CBPF-MO-003/98

"FORM versão 3.0"

Patrícia Duarte Peres
Patricia Macedo da Costa Jorge
Renata Alves Campos
Viviane Monteiro Braconi
Letícia M. Gonçalves Furtado
Andreas Christo Koussoula Mathiel

Universidade Católica de Petrópolis
e
Centro Brasileiro de Pesquisas Físicas
(UCP - CBPF)

Julho 1998

Utilizando o FORM 3.0

Introdução

Apresentação

Uma primeira noção que se deve ter de um programa FORM é a de que este consiste de uma seqüência de declarações e instruções que executam os cálculos desejados. Diferentemente de outros programas de computação algébrica, onde o cálculo é feito interativamente com o usuário através de ambientes de desenvolvimento integrado, um trabalho em FORM é desenvolvido como um sistema em lotes. Nos sistemas de computação algébrica um trabalho é realizado através de frequentes interações do usuário com o sistema. Os cálculos são feitos em uma sessão, consistindo de instruções enviadas ao sistema e respostas a estas instruções. Ao contrário, no FORM não se tem uma sessão de interação com o usuário, mas um programa que consiste de uma seqüência de declarações e comandos que realizarão os cálculos desejados. Um programa FORM pode ser visto como um lote (ou seja, um conjunto) de instruções que serão executadas uma após a outra até o fim do programa, sem nenhuma interferência do usuário.

A comparação do FORM com outros programas de computação algébrica, como **Maple** ou **Derive**, não é muito coerente. Estes sistemas são voltados para cálculos completamente diferentes dos que podem ser realizados com o uso do FORM. Ele foi desenvolvido, basicamente, para auxiliar o trabalho que possua pelo menos uma destas características:

- manipulação de grandes fórmulas;
- padrão de substituição em fórmulas;
- cálculo com álgebra não-comutativa;
- cálculo em Física de Altas Energias.

Esta apostila consiste de uma introdução ao FORM. Serão discutidos os principais comandos e características do mesmo. O curso não esgotará por completo o assunto, sendo a consulta ao manual do FORM e a outros livros, com vistas ao apronfundamento, saudável e aconselhável.

Primeiros Exemplos

Para realizar um cálculo usando o FORM, temos que escrever um programa, salvá-lