gera uma sequência, independente do tipo da expressão L. Em geral essa forma do comando *seq*( ) substitui o comando $op(map(f, L))$.

2.4 O Procedimento Split

Vamos supor que temos uma lista e queremos separar os elementos repetidos dos elementos não repetidos. Por exemplo,

> $split([a, b, b, c, d, c])$;

$$[\{b, c\}, \{a, d\}]$$

Para construir o procedimento *split*, vamos fazer um *loop* que corre sobre todos os elementos da lista de entrada. Para cada elemento será feito um teste para verificar se ele é único ou não. Caso o elemento seja único, ele será colocado no conjunto dos elementos não repetidos, caso contrário ele será colocado no conjunto dos repetidos. O procedimento fica então da seguinte forma:

```
split1 := proc(L : list) local S1, S2, i;
 S1 := {};
 S2 := {};            # iniciação de S1 e S2
for i to nops(L) do
    if member (L[i], subsop (i = NULL, L))
    then  S1 := {op(S1), L[i]}
    else  S2 := {op(S2), L[i]}
    fi
od;
[S1, S2] end;
```

Esse procedimento tem um algoritmo bastante simples porém não é eficiente quando se tem muitos elementos repetidos.

Vamos ver o tempo de CPU do seguinte comando para posteriormente comparar o tempo gasto por uma segunda versão do procedimento *split*:

> $time(\ ):$
> $split1([i\$500, j])$;

$$[\{i\}, \{j\}]$$

```
> time( ) - " ";
```

1.416 # segundos

Observe que na lista de entrada o elemento i aparece 500 vezes. O procedimento *split*1 faz 500 testes verificando se i é repetido, sendo que bastava testar uma única vez. Vamos fazer outra versão do *split* eliminando da lista de entrada os elementos iguais ao que já foram testados. Vamos criar uma segunda lista que inicialmente será igual a lista de entrada e que vai diminuindo até se tornar a lista nula a medida que os testes forem sendo feitos. Uma vez que não sabemos previamente quantas voltas o *loop* terá, não vamos usar o comando *for do od*, mas sim *while do od*:

```
split2 := proc (L1 : list)
local L2, L3, S1, S2, elem1;
L2 := L1;
S1 := NULL;
S2 := NULL;
while L1 <> [ ] do
  elem1 := L2[1];
  L3 := subsop (1 = NULL, L2);
  if member (elem1, L3)
  then   S1 := S1, elem1;
         L2 := subs(elem1 = NULL, L3)
 else    S2 := S2, elem1;
       L2 := L3;
  fi
od;
[{S1}, {S2}]
end :
```

Observe que seja qual for o resultado do teste $member(elem1, L2)$ a lista $L2$ vai diminuir de tamanho. Isto garante que o *loop* terá um fim. Vamos comparar a eficiência dessa última versão com a primeira:

```
> time( ) :
> split2([i$500, j]);
```

$$[\{i\}, \{j\}]$$

```
> time( ) - " ";
```

0.05

Para esse exemplo particular, *split*2 foi 30 vezes mais rápido do que *split*1. Vamos comparar agora a eficiência desses algoritmos para listas com muitos elementos diferentes. Observe que não vamos pedir para o Maple mostrar os resultados, uma vez que são muito grandes.

$> readlib(showtime)(\ );$
$O1\ :=\ split1([\prime a.i\prime \$i = 1..500]) :$
$time\ 2.65\ words\ 767549$
$O2\ :=\ split2([\prime a.i\prime \$i = 1..500]) :$
$time\ 1.15\ words\ 386146$

Novamente o procedimento *split*2 é mais eficiente. Nesse caso o motivo está relacionado com o espaço de memória utilizado como pode ser visto na área de *words*. No procedimento *split*2, o tamanho da lista se reduz a cada volta do *loop*, enquanto que no *split*1 o tamanho da lista se mantém inalterado. A eficiência será cada vez mais diferente a medida que o tamanho da lista de entrada aumentar.

Podemos fazer uma versão mais eficiente ainda para o procedimento *split* especialmente no caso de listas com muitos elementos diferentes. Note que o procedimento *split*2 gera uma quantidade grande de atribuições no *loop while do od*. Vimos que a construção de uma sequência através do comando *seq*( ) é mais eficiente do que através de um *loop for do od*, pois uma série de atribuições é evitada. Nesse mesmo espírito podemos programar a seguinte versão:

$split3\ :=\ proc(L)$
$local\ S1, S2, f;$
$S1\ :=\ \{op(L)\};$
$f\ :=\ proc(x, L)\ local\ pos;\ member(x, L, pos);\ pos{=}NULL\ end;$
$S2\ :=\ \{op\ (subsop\ (seq\ (f(i, L), i = S1), L))\};$
$[S2, S1\ minus\ S2]$
$end :$

Vamos comparar a eficiência de *split*2 e *split*3:

$> readlib(showtime)(\ );$
$O1\ :=\ split2([\prime a.i\prime \$i = 1..500]) :$
$time\ 1.15\ words\ 395357$
$O2\ :=\ split3([\prime a.i\prime \$i = 1..500]) :$
$time\ 0.21\ words\ 23654$

Esses números falam por si só.

Observe que na versão *split*3, temos um procedimento atribuído a uma variável local. De fato, podemos ter qualquer estrutura de dados atribuída a uma variável local.

O procedimento f não pode ter apenas x como argumento porque precisamos repassar a lista L, que é argumento de *split*3, para dentro de f. Isso sempre é necessário quando temos procedimentos encaixados. Os parâmetros e as variáveis locais de um procedimento não tem relação nem com as variáveis locais do procedimento interno nem com as variáveis globais. Se não tivéssemos repassado L, o comando $member(x, L, pos)$ iria se referir a variável global L nas versões 5.2 e anteriores e a variável local L de f na versão 5.3.

2.5 Programação com Polinômios

Vamos retornar de novo à programação funcional do capítulo anterior para analisar com mais detalhes a função que calcula a norma euclideana de polinômios.

A norma euclideana de um polinômio de uma variável

$$p = \sum_{i=0}^{n} a_i x^i$$

é dada por

$$||p|| = \sqrt{\sum_{i=0}^{n} a_i^2}$$

Para fazer um procedimento que calcula a norma de p, vamos extrair os coeficientes a_i usando o comando *coeff*(p, x, i). No entanto, o comando *coeff* só aceita a expressão p na forma expandida ou com as variáveis x^i fatoradas. Por exemplo, veja o que ocorre aqui:

$> \ (x+1)\hat{}2 + x \ ;$
$> \ coeff(p, x, 2);$
Error, unable to compute coeff
$> \ collect(p, x);$

$$x^2 + 3x + 1$$

$> \ coeff('', x, 2);$

$$1$$

Podemos escrever o procedimento da seguinte forma

$> \ NormaEuclideana \ := \ (p, x) \ ->$
$sqrt(sum('coeff(expand(p), x, i)', \ 'i' = 0..degree(p, x));$
$NormaEuclideana \ := \ (p, x) -> sqrt\left(\sum_{'i'=0}^{degree(p,x)} \ 'coeff(expand(p), x, i)^{2'}\right)$

Usamos o comando $degree(p, x)$ que dá o grau do polinômio p na avariável x. Observe que usamos as aspas direitas no comando $'coeff(expand(p), x, i)'$ e no índice de soma $'i'$. Isso é necessário porque o comando $sum(\ )$ avalia o argumento e o índice de soma antes de executar o somatório. Assim, se i já tivesse um valor numérico, o procedimento sem as aspas não funcionaria corretamente.

A norma de $ax^2 + bx + c$ pode ser calculada da forma:

$> \ NormaEuclideana(a * x\hat{}2 + b * x + c, x);$

$$(c^2 + b^2 + a^2)^{1/2}$$

Esse procedimento pode ser melhorado, pois ele calcula cada coeficiente do polinômio separadamente. O comando *coeff*(p, x, i) é executado tantas vezes quanto for o grau do polinômio. Tente por exemplo, calcular a norma do polinômio $normal((x\hat{}900-1)/(x+1))$. Por outro lado, existe no Maple o comando *coeffs* que dá diretamente todos os coeficientes de um polinômio na forma de uma sequência. Por exemplo:

$> \ coeffs(x\hat{}2 + 3 * x + 2, x);$

$$2,\ 3,\ 1$$

Uma vez tendo os coeficientes numa sequência, queremos então elevar cada termo da sequência ao quadrado, o que pode ser feito com o seguinte comando

$$map(x\text{-}\!>\ x\hat{}2, [coeffs(expand(p), x)])$$

Note que precisamos colocar a sequência numa lista. O último passo é converter essa lista numa expressão tipo ' + '. O procedimento fica da seguinte forma:

$$NormaEuclidiana1\ :=\ (p, x)\text{-}\!>$$

$$sqrt(convert\ (map(x\text{-}\!>\ x\hat{}2, [coeffs(expand(p), x)]), `i`)\ :$$

Assim,

$>\ time(\ ):$
$> NormaEuclidiana1(normal((x\hat{}900 - 1)/(x + 1)), x);$

$$30$$

$> time(\ ) - \ ''\ '';$

$$0.133 \qquad \#(segundos)$$

Vamos comparar com o tempo que leva a primeira versão:

$>\ time():$
$> NormaEuclidiana(normal((x\hat{}900 - 1)/(x + 1)), x);$

$$30$$

$> time() - \ ''\ '';$

$$21.967$$

2.6 Exercícios

1) a) A função *irem* usada no procedimento *MDC* não é simétrica nos seus argumentos. Explique porque a função *MDC* é simétrica, isto é $MDC(a, b) = MDC(b, a)$.
b) Verifique que o procedimento *MDC* não funciona para números inteiros negativos.
c) Como se altera o procedimento para que ele funcione na situação descrita em b).

2) Como se define um *loop* usando o comando *for do od* que executa a mesma tarefa que os seguintes comandos:
a) seq1:=seq(termo(a+i*n), i=0..(b-a)/n);
b) sum1:=sum('termo(i)', 'i'=a..b);
c) prod1:=prod('termo(i)', 'i'=a..b);
d) list1:=[seq(termo(i), i=a..b)];
e) set1:={seq(termo(i), i=a..b)};
f) seq2:=seq(f(i),i=L);

3) De um exemplo tal que comando $seq(f(i), i = L)$ fornece um resultado diferente do comando $op(map(f, L))$, onde f é uma função e L é uma estrutura de dados ou uma expressão algébrica.

4) Quais são os comandos para se gerar os seguintes resultados:
a) $\{\{0\},\ \{\{0\}\},\ \{\{\{0\}\}\}, \cdots\}$ com 10 elementos.
b) [tan(x), tan(tan(x)), tan(tan(tan(x))), $\cdots$] com 10 elementos.
c)$[0, [0, [0, \cdots, [0]]]]]]]]]$

5) Modifique a versão 3 do procedimento *split* de forma que os elementos não repetidos fiquem numa lista com a mesma ordem da lista original. Por exemplo:

$> split([i, z, i, w, a]);$

$$[\{i\}, [z, w, a]]$$

6) Escreva um procedimento que calcula a norma de Frobenius de uma matriz A $m \times n$. A norma de Frobenius é dada por

$$\sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |A_{ij}|^2}$$

(Monagan).

7) a) Faça um procedimento que retorna o maior número entre um conjunto de números. Por exemplo:

$> maximum(\{2, -5, 10, 0\});$

$$10$$

b) Teste seu procedimento para o conjunto $\{2,\ Pi,\ 0\}$. Caso o resultado não seja Pi, modifique o procedimento de forma a aceitar quaisquer números reais $(E, Pi, gamma, ...)$.

8) O polinômio $a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$ de uma variável pode ser escrito como uma lista da seguinte maneira

$$[a_n,\ a_{n-1}, \cdots,\ a_1,\ a_0]$$

a) Escreva um procedimento que soma 2 polinômios na forma de listas. Por exemplo

$> addpoly([2, 3, 0], [7, -5, 1, 2]);$

$$[7, -3, 4, 2]$$

b) Escreva um procedimento que multiplica 2 polinômios na forma de listas. Por exemplo:

$> mulpoly([1/2, -2, 1], [-1, 1]);$

$$[-1/2, 5/2, -3, 1]$$

9) Faça um procedimento booleano de nome *ispermute* que tem 2 listas e o nome *odd* ou *even* como argumentos. O procedimento retorna *true* ou $false$ caso essas listas seja permutações ímpares ou pares entre si. Por exemplo:

$>$ $ispermute([a, b, c, d], [d, a, b, c], even)$;

$$false$$

$>$ $ispermute([1, 2, 3], [1, 3, 2], odd)$;

$$true$$

10) Faça os procedimentos que dão as permutações pares e ímpares de uma dada lista. Por exemplo

$>$ $odd\_permute([a, b, c])$;

$$[[a, c, b], [b, a, c], [c, b, a]]$$

$>$ $even\_permute([a, b, c])$;

$$[[a, b, c], [b, c, a], [c, a, b]]$$

11) Estenda ao procedimento *NormaEuclidiana* para polinômios de várias variáveis. A norma euclidiana do polinômio

$$p(x) = \sum_i \sum_j \cdots \sum_k a_{ij\cdots k} x^i y^j \cdots z^k$$

nas variáveis $x, y, \cdots, z$ é dada por

$$||p(x)|| = \sqrt{\sum_i \sum_j \cdots \sum_k a^2_{ij-k}}$$

Faça de forma que, se o segundo argumento do procedimento for uma variável, ele retorna a norma do polinômio em uma variável, e se o segundo argumento for uma lista ou um conjunto de variáveis, o procedimento retorna a norma com relação a essas variáveis. Por exemplo:

$>$ $NormaEuclidiana2(a * x * y + b * x + c * y + 1, [x, y])$;

$$(a^2 + b^2 + c^2 + 1)^{1/2}$$

$>$ $NormaEuclideana(a * x * y + b * x + c * y + 1, x)$;

$$((a\ y + b)^2 + (c\ y + 1)^2)^{1/2}$$

3 Programação Recursiva

3.1 A Função Fatorial

Um procedimento é recursivo quando ele chama a si mesmo durante a execução de um algoritmo. Quando a definição de uma função está numa forma recursiva, em geral é mais simples construir um procedimento recursivo para essa função do que um procedimento não recursivo. Por exemplo, a função $fatorial$ de um número inteiro não negativo pode ser definida da seguinte forma recursiva:

$$fatorial(n) \;=\; n\; fatorial(n-1)$$
$$fatorial(0) \;=\; 1$$

Observe que no lado direito da primeira equação aparece a própria função $fatorial$ aplicada em $n-1$. A medida que a função chama a si mesma, o argumento vai decrescendo. A segunda equação é uma equação de finalização que interrompe o ciclo de autoreferência.

Um procedimento para essa função poderia ser:

$$fatorial \;:=\; proc(n)$$
$$if\; n = 0\; then\; 1\; else\; n * fatorial(n-1)\; fi$$
$$end$$

Assim:

$$>\; fatorial(10);$$

3628800

As funções recursivas são fáceis de serem implementadas e o uso da recursão em programação é extremamente útil. A condição de finalização é essencial para que a recursão não leve a um *loop* infinito. Ela deve ser analisada com detalhes de forma a cobrir todas as possibilidades. Por exemplo, se usarmos o procedimento $fatorial$ definido da forma acima, dando uma variável não numérica como argumento cairemos numa recursão infinita:

$$>\; fatorial(z);$$
$$/user/local/bin:\; 27666\; Memory\; fault$$

Como z não é um número, a condição de finalização nunca é satisfeita. Para resolver esse problema, podemos modificar o procedimento $fatorial$ acrescentando o teste dinâmico no argumento da forma:

$$fatorial \;:=\; proc(n : nonnegrint)$$
$$if\; n = 0\; then\; 1\; else\; n * fatorial(n-1)\; fi$$
$$end:$$

Assim:

$$>\; fatorial(z);$$
$$Error,\; fatorial\; expects\; its\; 1st\; argument,\; n,\; to\; be\; of\; type\; nonnegint,$$
$$but\; received\; z.$$

Dessa forma, o procedimento só executa o algoritmo recursivo para argumentos inteiros não negativos, caso contrário retorna uma mensagem de erro. Por outro lado, é mais interessante que o procedimento retorne a mesma entrada quando o argumento é uma variável livre, da seguinte forma:

```
> fatorial(z);
```

$$fatorial(z)$$

Assim, o resultado poderia ser usado em cálculos posteriores até que z assumisse um valor numérico. Para implementar essa característica, usamos as aspas direitas que retarda a avaliação:

```
fatorial := proc(n)
    if type(n, name) then 'procname'(n)
    elif n = 0 then 1
    elif type(n, posint) then n * procname(n - 1)
    else ERROR (`the argument must be type nonnegint or type name`)
    fi
    end:
```

Toda vez que a variável *procname* aparece dentro de um procedimento, ela é avaliada no nome do procedimento. Assim, se z for tipo *name*, o valor retornado será $fatorial(z)$. Observe que as aspas direitas retardam a avaliação, o que evita uma recursão inifinita. Se aparecer $fatorial(z)$ numa expressão algébrica, o programa verifica se z é tipo *name*, caso verdadeiro, ele mantém $fatorial(z)$ inalterado. Por exemplo:

```
> 2 * fatorial(a);
```

$$2\ fatorial(a)$$

```
> a := 5;
```

$$a := 5;$$

```
> " ";
```

240

Dessa forma, o procedimento fica dentro da regra geral da computação algébrica que é retornar a própria entrada caso nada possa ser avaliado.

O comando $ERROR(`comentário`)$ serve para interromper a execução do algoritmo e mostrar um comentário de erro. Aqui usamos as aspas esquerdas caso haja caracter não alfanumérico no comentário. Assim:

```
> fatorial(-10);
Error, (in fatorial) argument must be type nonnegint or type name
```

O procedimento $fatorial$ ainda pode ser melhorado para evitar testes desnecessários, de forma a aumentar a eficiência do algoritmo. Veremos isso mais adiante.

3.2 Os Números de Fibonacci

Outro exemplo de função recursiva é a função $fibonacci$ definida da seguinte forma:

$$fibonacci(n) \;=\; fibonacci(n-1) \;+\; fibonacci(n-2)$$
$$fibonacci(0) \;=\; 0,$$
$$fibonacci(1) \;=\; 1$$

Observe que a função $fibonacci$ refere-se a si mesma 2 vezes com os argumentos $n-1$ e $n-2$, por isso temos 2 equações de finalização. Seguindo a última versão do procedimento $fatorial$, podemos escrever a seguinte versão para $fibonacci$:

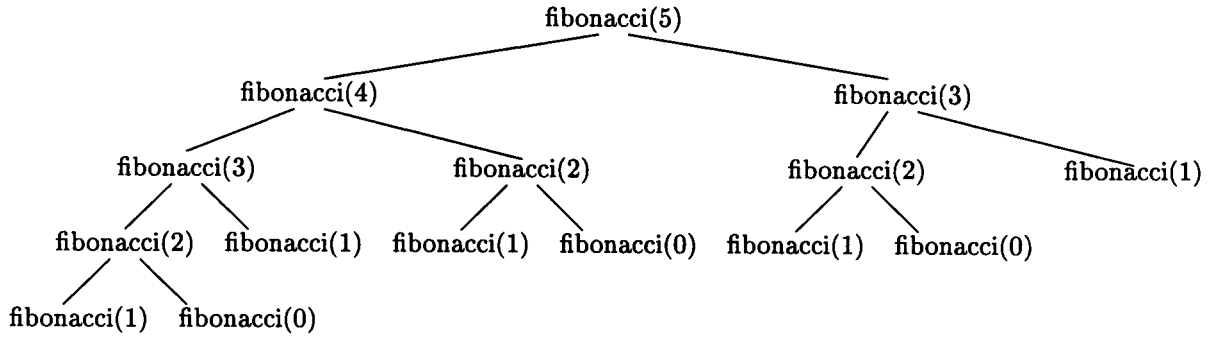
```
fibonacci := proc(n)
    if type (n, name) then `procname`(n)
    elif n = 0 or n = 1 then n
    elif type(n, posint) then procname(n - 1) + procname(n - 2)
    else ERROR(`argument must be type nonnegint or type name`)
    fi
    end
```

Assim:

```
> fibonacci(i) $ i = 0..10;
```

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55$$

Essa versão do procedimento $fibonacci$ tem um sério problema que vem do fato da função se referir a si mesma 2 vezes. Esse procedimento não consegue calcular, por exemplo, $fibonacci(50)$. Para visualizar o problema, veja na figura abaixo como é feito o cálculo de $fibonacci(5)$.



Podemos ver que $fibonacci(3)$ foi calculado duas vezes e $fibonacci(2)$ três vezes. No total, será necessário usar a função 15 vezes. Não é difícil mostrar que o número de vezes de chamada da própria função no cálculo de $fibonacci(n)$ é

$$\sum_{i=1}^{n} fibonacci(i) + fibonacci(n-1)$$

Para $n = 50$ temos que a função $fibonacci$ é chamada 40.730.022.147 vezes. Na estação Sparc 10 onde os cálculos dessa apostila estão sendo feito, o tempo para calcular

$fibonacci(10)$ é 0.466 segundos. Usando a relação acima, temos que para o número 10, a função é chamada 177 vezes. Daí podemos estimar o tempo médio de cada chamada como sendo 0.0026 segundos. Para calcular $fibonacci(50)$ a estimativa é que a estação levaria 0.0026∗40.730.022.147 segundos, ou seja, mais de 3 anos para achar o resultado. Uma maneira de resolver esse problema é, uma vez calculado $fibonacci$ de um número, armazenar o resultado em uma tabela de forma que a próxima referência a $fibonacci$ do mesmo número, não obrigasse o sistema a recalcular, usando o algoritmo, mas sim se referir ao resultado guardado na tabela. O sistema deverá então, antes de usar o algoritmo, verificar se o número pedido está na tabela ou não.

Esse tipo de armazenamento é feito automaticamente se colocarmos a opção $remember$ no procedimento. Nesse caso o cálculo de $fibonacci(50)$ é imediato:

$> \ readlib(showtime)(\ );$
$O1 \ := \ fibonacci(50)$

$$12586269025$$

$time$ 0.08
$O2 \ := \ off;$
$>$

A $remember\ table$ é o quarto operando do procedimento, assim para ver o que foi guardado nela:

$> \ op\ (4,\ eval(fibonacci));$

$$table([$$
$$15 = 610$$
$$20 = 6765$$
$$\vdots$$
$$10 = 55$$
$$])$$

Ou seja, foram guardados os valores dos 50 primeiros números de $fibonacci$. A $remember\ table$ pode ser apagada com o comando $forget$:

$> \ readlib(forget)(fibonacci);$
$> \ op(4,\ eval(fibonacci));$ $\quad \#\ para\ verificar$

$$table([\ ])$$

Podemos acrescentar a mão novos valores na $remember\ table$

$> \ fibonacci(2) \ := \ 1 : \ fibonacci(3) \ := \ 2 :$
$> \ op(4,\ eval(fibonacci));$

$$table([$$
$$2 = 1$$
$$4 = 3$$
$$])$$

Podemos apagar um único valor da *remember table*

> $forget(fibonacci,\ 4)$;
> $op(4,\ eval(fibonacci))$;

$$table\ ([$$
$$2 = 1$$
$$])$$

Usando os recursos da *remember table* podemos agora programar a função $fatorial$ (seção 1) de forma bem simples, através dos seguintes comandos:

> $fatorial\ :=\ n\ \rightarrow\ n\ *\ fatorial(n-1)$
> $fatorial\ :=\ n\ \rightarrow\ n\ fatorial(n-1)$
> $fatorial(0)\ :=\ 1$:

Assim:

> $fatorial(5)$;

$$120$$

> $op(4,\ eval(fatorial))$;

$$table\ ([$$
$$0 = 1$$
$$])$$

Podemos ver que atribuição $fatorial(0) := 1$ colocou o valor 1 na *remember table* de $fatorial$, de forma a evitar a recessão infinita. É claro que essa forma de definir $fatorial$ tem os problemas de recursão inifinita quanto o argumento não é um inteiro positivo.

3.3 Os Polinômios de Fibonacci

Vamos ver mais um exemplo de procedimentos recursivos onde é necessário o uso da *remenber table*. Os polinômios de Fibonacci são definidos da forma

$$\begin{aligned} F_n(x) &= x\ F_{n-1}(x) + F_{n-2}(x) \\ F_0(x) &= 1 \\ F_1(x) &= x \end{aligned}$$

Vamos inicialmente usar um algoritmo igual ao procedimento anterior dos números de Fibonacci. Assim

```
F := proc(n, x : name)
options remember;
if type(n, name) then 'procname'(x)
elif n = 0 then 1
elif n = 1 then x
elif type(n, posint) then x * F(n − 1, x) + F(n − 2, x)
else ERROR(‘first argument must be type nonnegint or type name‘)
fi
end
```

Assim

$> F(5, x)$:

$$x(x(x(x^2+1)+x)+x^2+1)+x(x^2+1)+x$$

$> expand(")$:

$$x^5+4x^3+3x$$

Da forma como F foi construído, o usuário tem que dar o comando *expand* toda vez que usar F, caso queira os polinômios na forma usual. Um problema mais sério ocorre na *remember table*, pois:

$> op(4, eval(F))$;

$$table([$$

$$\begin{aligned}
(1,x) &= x \\
(4,x) &= x(x(x^2+1)+x)+x^2+1 \\
(3,x) &= x(x(x(x^2+1)+x)+x^2+1)+(x^2+1)+x \\
(3,x) &= x(x^2+1)+x \\
(0,x) &= 1 \\
(2,x) &= x^2+1
\end{aligned}$$

$$])$$

Os polinômios são guardados na forma não expandida que nesse caso significa mais uso de memória e maior lentidão no processo recursivo do cálculo.

Outro fator que provoca lentidão são os testes desnecessários do tipo $type(n,\ posint)$, pois uma vez que a recursão é iniciada, com um comando do tipo $F(50, x)$ por exemplo, todas as outras chamadas do procedimento serão com números inteiros positivos até $F(0, x)$. Assim, não é necessário testar se eles são positivos. Para corrigir esse problema é necessário fazer dois procedimentos, o primeiro faz os testes para saber se a entrada é tipo *name* ou *nonnegint*. Se for tipo *nonnegint*, o procedimento chama um segundo procedimento que fará a recursão, sem testar a positividade do argumento. Para corrigir esses 2 pontos que foram levantados, temos os seguintes procedimentos:

```
F := proc(n, x : name)
if type(n, name) then 'procname'(n, x)
elif type(n, nonnegint) then `F/recursive`(n, x)
elif ERROR(`first argument must be type name or type nonnegint`)
fi
end

`F/recursive` := proc(n, x)
options remember, system ;
if n = 0 then 1
elif n = 1 then x
else expand(x * procname(n − 1, x) + procname(n − 2, x))
fi
end
```

Agora temos:

```
> F(5, y)
```

$$y^5 + 4y^3 + 3y$$

```
> op(4, eval(F));
>
```

O Maple não retorna nada no último comando pois o *option remember* foi colocado no procedimento `F/recursive`. De fato

```
> op(4, eval(`F/recursive`));
```

$$\begin{array}{c} table\ ([\\ (0, y) = 1 \\ (3, y) = y^3 + 2y \\ (5, y) = y^5 + 4y^3 + 3y \\ (2, y) = y^2 + 1 \\ (1, y) = y \\ (4, y) = y^4 + 3y^2 + 1 \\ ]) \end{array}$$

Observe que o F retorna os polinômios de Fibonacci na forma expandida e consequente são guardados na *remember table* de $F/recursive$ na forma expandida. Podemos agora calcular $F(50, y)$:

```
> readlib(showtime)( ) :
O1 := F(50, y);
```

$$1 + 1166803110y^{16} + \cdots + 1128y^{46} + 16215y^{44}$$

```
time : 0.11
O2 := off;
>
```

Nos procedimentos $fatorial$ e $fibonacci$ não é necessário colocar qualquer comando de simplificão no cálculo recursivo, como foi necessário o comando *expand* no procedimento '$F/recursive$', pois como se trata de multiplicação de números, as simplificações são automáticas. A opção *remember*, por sua vez, não é necessária no $fatorial$. Caso ela seja usada, o procedimento não se tornará mais eficiente, a menos que haja uso sucessivo do procedimento, pois nesse caso os resultados de cálculos anteriores serão usados.

Por outro lado, a separação do procedimento $fatorial$ em dois, numa parte puramente recursiva e noutra parte onde os testes tipo *name* e *nonnegint* são feitos, torna-o mais eficiente. O ganho de velocidade de cálculo pode ser visto através do comando *showtime*( ) para uma aplicação particular.

No procedimento *'F/recursive'* usamos a opção *system*, que permite ao sistema apagar a *remember table* desse procedimento toda vez que houver *garbage collection*, isto é, limpeza de dados sem referência armazenados na memória. O *garbage collection* é feita automaticamente numa certa frequência. O usuário pode forçar a limpeza ou mudar a frequência através do comando *gc*( ).

3.4 Forma Geral

O esqueleto dos procedimentos recursivos que estamos examinando pode ser colocado na seguinte forma geral:

NOME DO PROCEDIMENTO := *proc*(args)
local sequência de nomes;
global sequência de nomes;
options remember;
comando1;
⋮
comandoN;
if teste de finalização da recursão *then* resultado
elif ⋯
elif condição1 *then* ⋯ *procname*(newargs) ⋯
elif ⋯
fi
end

Alguns comandos que aparecem na forma geral acima são opcionais, enquanto que outros são essencias[3].

Caso haja variáveis a serem iniciadas, os comandos de iniciação devem vir antes do comando *if then else fi*. É esse comando que retorna o valor do procedimento, por isso ele deve ser o último comando.

É fundamental para a convergência do processo recursivo de cálculo, que nas sucessivas chamadas do procedimento, os novos argumentos sejam de alguma forma menores que os

[3]Na versão 5.2 e anteriores, o comando *local* é estritamente necessário, caso haja variáveis locais. A ausência desse comando implica que as variáveis do procedimento serão consideradas variáveis globais. Nesse caso, perde-se o controle do valor dessas variáveis a medida que o procedimento chamar a si mesmo no processo recursivo, pois a mesma variável será usada em níveis diferentes da recursão.

originais. Isto é, *newargs* deve de alguma forma ser menor que $args$. A forma de terminar a recursão é estabelecer um valor mínimo para a variável que esteja descrevendo o tamanho dos argumentos. Isso é feito no teste de finalização da recursão.

Se o teste de finalização for verdadeiro, o procedimento sai da recursão e retorna o valor de *resultado*. Portanto a variável *resultado* tem que ter forma correta de saída, isto é, se o procedimento retorna saídas tipo *list*, *resultado* deve ser uma lista, etc. O mesmo ocorre para qualquer outra saída do comando $if\ then\ else\ fi$. Por exemplo, as saídas onde aparecem o comando $procname(newargs)$ também tem que ter o mesmo tipo de *resultado*.

3.5 Procedimento *Split* Recursivo

Para frizar o método de construção de procedimentos recursivos, vamos fazer uma versão recursiva do procedimento *split* da seção 2.4. O procedimento *split* tem uma lista como entrada, e ele separa os elementos repetidos dos não repetidos em 2 conjuntos distintos. A saída é a lista desses conjuntos. Durante a elaboração do procedimento, vamos supor que o procedimento já está pronto, porém ele só pode ser usado em listas menores que a original. Ou seja, se a lista de entrada é

$$L = [\ell_1, \ell_2, \cdots, \ell_n]$$

então podemos usar o procedimento na lista sem o elemento ℓ_1, ou seja podemos usar o comando $split(subs(\ell_1 = NULL\ ,\ L))$ cujo resultado será uma lista na forma

$$\begin{array}{ccc} [\{a_1, a_2 \cdots\} & , & \{b_1, b_2, \cdots\}] \\ \downarrow & & \downarrow \\ \text{conjunto dos} & & \text{conjunto dos} \\ \text{elementos repetidos} & & \text{elementos não repetidos} \end{array}$$

O que precisamos fazer então, é verificar se ℓ_1 é um elemento repetido ou não. Se for, vamos colocá-lo no conjunto $\{a_1, a_2 \cdots\}$, caso contrário vamos colocá-lo no conjunto $\{b_1, b_2, \cdots\}$. O procedimento abaixo segue o que acabamos de afirmar:

```
split4 := proc (L)
local ℓ1;
if L = [ ] then [{ },{ }]
else ℓ1 := L[1];
     RES := procnane(subs(ℓ1 = NULL, L));
     if member (ℓ1, [L[2 ··· nops(L)]])
     then [{ℓ1, op(RES[1])}, RES[2] ]
     else [RES[1], {ℓ1, op(RES[2])}]
     fi
fi
end
```

Observe que todas as saídas do procedimento tem a forma $[\{\cdots\},\ \{\cdots\}]$, inclusive quando a entrada é uma lista nula. Nesse caso, o procedimento não entra na recursão e o resultado

$[\{\ \}, \{\ \}]$ é dado imediatamente. Quando o cálculo entra no processo recursivo, o procedimento é aplicado em listas cada vez menores, e a recursão termina quando o número de elementos da lista de entrada for zero. Para acompanhar o processo de cálculo deve-se aumentar o valor da variável global *printlevel*, que inicialmente é 1:

$$> \ printlevel \ := \ 25;$$

$$printlevel \ := \ 25$$

$$> \ split4([i, i, j, k, i, j]);$$

$$\{-- > \ enter \ split4, \ args = [i, i, j, k, i, j]$$

$$\ell 1 \ := i$$

$$\{-- > \ enter \ split4, \ args = [j, k, j]$$

$$\ell 1 \ := j$$

$$\{-- > \ enter \ split4, \ args = [k]$$

$$\ell 1 \ := k$$

$$\{-- > \ enter \ split4, \ args = [\ ]$$

$$[\{\}, \{\}]$$

$$< -- \ exit \ split4 \ (now \ in \ split4) = [\{\}, \{\}]\}$$

$$RES \ := [\{\}, \{\}]$$

$$[\{\}, \{k\}]$$

$$< -- \ exit \ split4 \ (now \ in \ split4) = [\{\}, \{k\}]\}$$

$$RES \ := [\{\}, \{k\}]$$

$$[\{j\}, \{k\}]$$

$$< -- \ exit \ split4 \ (now \ in \ split4) = [\{j\}, \{k\}]\}$$

$$RES \ := [\{j\}, \{k\}]$$

$$[\{i, j\}, \{k\}]$$

$$< -- \ exit \ split4 \ (now \ at \ top \ level) = [\{i, j\}, \{k\}]$$

$$[\{i, j\}, \{k\}]$$

Podemos ver que o procedimento *split*4 é aplicado a argumentos cada vez menores até chegar a lista nula, cujo resultado é $[\{\}, \{\}]$. A partir daí, os elementos são colocados dentro de $[\{\}, \{\}]$, do último ao primeiro, isto é, primeiramente k foi colocado, depois j e finalmente i. Esse é a forma que os procedimentos recursivos funcionam.

3.6 Multiplicação de Polinômios na Forma de Listas

Vamos construir uma versão recursiva para o procedimento *addpoly* que soma dois polinômios na forma de listas. Um polinômio de uma variável pode ser colocado na forma de uma lista da seguinte maneira:

$$a_n x^n + \cdots + a_1 x + a_0 \rightarrow [a_n, \cdots, a_1, a_0]$$

Assim addpoly $([2,3,0],\ [7,-5,1,2])$ deve retornar $[7,-3,4,2]$ pois

$$(2x^2+3x)+(7x^3-5x^2+x+2) = 7x^3-3x^2+4x+2$$

Para desenvolver o algoritmo desse procedimento, vamos supor que temos 2 polinômios, por exemplo $[a_n, \cdots, a_1, a_0]$ e $[b_m, \cdots, b_1, b_0]$. Na forma como esses polinômios estão escritos, sabemos que o último termo do polinômio soma é a_0+b_0. Os outros termos podem ser encontrados somando as listas restantes que são $[a_n, \cdots, a_2, a_1]$ e $[b_n, \cdots, b_2, b_1]$. Assim, o polinômio soma é

$$[addpoly([a_n, \cdots, a_1], [b_m, \cdots, b_1])\ ,\ a_0 + b_0]$$

Observe que aqui o procedimento *addpoly* foi usado para somar polinômios menores que os originais de forma a garantir o término de recursão. Mais cedo ou mais tarde, um dos argumentos será o polinômio nulo, isto é, uma lista vazia, o que deve terminar a recursão. Assim devemos impor que

$$addpoly\ ([\,],\ [b_m, \cdots, b_0]) \rightarrow [b_m, \cdots, b_0]$$
$$addpoly\ ([a_n, \cdots, a_0],\ [\,]) \rightarrow [a_n, \cdots, a_0]$$

O procedimento fica então da seguinte forma

$addpoly\ :=\ proc(L1, L2)$
$local\ n1, n2;$
$n1\ :=\ nops(L1);$
$n2\ :=\ nops(L2);$
$if\ n1 = 0\ then\ L2$
$elif\ n2 = 0\ then\ L1$
$else\ [op(procname([L1[1..n1-1]], [L2[1..n2-1]])), L1[n1] + L2[n2]]$
fi
end

Assim

$>\ addpoly([4,2,1,-2],\ [-1,-1,2]);$

$$[4,-1,0,0]$$

3.7 Exercícios

1) a) Escreva uma versão para o procedimento $fatorial$, que faz os testes relevantes sobre o argumento e que chama um outro procedimento puramente recursivo.
b) Compare o tempo que leva essa versão para calcular 1000!/999! com o tempo que leva o procedimento da seção 3.1.

2) a) Escreva um procedimento que calcula os polinômios de Laguerre, definidos da seguinte forma

$$L_n(x) = \frac{2n-1-x}{n}\, L_{n-1}(x) - \frac{n-1}{n}\, L_{n-2}(x)$$

$$L_0(x) = 1$$
$$L_1(x) = 1-x$$

b) Calcule $L_{50}(x)$ e compare com o resultado do comando $orthopoly[L](50,x)$. c) Escreva agora um procedimento que calcula os polinômios de Laguerre generalizados, definidos por

$$L_n(a,x) = \frac{2n+a-1-x}{n}\, L_{n-1}(a,x) - \frac{n+a-1}{n}\, L_{n-2}(a,x)$$

$$L_0(a,x) = 1$$
$$L_1(a,x) = 1+a-x$$

3) a) Faça uma versão recursiva do procedimento MDC que acha o maior divisor comum de 2 inteiros.
b) Faça um procedimento recursivo MMC que acha o menor múltiplo comum de 2 inteiros.

4) a) Faça uma versão recursiva do procedimento $maximum$, que acha o maior número entre um conjunto de números.
b) Estenda o procedimento de forma a aceitar variáveis não numérica. Por exemplo:

$> maximum(\{2, Pi, 10, a, -1, b\});$

$$maximum(\{10, a, b\})$$

c) Verifique se sua versão retorna a^2+1 para o conjunto $\{-2, a\hat{}2, a\hat{}2+1\}$. Caso negativo, estenda o procedimento para incluir possibilidades desse tipo.
d) Uma vez que maximum pode retornar não avaliado, estenda o procedimento para aceitar entradas do tipo

$> maximum(\{4, a\hat{}2,\ maximum(\{10, a\hat{}2-1\}),\ maximum(\{20, b, Pi\})\});$

$$maximum(\{20, b, a^2\})$$

5) a) Faça o procedimento $combine(S,n)$ que encontra os subconjuntos com n os elementos do conjunto S. Por exemplo:

$>$ $combine(\{a,b,c,d\},3)$;

$$\{\{a,b,c\},\ \{a,b,d\},\ \{a,c,d\},\ \{b,c,d\}\}$$

b) Estenda o procedimento para que ele admita elementos repetidos. Por exemplo.

$>$ $combine([a,b,b,c],2)$;

$$[[a,b],\ [a,c],\ [b,b],\ [b,c]]$$

(Monagan).
c) Faça um procedimento recursivo que encontra as permutações ímpares ou pares de uma lista. Por exemplo:

$>$ $Permute([a,b,c],odd)$;

$$[[a,c,b],\ [c,b,a],\ [b,a,c]]$$

$>$ $Permute([a,b,c],\ even)$;

$$[[a,b,c],\ [c,a,b],\ [b,c,a]]$$

6) a) Escreva um procedimento recursivo que multiplica polinômios na forma de listas. Por exemplo.

$mulpoly([1/2,-2,1],\ [-1,1])$;

$$[-1/2,\ 5/2,\ -3,\ 1]$$

pois $(1/2x^2-2x+1)*(-x+1) = -1/2x^3+5/2x^2-3x+1$
b) Escreva um procedimento de nome *convert/plist* que converte um polinômio de uma variável em uma lista da seguinte forma:

$$a_n x^n + \cdots + a_1 x + a_0 \rightarrow [a_n, a_{n-1}, \cdots, a_1, a_0]$$

O primeiro argumento do procedimento é um polinômio e o segundo a variável do polinômio. Verifique se o procedimento funciona da seguinte forma:

$>$ $convert(4*x\hat{}3-x,\ plist\ ,\ x)$;

$$[4,0,-1,0]$$

c) Escreva um procedimento análogo de nome *convert/lpoly* que converte uma lista em um polinômio.
d) Faça um procedimento $teste(p1,p2,variavel)$ que testa o procedimento *mulpoly* da seguinte forma: O procedimento converte $p1$ e $p2$ em listas, multiplica-os usando o procedimento *mulpoly* e converte novamente em polinômio. Esse resultado é subtraído de $p1*p2$. O procedimento *teste* deve retornar sempre o valor zero caso os procedimento dos ítens a), b) e c) estejam corretos. Gere polinômios randônicos e faça alguns testes.

7) Uma *linked list* é uma lista que tem uma estrutura recursiva com o terminador NIL. Por exemplo, a lista $[a,b,c]$ é escrita na forma de *linked list* por $[a,[b,[c,NIL]]]$.
a) Escreva um procedimento que retorna uma *linked list* nula de n zeros. Por exemplo:

$> \ linkedzeros(4);$

$$[0,[0,[0,[0,NIL]]]]$$

b) Escreva um procedimento que conta o número de elementos de uma *linked list*. Por exemplo:

$> \ nopsll([a,[b,[c,NIL]]]);$

$$3$$

c) Escreva um procedimento que reverte uma *linked list*. Por exemplo:

$> \ reverse([a,[b,[c,NIL]]]);$

$$[c,[b,[a,NIL]]]$$

d) Crie um novo tipo no Maple chamado *linkedlist* que verifica se uma lista é uma *linked list* ou não. Por exemplo:

$> \ type([a,[b,[c,NIL]]],\ linkedlist);$

$$true$$

As seguintes listas devem retornar false: $[a,b,c,NIL]$, $[NIL,[a,[b,NIL]]]$, $[a,[b,[c,f]]]$. A entrada NIL também retorna falso.

8) Um polinômio de uma variável pode ser colocado na forma de *linked list* da seguinte maneira:

$$a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0 \rightarrow [a_n,[a_{n-1},[\cdots,[a_1,[a_0,NIL]]\cdots]]$$

a) Faça um procedimento que soma polinômios na forma de *linkedlist*
b) Refaça os problemas 6a) até d) no caso de *linked list*.

9) É mais eficiente representar polinômios de grau alto com muitos coeficientes nulos na forma de listas dos coeficientes não nulos apenas. Nesse caso é necessário especificar o grau de cada termo. Por exemplo, o polinômio $2x^{100} + 3x^2 + 1$ é representado por $[[2,100],[3,2],[1,0]]$ ou na forma de linked list por $[[2,100],[[3,2],[[1,0],NIL]]]$. Refaça os problemas 6 e 8 para esse tipo de representação polinômios, que são chamados de esparsos.

4 Mais sobre Procedimentos

4.1 Procedimentos com Número Variável de Argumentos

A maioria dos procedimentos que vimos até agora tinha um número fixo de argumentos. Cada argumento recebia um nome especificado pela sequência de parâmetros de entrada do comando *proc*( ), e esse nome era usado em um ou mais lugares dentro do corpo do procedimento.

Em alguns procedimentos é interessante ter a possibilidade de se variar o número de argumentos, como é o caso do procedimento *maximum* que retorna o maior número entre vários números. Poderíamos nesse caso chamar o procedimento com dois argumentos, por exemplo $maximum(3, 0)$ ou com quatro argumentos $maximum(-1, 5, 0, 2)$. Outro exemplo é o procedimento *addpoly* que adiciona polinômios na forma de listas. É interessante estender esse procedimento para que ele possa somar vários polinômios, e não apenas dois. Assim, ele poderia ser chamado com dois ou mais argumentos dependendo se queremos somar dois ou mais polinômios.

Dentro do corpo de um procedimento as variáveis *args* (arguments) e *nargs* (number of arguments) são variáveis especiais que tem os seguintes valores: A variável *args* representa a sequência dos argumentos e *nargs* representa o número de argumentos com que o procedimento foi chamado. Por exemplo, $args[i]$ representa o i-ésimo argumento e $args[2...nargs]$ representa a sequência de argumentos de entrada sem o primeiro argumento.

Usando as variáveis $args$ e $nargs$, o procedimento *maximum* pode ser escrito da seguinte forma:

```
maximum := proc( )
local max, i;
max := args[1];
for i from 2 to nargs do
    if args[i] > max then max := args[i] fi
od;
max
end;
```

Assim, podemos chamar o procedimento com dois ou três argumentos, por exemplo:

```
> maximum(10, maximum(11, -2, 5));
```

$$11$$

Vamos acompanhar o algoritmo do procedimento com o seguinte exemplo:

```
> printlevel := 10 :
> maximum(-2, 10, 0);
{--> enter maximum, args = -2, 10, 0
```

$$max := -2$$

$$max := 10$$

$$10$$

$<-- \ exit\ maximum\ (now\ at\ top\ level) = 10\}$

$$10$$

Podemos ver que a variável $args$ assume o valor $-2, 10, 0$ que é a sequência de parâmetros de entrada. O primeiro valor que a variável local max recebe é -2 devido ao comando $max\ :=\ args[1]$. O próximo passo é a execução do *loop* com i variando de 2 até 3, onde 3 é o valor de $nargs$. O primeiro teste do comando $if\ then\ fi$ é $10 > 2$. Uma vez que é verdadeiro o comando $max\ :=\ arg[2]$ é executado e retorna $max\ :=\ 10$ na terceira linha. O último teste $0 > 10$ é falso, de forma que o valor da variável max não é modificado. O 10 que aparece na quarta linha é a saída do comando max que é o último comando do procedimento, que nesse exemplo é o valor do procedimento.

Para o procedimento *addpoly* aceitar mais de dois argumentos, ele pode ser modificado da seguinte forma:

$addpoly\ :=\ proc(p1, p2)$
$local\ n1, n2, n, i;$
$if\ nargs > 2\ then\ RETURN(procname(args[1], procname(args[2..nargs])))fi;$
$n1\ :=\ nops(p1);$
$n2\ :=\ nops(p2);$
$n\ :=\ n1 - n2;$
$if\ n < 0\ then\ RETURN(procname(p2, p1))\ fi;$
$[p1[1..n],\ seq(p1[n + i] + p2[i],\ i = 1..n2)]$
end

Observe que apesar da definição do procedimento especificar dois argumentos apenas, $p1$ e $p2$, podemos chamá-lo com três ou mais argumentos. Por exemplo:

$>\ addpoly([3, -2, -1], [5, 0], [-2, 0, 1]);$

$$[1, 3, 0]$$

O primeiro e o segundo argumentos recebem o nome de $p1$ e $p2$ no corpo do procedimento. Os outros argumentos, caso existam, devem ser referidos como $args[3]$, $args[4]$, etc.

Um procedimento pode ser chamado com qualquer número de argumentos independente da sua definição. Porém os parâmetros no corpo do procedimento não podem se referir a um argumento ausente. Por exemplo:

$>\ addpoly([1, 2, 3]);$
$Error,\ (in\ addpoly)\ addpoly\ uses\ a\ 2nd\ argument,\ p2,\ which\ is\ missing$

No caso do procedimento addpoly, podemos acrescentar o comando

$if\ nargs < 2\ then\ RETURN(args)\ fi;$

logo após o comando *local*, para evitar que esse tipo de erro ocorra. Assim

$>\ addpoly([1, 2, 3]);$

$$[1, 2, 3]$$

4.2 Avaliação das Variáveis Locais, Globais e dos Parâmetros

No uso interativo do Maple, algumas variáveis são avaliadas completamente enquanto que outras não. Por exemplo, observe as seguintes atribuições:

```
> a := b:
> b := c:
> c := d:
```

Agora note que

```
> a;
```

$$d$$

O valor que o comando a; retorna é a avaliação completa da variável a. Isso não ocorre para tabelas, *arrays*, matrizes, vetores e procedimentos. O nome dessas estruturas avaliam em si próprio. Para obter a avaliação completa temos que usar o comando *eval*( ). No caso das tabelas, matrizes, arrays e vetores, para se obter uma avaliação completa, inclusive dos elementos é necessário o comando *map*(*eval*, *nome da variável*). Por exemplo:

```
> A := linalg[matrix](2, 2, [z$4]):
> z := 0:
```

Observe agora as diferenças com relação a avaliação:

```
> A;
```

$$A$$

```
> eval(A);
```

$$\begin{bmatrix} z & z \\ z & z \end{bmatrix}$$

```
> map(eval, A);
```

$$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

Esses exemplos foram dados no uso interativo do Maple. O que ocorre dentro de um procedimento? A resposta é: As variáveis globais tem a mesma regra de avaliação dentro e fora dos procedimentos. As variáveis locais, por outro lado, só avaliam em um nível apenas. Veja o seguinte exemplo:

```
f := proc( ) local a, b;
a := b;
b := 33;
a, eval(a)
end;
```

Observe que:

$> \ f(\ );$

$$b, \quad 33$$

Ou seja, no comando $a, eval(a)$, a variável a avaliou em um nível apenas enquanto que $evalb(a)$ avaliou completamente, por isso o resultado é $b, 33$.

Podemos também obter as avaliações intermediárias através do comando $eval(expr, n)$ onde n é um inteiro que diz o nível de avaliação. Por exemplo, no modo interativo

$> \ a \ := \ b :$
$> \ b \ := \ c :$
$> \ c \ := \ 33 :$
$> \ eval(a, 1), eval(a, 2), eval(a, 3);$

$$b, c, 33$$

O resultado será o mesmo se for feito dentro de um procedimento, pois aqui a avaliação está sendo controlada nível por nível.

Por que as variáveis locais são avaliadas em um nível apenas? A resposta é: Por motivo de eficiência. A avaliação completa de uma variável leva mais tempo de computação do que a avaliação em apenas um nível. Na maioria das vezes, as variáveis locais não precisam ser avaliadas completamente. Nos exemplos seguintes vamos construir dois procedimentos iguais a menos de uma única diferença: No primeiro, a variável ℓ é local e no segundo, ela é global. Esses procedimentos calculam a soma dos elementos de uma lista:

$demo1 \ := \ proc(L)$
$local \ \ell, s, t;$
$t \ := \ time(\ );$
$\ell \ := \ L;$
$s \ := \ 0;$
$for \ i \ to \ nops(\ell) \ do \ s \ := \ s + op(i, \ell) \ od;$
$print(cat(`Tempo \ de \ uso \ de \ CPU : \ `, round((time(\ ) - t) * 1000), `\ milisegundos`));$
$print(`Resultado \ da \ soma : \ `.s)$
end

$demo2 \ := \ proc(L)$
$local \ s, t;$
$global \ \ell;$
$t \ := \ time(\ );$
$\ell \ = \ L;$
$s \ := \ 0;$
$for \ i \ to \ nops(\ell) \ do \ s \ := \ s + op(i, \ell) \ od;$
$print(cat(`Tempo \ de \ uso \ de \ CPU : \ `, round((time(\ ) - t) * 1000), `\ milisegundos`));$
$print(`Resultado \ da \ soma : \ `.s)$
end

Vamos comparar agora a eficiência dessas duas versões[4]:

[4]Nas versões 5.2 e anteriores deve-se retirar o comando *global* ℓ do procedimento *demo*2.

$> \ L \ := \ [seq((-1)\hat{}i * a, \ i = 1..100)] :$
$> \ a \ := W(x) :$
$> demo1(eval(L, 1));$

$$Tempo\ de\ uso\ de\ CPU : \ 17\ milisegundos$$

$$Resultado\ da\ Soma : \ 0$$

$> \ demo2(eval(L, 1));$

$$Tempo\ de\ uso\ de\ CPU : \ 4883\ milisegundos$$

$$Resultado\ da\ soma : \ 0$$

Não analisamos ainda a avaliação dos parâmetros de um procedimento. Eles são avaliados completamente ou não? Veja o seguinte procedimento:

$f \ := \ proc(a)$
$a \ := \ 33;$
a
$end :$

Qual é o resultado do comando $f(b)$? Se os parâmetros avaliam em um nível apenas, o resultado é b caso contrário é 33. Vamos tirar a prova:

$> \ f(b);$

$$b$$

$> \ ";$

$$33$$

Podemos ver que os parâmetros, tem a mesma regra de avaliação das variáveis locais, ou seja, eles avaliam em um nível apenas.

4.3 Procedimentos que Retornam mais de um Valor

Existem várias formas de um procedimento retornar um valor ou de mostrar um resultado na tela. A forma que mais usamos até agora foi retornar o valor do último comando. Outra forma que vimos foi interromper o algoritmo e retornar uma mensagem de erro, através do comando ERROR. Nesse caso o valor retornado é NULL.

Podemos também mostrar um resultado na tela com o comando *print*( ) sem interromper o algoritmo. Nesse caso o valor que o procedimento retorna vai depender da continuação do algoritmo. O comando *print*( ) apenas mostra o resultado na tela sem retornar valor algum.

Uma outra forma ainda, é através do comando *RETURN*(*expr*) que finaliza o algoritmo e retorna o valor de *expr*. O comando *RETURN* pode ser colocado em qualquer ponto do programa. Se ele for o último comando do procedimento, não é necessário escrever *RETURN*(*expr*), basta escrever *expr*.

Vamos ver agora uma outra forma de retornar um valor muito usada nos procedimentos do próprio Maple. Por exemplo, o procedimento booleano *member*( ) da *library* do Maple funciona da seguinte forma:

$> \ member(b, [a, a, b, c, b, c], pos);$

$$true$$

$> \ pos;$

$$3$$

O resultado do primeiro comando é *true* porque b é um elemento da lista $[a, a, b, c, b, c]$. Note que esse procedimento retorna um segundo valor que é a posição da primeira ocorrência do elemento b na lista. Esse valor for gravado na variável *pos* que foi usada como terceiro argumento.

Se tentarmos examinar o código do programa *member*( ), nos frustraremos pois

$> \ print(member);$
$proc(\ )\ options\ builtin;\ 101\ end$

A opção *builtin* quer dizer que o procedimento pertence ao *kernel* do Maple e portanto o seu código está escrito na linguagem de programação *C* e não está disponível para o usuário. A lista dos comandos que pertencem ao *kernel* pode ser obtida através do comando ?*internal*.

A lista dos procedimentos para os quais o usuário tem acesso ao código, que está escrito na linguagem de programação do próprio Maple, pode ser obtida através do comando ?*external*. Para ler o código do programa, deve-se usar os comandos

$> \ interface(verboseproc = 2);$
$> \ print(nome\ do\ procedimento);$

Uma vez que o procedimento *member*( ) está escrito na linguagem *C*, vamos construir o procedimento de nome *Member* usando a linguagem de programação do Maple.

$Member\ :=\ proc(x, L, position : name)$
$local\ i;$
$for\ i\ to\ nops(L)\ do$
$\quad if\ L[i] = x\ \ then\ position := i;\ \ RETURN(true)\ fi$
$od;$
$false$
end

Assim

$> \ Member(2, [1, 2, 3, 4]);$

$$true$$

Porém

$> \ Member(b, [a, b, b, c], pos);$
$Error,\ Member\ expects\ its\ 3rd\ argument,\ position,\ to\ be\ of\ type\ name,$
$but\ received\ 3$

Esse erro ocorreu porque os procedimentos avaliam seus argumentos antes de executar o algoritmo. Aqui a variável *pos* já tinha previamente o valor 3. Assim o terceiro argumento não passou no teste dinâmico que verifica se ele é do tipo *name*. Esse problema é evitado usando-se as aspas direitas:

> $Member(b, [a, b, b, c], ' pos')$;

$$true$$

> pos;

$$2$$

4.4 Programação com Variáveis Tipo '.'

Vimos atrás que as variáveis locais avaliam em um nível apenas enquanto que as variáveis globais avaliam completamente. Vimos também que podemos controlar a avaliação nível por nível com o comando $eval(expr, n)$ onde n é um número inteiro que dá o nível da avaliação. O controle da avaliação é crucial quando estamos lidando com variáveis tipo '.'. O ponto é o operador de concatenação, isto é, ele junta os seus argumentos, por exemplo

> $A.B$;

$$AB$$

> $A.B.C$;

$$ABC \qquad \text{\# combinação de '.'}$$

A primeira variável não é avaliada enquanto que todas as outras são, por exemplo:

> $A := 1$:
> $B := 2$:
> $C := z$:
> $A.B.C$;

$$A2z$$

Para que a primeira variável também seja avaliada podemos usar o comando $cat(\ )$ ou entrar a seguinte forma

> "$.A.B.C$;

$$12z$$

Suponha que queremos atribuir uma expressão tipo '.' a uma variável, por exemplo, $expr = A.i.j$, e posteriormente substituir valores numéricos para i e j que vão ser considerados índices da expressão $A.i.j$.

A seguinte forma não funciona:

> $expr := A.i.j;$

$$expr := Aij$$

pois dessa maneira atribuimos a variável Aij a $expr$ e não $A.i.j$. Na variável Aij não podemos substituir valores numéricos para i e j. Para fazer da forma correta, temos que retardar a concatenação:

> $expr := 'A.i.j';$

$$expr := A.i.j.$$

Para substituir o índice i e j por números, a forma $subs(i = 1, j = 2, expr)$ não funciona, pois o comando $subs(\ )$ avalia seus argumentos antes de executar a substituição. A avaliação completa de $expr$ é Aij. Temos então que avaliar $expr$ em um nível apenas:

> $subs(i = 1, j = 2, eval(expr, 1));$

$$A.1.2$$

> $eval(");$

$$A12$$

Uma outra forma de obter o mesmo resultado sem usar o comando *eval(expr,n)* é

> $expr := "A.i.j";$

$$expr := 'A.i.j'$$

> $(eval@subs)(i = 1, j = 2, expr);$

$$A12$$

Nessa forma, o nível de avaliação está sendo controlado apenas pelas aspas.

Vamos agora seguir a construção do procedimento *dotsum* que retorna o somatório de variáveis tipo '.', com os índices variando de 1 até 2. Por exemplo

> $dotsum('A.i.j', [i, j]);$

$$A11 + A12 + A21 + A22$$

Ou seja, *dotsum* tem dois argumentos. O primeiro é uma variável tipo '.' ou uma soma de variáveis tipo '.'. O segundo é a lista dos índices que serão somados de 1 a 2. Assim $dotsum('A.i.j', [i, j])$ calcula

$$\sum_{j=1}^{2} \sum_{i=1}^{2} A.i.j$$

A primeira tentativa de construção do procedimento *dotsum* poderia ser:

$dotsum\ :=\ proc(expr:algebraic,\ L:list)$
$local\ i, s;$
$s\ :=\ expr;$
$for\ i\ in\ L\ do$
$\quad s\ :=\ sum(s, i = 1..2)$
$od;$
s
end

Veja o que ocorre agora:

$>\ dotsum('A.i.j.', [i, j]);$

$$2\ A1j + 2\ A2j$$

Assim, não obtivemos o resultado desejado. Para corrigir essa versão do *dotsum* temos que acompanhar passo a passo a avaliação das expressões dentro do procedimento. Primeiramente vamos aumentar o valor da variável *printlevel*

$>\ printlevel\ :=\ 10:$
$>\ dotsum('A.i.j', [i, j]);$
$\{-->\ enter\ dotsum,\ args = A.i.j, [i, j]$

$$s\ :=\ A.i.j$$

$$s\ :=\ A1j + A2j$$

$$s\ :=\ 2\ A1j + 2\ A2j$$

$<--\ exit\ dotsum\ (now\ at\ top\ level) = 2 * A1j + 2 * A2j\}$

$$2\ A1j + 2\ A2j$$

O primeiro passo foi atribuir $A.i.j$ à variável local s. Isso foi bem sucedido. O segundo passo foi fazer o somatório na variável i de 1 a 2 e atribuir a variável s. O somatório foi bem sucedido, pois s é uma variável local e portanto avaliou em um nível apenas dando $A.i.j$. Assim o índice i pôde ser substituído pelos valores numéricos. O problema ocorreu devido a atribuição do resultado do somatório a s pois o resultado foi avaliado dando $A1j + A2j$ na atribuição.

Vamos usar as aspas direitas para controlar o nível de avaliação, nesse caso o procedimento fica da seguinte forma

$dotsum\ :=\ proc(x : algebraic, L : list)$
$local\ i, s;$
$s\ :=\ "x";$
$for\ i\ in\ L\ do$
$\quad s, i = 1..2;$
$\quad s\ :=\ "sum("")"$
$od;$
$eval("")$
end

Assim

$> \ dotsum('A.i.j', [i, j]);$

$\{-- >$ enter dotsum, args = A.i.j, [i,j]

$$s \ := \ 'A.i.j'$$

$$'A.i.j', \ i \ = \ 1..2$$

$$s \ := \ ' \sum_{i=1}^{2} \ 'A.i.j''$$

$$' \sum_{i=1}^{2} \ 'A.i.j'', \ j \ = \ 1..2$$

$$s \ := \ ' \sum_{j=1}^{2} \ ' \sum_{i=1}^{2} \ 'A.i.j'''$$

$\{-- > \ enter \ sum, \ args \ = \ sum('A.i.j', i \ = \ 1..2), \ j \ = \ 1..2$

$$xx \ := \ j$$

$$a \ := 1$$

$$b \ := \ 3$$

$$dab \ := \ 2$$

$\{-- > \ exit \ sum \ (now \ in \ dotsum) \ = \ A11 + A21 + A12 + A22\}$

$$A11 + A21 + A12 + A22$$

$\{-- > \ exit \ dotsum \ (now \ at \ top \ level) \ = \ A11 + A21 + A12 + A22\}$

$$A11 + A21 + A12 + A22$$

Usamos duas aspas nos comandos $s := \ ''x''$ e $s := \ ''sum(")''$ porque uma delas é retirada pelo comando de atribuição enquanto que a outra é usada no argumento do comando $sum(\ )$ como pode ser visto nos resultados intermediários devido ao *printlevel* alto. Note que colocamos a expressão $s, i = 1..2$ fora do comando $''subs(")''$ pois queremos que s e i sejam avaliados em um nível, o que não ocorreria se eles estivessem dentro das aspas do comando $''subs(")''$. Uma outra forma de forçar a avaliação de subexpressões dentro de aspas é através do comando $subs(\ )$ que aqui ficaria da seguinte forma

$$subs('dummy' = (s, i = 1..2), \ ''sum(dummy)'')$$

Vimos no capítulo 2, que os *loops for do od* são, em geral, pouco eficientes comparados com o comando $seq(\ )$ pois eles geram uma grande quantidade de atribuições intermediárias o que não ocorre com o comando $seq(\ )$.

O operador de concatenação pode ser usado das seguintes formas:

$> \ A.(1..2);$

$$A1, A2$$

> $A.(1..2).(1..2)$;

$$A11, A12, A21, A22$$

Esse último resultado pode ser convertido em uma expressão tipo ' + ':

> $convert([''], ' + ')$;

$$A11 + A12 + A21 + A22$$

Usando essa propriedade do operador de concatenação, pode-se fazer uma versão mais eficiente do procedimento *dotsum* tal que as expressões tipo

$$\sum_{j=1}^{2} \sum_{i=1}^{2} A.i.j$$

sejam geradas com comandos tipo $convert([A.(1..2).(1..2)], ' + ')$.

4.5 Programação sobre a Estrutura das Expressões Algébricas

Suponha que temos uma expressão algébrica onde aparecem termos na forma de raíz quadrada, por exemplo

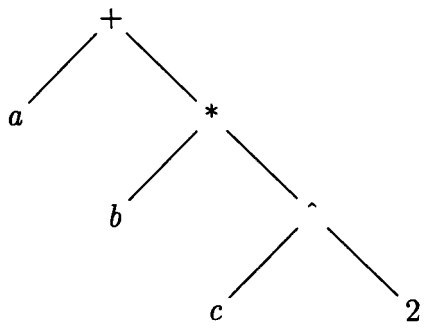
$$expr \;=\; \frac{(a + \sqrt{2})(\sqrt{a} + 2)}{\sqrt{3}\; b}$$

Queremos construir o procedimento *sroots* que extrai os termos que são raízes quadradas. Assim

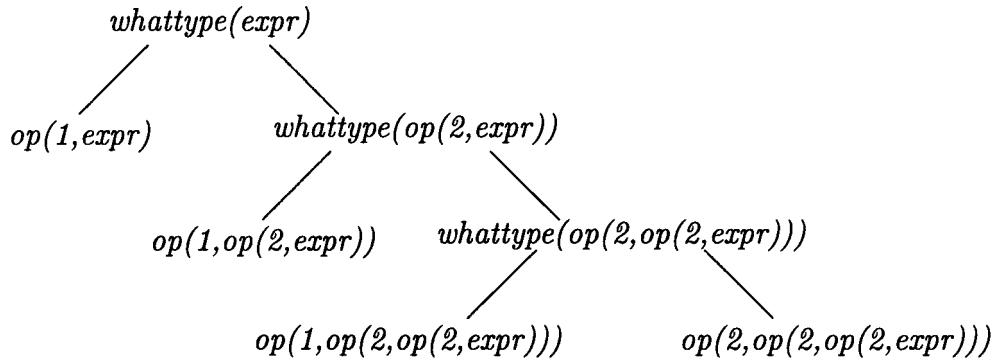
> $sroots(expr)$;

$$2^{1/2},\; 3^{1/2},\; a^{1/2}$$

Uma expressão algébrica no Maple pode ser vista, em primeira aproximação, como uma árvore onde as bifurcações dos galhos são caracterizadas pelos operadores ' + ', ' * ' e '^' e as folhas são os nomes e os números inteiros. A árvore da expressão $a + b * c^2$ pode ser colocado na forma:



O primeiro operador (1º nível) é chamado de operador de superfície, que aqui é o ' + '. Observe que essa expressão tem três níveis de operadores. Os operadores e as 'folhas' são obtidos com o comando *whattype*( ) e *op*( ) respectivamente. No caso da expressão $expr = a + b * c^2$ temos a seguinte correspondência:



Quanto mais profundo for o nível, mais $op's$ encaixados serão necessários para se extrair um elemento da expressão.

Geralmente, o método recursivo de programação é o mais indicado para ser usado em procedimentos que devem penetrar em todos os níveis de uma expressão algébrica. Se o tipo da expressão é ' + ', ' * ' ou '^ ', o procedimento deve partir a expressão nas partes dadas pelo comando *op*( ) e continuar a partir as sub-expressões. Se o tipo da expressão é *name* ou *integer*, a recursão deve finalizar. Dessa forma a expressão é quebrada nas suas partes elementares. Se as condições de finalização da recursão envolvem tipos mais gerais, que incluem os tipos *name* e *integer*, a expressão não será partida em "átomos" mas sim em "moléculas".

Vamos aplicar a teoria de programação recursiva para o procedimento *sroots*. No método recessivo, assume-se que o procedimento já está pronto, porém só pode ser usado em expressões menores que a original. Suponha que o argumento seja uma expressão tipo '+', por exemplo, $A + B + C$ onde A, B e C podem ser também expressões algébricas. Assim $sroots(A + B + C)$ deve retornar

$$sroots(A), sroots(B), sroots(C)$$

pois a saída de cada um dos termos acima é uma sequência de raízes que juntam para formar uma sequência maior. Note que *sroots* foi aplicado à expressões menores que a original. Sabemos que há duas formas de se obter o resultado acima, a partir da expressão $expr = A + B + C$. A primeira é

$$seq(sroots(i), i = expr)$$

e a segunda é

$$op(map(sroots, \{op(expr)\}))$$

A primeira forma não elimina as raízes repetidas. Para tal, tem-se que fazer a seguinte modificação

$$op(\{seq(sroots(i), i = expr)\})$$

O mesmo método se aplica às expressões tipo '*' e '^'.

Para finalizar a recursão, tem-se que considerar os seguintes casos. Se a expressão for tipo *sqrt*, o procedimento *sroots* deve retornar a própria expressão de entrada. Se a expressão for tipo *name* ou *numeric*, *sroots* deve retornar *NULL*.

O procedimento *sroots* pode ser escrito na forma

```
sroots := proc(expr:algebraic)
local i;
if type(expr, sqrt) then expr
elif type(expr, {name, numeric}) then NULL
elif type(expr, {'+', '*', '^'}) then op({seq(procname(i), i = expr)})
elif ERROR(cat('sroots does not deal with expression type ', whattype(expr)))
fi
end
```

Assim

```
> expr := (a + sqrt(2)) * (sqrt(a) + 2)/(sqrt(3) * b);
```

$$expr := 1/3\ \frac{(a + 2^{1/2})(a^{1/2} + 2)3^{1/2}}{b}$$

```
> sroots(expr);
```

$$3^{1/2},\ 2^{1/2},\ a^{1/2}$$

Note que usamos um conjunto de tipos no comando *type(expr, {name, numeric})*. Esse comando é equivalente a

$$type(expr, name)\ or\ type(expr, numeric)$$

Existem várias formas de se combinar tipos no Maple, na verdade existe uma álgebra de tipos. Por exemplo, podemos usar o tipo $name\hat{}integer$, tal que $type(expr, name\hat{}integer)$ retorna *true* se a expressão for do tipo *name* elevada a um expoente inteiro, e *false* caso contrário. Para saber quais são as possibilidades da álgebra dos tipos usa-se o comando $?type, structured$.

Um usuário pode criar novos tipos que serão reconhecidos pelo comando *type( )*. Para isso, deve-se construir um procedimento booleano que tenha o nome da seguinte forma:

$$`type/nome\ do\ tipo` := proc(expr, ...)$$

A chamada $type(expr, nome\ do\ tipo)$ do comando *type( )* usual do Maple é equivalente a chamada $`type/nome\ do\ tipo`(expr, ...)$ do procedimento construído pelo usuário.

Vamos dar um exemplo de construção do tipo *irrational*. O Maple tem implementado os tipos *rational* e *realcons* que servem para verificar se uma variável pertence ao domínio dos números racionais e reais, respectivamente. Observe que o tipo *irrational* não existe usualmente:

```
> type(sqrt(2), irrational);
Error, type irrational does not exist in maple
```

Vamos construir o seguinte procedimento booleano:

$$\begin{array}{l} `type/irrational` \ := \ proc(x) \\ type(x, realcons) \ and \ not \ type(x, rational) \\ end \end{array}$$

Agora

$$> \ type(sqrt(3), irrational);$$

$$true$$

$$> \ type(2/5, irrational);$$

$$false$$

Se analisarmos de maneira geral o procedimento $sroots$, podemos ver que o tipo de expressão que esse procedimento retorna ou é uma sequência ou um caso particular de sequência que pode ser um único elemento ou a sequência nula (NULL). Vamos construir agora um outro procedimento chamado $fsroots$ que está relacionado com o $sroots$. O procedimento $fsroots$ retorna uma expressão do mesmo tipo da expressão de entrada, porém aplica uma função que vem especificada como primeiro argumento em todas as sub-expressões tipo $sqrt$. Por exemplo:

$$> \ expr \ := \ sqrt(2) + sin(sqrt(a) * b + 3);$$

$$expr \ := \ 2^{1/2} + sin(a^{1/2} b + 3)$$

$$> \ fsroots(f, expr);$$

$$f(2^{1/2}) + sin(f(a^{1/2}) \ b + 3)$$

$$> \ fsroots(x \ -> \ x\hat{}2, expr);$$

$$2 + sin(a \ b + 3)$$

As condições de finalização para o procedimento $fsroots$ são iguais ao do $sroots$, porém o tipo de saída do comando $op(\{seq(procname(i), i = expr)\})$ do $sroots$ é um sequência. No $fsroots$ a saída deve ter o mesmo tipo de entrada, assim temos que usar o comando $map(\ )$ no lugar de $seq(\ )$ da seguinte maneira:

$$map(\ (a, b) -> fsroots(b, a), \ expr \ , f)$$

ou

$$map(\ unapply(procname(b, a), a, b), \ expr, \ f)$$

Observe que a função $(a, b) -> fsroots(b, a)$ inverte os argumentos, caso contrário o resultado do comando acima seria algo do tipo $fsroots(sub-expr, f)$ o que resultaria em erro pois é o procedimento f que deve vir como primeiro argumento e não $sub–expr$. O mesmo erro seria cometido se usássemos $map(fsroots, expr, f)$. Na versão com o operador $->$ não podemos usar $procname$, pois esse operador não avalia seus argumentos, porém na versão com o operador $unapply(\ )$ é permitido. O procedimento $fsroots$ fica então:

```
fsroots := proc(f:{name,procedure},expr)
if type(expr,sqrt) then f(expr)
elif type(expr,{name,numeric}) then expr
elif type(expr,{`+`,`*`,`^`}) then map(unapply(procname(b,a),a,b),expr,f)
else ERROR( cat(`2nd argument cannot be type `, whattype(expr)))
fi
end
```

No Maple existem os tipos *function* e *procedure* que são tipos bem distintos. O que estamos chamando de função não é tipo *function* mas sim tipo *procedure*, por exemplo:

```
> type(x-> 1/x,function);
```

$$false$$

```
> whattype(x-> 1/x);
```

$$procedure$$

O tipo *function* se refere a seguinte estrutura: nome(args). Ou seja um nome aplicado a argumentos entre parênteses. Por exemplo

```
> type(sin(x),function);
```

$$true$$

As expressões tipo *function* são exemplos de expressões que tem o termo zero:

```
> expr := f(arg1,arg2,arg3);
```

$$expr := f(arg1, arg2, arg3)$$

```
> op(0,expr);
```

$$f$$

```
> op(expr);
```

$$arg1, arg2, arg3$$

```
> op(0..nops(expr));
```

$$f, arg1, arg2, arg3$$

4.6 Exercícios

1) Modifique o procedimento $fibonacci$ da seção 3.2 de forma que ele admita uma sequência de números inteiros positivos como argumentos. A saída deve ser a sequência dos números de Fibonacci correspondentes, por exemplo:

$> \ fibonacci(\$1..7);$

$$0, 1, 1, 2, 3, 5, 8$$

$> \ fibonacci(5, 8, -2);$
$Error, \ (in \ fibonacci) \ the \ arguments \ must \ be \ positive \ integers$

2) a) No problema 2 do capítulo 3, pedia-se para construir um procedimento para os polinômios de Laguerre e outro para os polinômios de Laguerre generalizados. Faça agora uma única versão do procedimento de tal forma que, se for chamado com dois argumentos, ele retorna o polinômio de Laguerre e se for chamado com três argumentos, ele retorna os polinômios de Laguerre generalizado.
b) Compare a sua versão do procedimento com a versão *orthopoly[L]* do próprio Maple.

3) Faça uma versão do procedimento $Member$ tal que $Member(a, L, pos)$ retorna $true$ ou $false$ caso a seja ou não elemento de L como antes, porém agora com a seguinte modificação: Se o elemento a ocorrer mais de uma vez em L, pos deve fornecer a sequência de posições de a em L. Por exemplo:

$> \ Member(b, [a, a, b, c, b, b,], position);$

$$true$$

$> \ position;$

$$3, 5, 6$$

O valor da variável *position* não deve ser modificado caso o elemento não pertença a lista ou ao conjunto.

4) Extenda o procedimento $Permute$ do problema 5 do capítulo 3 tal que ele possa ter três argumentos: Uma lista, o nome *odd* ou *even* e uma variável. Se o segundo argumento for *odd*, por exemplo, o procedimento retorna as permutações ímpares da lista e grava as permutações pares na variável que é dada no terceiro argumento.

5) Faça uma versão recursiva do procedimento *dotsum* da seção 4.4.

6) a) Faça um procedimento tal que permita o comando *convert*( ) converter variáveis tipo '.' numa lista ou num conjunto. Por exemplo:

$> \ convert('A.i.j', list);$

$$[A, i, j]$$

b) Faça agora o procedimento que permita o inverso:

> $convert([A, i, j], \text{`.`})$;

$$A.i.j$$

> $convert(\{a, b, c\}, \text{`.`})$:

$$a.b.c$$

7) a) Modifique o procedimento *dotsum* da seção 4.4 tal que não seja mais necessário fornecer os índices de soma. Por exemplo

> $dotsum('A.i.j')$;

$$A11 + A12 + A21 + A22$$

b) Caso a entrada seja uma soma de variáveis tipo '.', o procedimento só deve retornar o resultado caso os índices sejam compatíveis, caso contrário deve retornar uma mensagem de erro. Por exemplo:

> $dotsum('A.i + 2 * B.i')$;

$$A1 + A2 + 2\ B1 + 2\ B2$$

> $dotsum('A.i.j + B.i.k')$;
> *Error (in dotsum) wrong choice of the indices*

8) a) Faça uma versão mais eficiente do procedimento *dotsum* usando a concatenação com *range*. Ou seja, se a entrada for $A.i.j$, então o procedimento gera a expressão $A.(1..2).(1..2)$ e depois converte o resultado para uma expressão tipo '+'.
b) Compare a eficiência dessa versão com a versão do exercício 7.

9) Faça um procedimento de nome *produtotensorial* tal que, uma vez dada uma expressão algébrica que contém termos tipo '.', ele soma de 1 a 2 somente os índices repetidos que aparecem numa mesma variável ou num produto de variáveis tipo '.'. Por exemplo:

> $produtotensorial('A.i.j.j')$;

$$A.i.1.1 + A.i.2.2$$

> $produtotensorial('A.i * B.i.j')$;

$$A.1\ B.1.j + A.2\ B.2.j$$

> $produtotensorial('A.i + B.j.j * C.i')$;

$$A.i + (B.1.1 + B.2.2)\ C.i$$

Os índices que não aparecerem repetidos serão considerados livres e não devem ser somados. Não é permitido um índice aparecer mais de duas vezes numa mesma variável ou num produto de variáveis tipo '.'. Por exemplo:

> $produtotensorial('A.i.i * B.i')$;
> *Error (in produtotensorial) indices repeated more than 2 times*

Se a entrada for uma soma, cada termo tem que ter os mesmos índices livres, por exemplo

> $produtotensorial('A.i.j + B.i.i.j')$;
> *Error (in produtotensorial) wrong choice of free indices*

Os índices livres de $A.i.j$ são i e j enquanto que de $A.i.i.j$ é j.

Não é permitido termos tipo '.' com índices livres dentro de expoentes. Por exemplo

> $produtotensorial('A.i/B.j')$;
> *Error (in produtotensorial) free index in power*

> $produtotensorial('A.i/B.j.j\hat{}2')$;

$$\frac{A.i}{(B.1.1 + B.2.2)^2}$$

10) a) Como se modifica o procedimento *sroots* para que ele não selecione as raízes quadradas que aparecem em expoentes. Por exemplo:

> $sroots(sqrt(2)\hat{}sqrt(3))$

$$2^{1/2}$$

b) Como se modifica *sroots* para que ele selecione também as raízes que estão dentro de outras raízes, por exemplo:

> $sroots(2 * sqrt(2 + sqrt(3)))$;

$$3^{1/2},\ (2 + 3^{1/2})^{1/2}$$

c) Como se modifica *sroot* para que ele forneça todos os radicais e não apenas as raízes quadradas. Além disso ele deve selecionar também os radicais dentro de funções. Por exemplo:

> $sroots(sqrt(2) *\ sin(2\hat{}(2/3) * a)/sqrt(3 + a\hat{}(5/2)))$;

$$2^{1/2}, 2^{2/3}, a^{5/2}, \frac{1}{(3 + a^{5/2})^{1/2}}$$

11) Faça um procedimento que retorna a sequência de todas as variáveis indexadas de uma expressão. Por exemplo

> $indexednames(a[1] + a[2]sin(b[1] * a)/(c[1,2] * 2\hat{}d[1,2]))$;

$$a[1],\ a[2],\ b[1],\ c[1,2],\ d[1,2]$$

12) Faça um procedimento de nome *typeselect* tal que $typeselect(expr, tipo)$ seleciona as sub-expressões de *expr* que tenha o tipo especificado no segundo argumento. A saída deve ser uma sequência. Por exemplo:

$>$ $typeselect(2\hat{}(1/2) * sqrt(1 + 3\hat{}(1/2)),\ sqrt);$

$$2^{1/2},\ 3^{1/2},\ (1 + 3^{1/2})^{1/2}$$

$>$ $typeselect(2 * Pi(a + 3 * b + exp(x\hat{}2)), name);$

$$Pi,\ a,\ b,\ x$$

Os tipos a serem considerados são: *indexed*, *name*, *function* e todos os tipos numéricos.

13) Faça um procedimento de nome *typemap* tal que $typemap(f, expr, tipo)$ aplica o procedimento f à toda sub-expressão de *expr* que tem o tipo especificado no terceiro argumento. O procedimento *typemap* retorna uma expressão do mesmo tipo da expressão de entrada. Por exemplo:

$>$ $expr\ :=\ 2 + a + exp(1 + b\hat{}2);$

$$2 + a + exp(1 + b^2)$$

$>$ $typemap(f, expr, name);$

$$2 + f(a) + expr(1 + f(b)^2)$$

$>$ $typemap(g, expr, integer);$

$$g(2) + a + exp(g(1) + b^{g(2)})$$

14) Estenda o procedimento GCD da seção 2.2 para ele aceitar mais de dois argumentos podendo ser não numéricos. Os seguintes exemplos devem ser satisfeitos:

$GCD(a) \rightarrow abs(a)$
$GCD(o, a) \rightarrow abs(a)$
$GCD(a, b) - GCD(b, a) \rightarrow 0$
$GCD(a, GCD(b, c)) \rightarrow GCD(a, b, c)$
$GCD(a, GCD(b, GCD(c, -d))) \rightarrow GCD(a, b, c, d)$
$GCD(30, 70, 42) \rightarrow 2$
$GCD(a, 20, 15) \rightarrow GCD(a, 5)$

(Monagan).

15) a) Seja A uma matriz cujos coeficientes são inteiros com um algarismo. Uma matriz desse tipo pode ser gerada com o comando $linalg[randmatrix](3, 3, entries = rand(1..9))$. A partir dessa matriz é possível gerar números de 1 até 9 algarismos juntando elementos adjacentes da matriz. Por exemplo, se A é a matriz

$$\begin{bmatrix} 8 & 8 & 2 \\ & \searrow & \\ 1 \leftarrow & 2 \leftarrow & 5 \\ & & \\ 2 & 9 & 2 \end{bmatrix}$$

podemos gerar o número 8521 conforme mostrado através das setas. Um elemento só pode ser usado uma única vez na construção de um número. Faça um procedimento que dá o número de números primos que podem ser formados dessa forma. No exemplo acima temos 183 números primos.
b) Estenda o procedimento do ítem a) para ele aceitar matrizes quadrada de qualquer dimensão. Por exemplo, se a matriz for

$$\begin{bmatrix} 4 & 7 \\ 5 & 8 \end{bmatrix}$$

então o resultado deve ser 8, pois podemos construir os seguintes números primos: 5, 7, 47, 587, 487, 547, 857, 457.

Apêndice – Sugestões para alguns exercícios

Capítulo 1

2) a) Use o operador $ ou o comando *seq*( )
b) Converta a lista para conjunto
d) Use o comando $type(\cdots, integer)$

3) Use o comando *signum* ou $Heaviside$

5) a) Use o comando $rsolve$( )
b) Componha a função $x \text{–>} normal(x, expanded)$ com a função $unapply(\cdots, n)$.

6) Examine o resultado do seguinte comando:

$> \ seq(L[4-i], i = 1..3);$

7) 1ª forma: Selecione os elementos da lista que são iguais ao primeiro argumento e conte o número de elementos. Cuidado com funções encaixadas.
2ª forma: Conte o número de elementos da lista original e subtraia do número de elementos da lista que não contenha o primeiro argumento. Use o comando *subs*( ) e $NULL$. Qual forma é mais eficiente?

8) Use o fato de que se o número de ocorrências do primeiro argumento na lista for maior ou igual a 1, ele pertence a lista, se for zero ele não pertence a lista.

9) Use o comando *select*( ) duas vezes, uma para selecionar os elementos que ocorrem mais de uma vez e outra para selecionar os que ocorrerem uma vez.

10) 1ª forma: Faça um *loop* com o comando *for do od* de forma a eliminar a primeira ocorrência do elemento na lista.

2ª forma: Use o comando *member( )* para obter a posição de primeira ocorrência do elemento na lista e então use o comando *subsop( )*. Qual forma é mais eficiente?

11) Faça uma função booleana que verifica se uma lista tem elementos repetidos ou não. A partir dessa função use o comando *select( )*.

12) Some as funções *lcoeff* e *tcoeff* e então aplique à expressão expandida e à variável.

13) Use o comando *convert(*$\cdots$, ‘ + ‘*)*.

14) 1ª forma: Use o comando *map(unapply,*$\cdots$*)*.
2ª forma: Use o comando *map( )* para transformar cada elemento da matriz em uma função. Use o comando *subs( )* para repassar o parâmetro para dentro da função mais interna.

15) Use o comando *linalg[innerprod]* para o produto interno. É necessário usar também os comandos *evalm( )* e *map(normal,···)*.

Capítulo 2

2) Use o modelo da seção 2.3 e faça as modificações específicas de cada ítem.

3) Tome L como sendo um conjunto de números positivos e negativos, e f como sendo a função *abs( )*.

4) a) Crie primeiro uma função que faz um nível de encaixe. Use o comando *seq( )* e o operador @@.

7) a) Inicie uma variável como sendo o primeiro elemento do conjunto. Faça um *loop* sobre o conjunto. Se o i-ésimo elemento for maior que a variável, modifique o valor da variável, caso contrário não modifique. O resultado deve ser o valor da variável.
b) Use *signum( )* ou *Heaviside( )*.

9) Transforme a primeira lista em uma lista de números naturais. Use o comando *member( )* para converter a segunda lista nos números correspondentes. Por exemplo

$$[a, b, c,] \text{ e } [b, c, a] \rightarrow [1, 2, 3] \text{ e } [2, 3, 1]$$

Ordene a segunda lista de forma que a cada troca de pares de números, uma variável iniciada em *true* ou *false* inverte o seu valor. O resultado é o valor variável.

10) Use o comando *permute( )* do Maple e selecione as listas com o comando *ispermute* do exercício 9.

Capítulo 3

3) a) Use o fato de que *MDC*(a,b) é igual a *MDC*(a, r), onde r é o resto da divisão entre a e b.

4) a) Elimine o primeiro elemento do conjunto e use o próprio procedimento. Compare o primeiro elemento com as possíveis saídas do comando recursivo.
b) Igual a sugestão de a), porém agora todas as possibilidades de saída da chamada recursiva com um elemento a menos devem ser rigorosamente analizadas. Por exemplo, se o conjunto original é $\{a, b, c, \cdots\}$, a chamada recursiva é $maximum(\{b, c, d, \cdots\})$. As possibilidades de saída podem ser um número, uma variável tipo nome, ou até mesmo $maximum(\{i, j, \cdots\})$. Cada um desses casos deve ser analisado.
d) No início do algoritmo, junte todos os conjuntos num só, por exemplo, $maximum(\{a, maximum(\{b, c\})\})$ é equivalente a $maximum(\{a, b, c\})$.

References

[1] André Heck, *Introduction to Maple*, Springer-Verlag, 1993.

[2] Michael Monagan, *"Programming in Maple: The basics"*, *Share Library* do Maple V.3 ou *anonymous@neptune.inf.ethz.ch* arquivo *pub/maple/5.0/doc/program.tex*

[3] B.W. Char, K.O. Geddes, G.H. Gonnet, B.L. Leong, M.B. Monagan and S.M. Watt, *Maple V Library Reference Manual*, Springer, first edition. 1991.

[4] B.W. Char, K.O. Geddes, G.H. Gonnet, B.L. Leong, M.B. Monagan and S.M. Watt, *Maple V Language Reference Manual*, Springer, first edition, 1991.

[5] B.W. Char, K.O. Geddes, G.H. Gonnet, B.L. Leong, M.B. Monagan and S.M. Watt, *First Leaves: A Tutorial Introduction*, Springer, first edition, 1992.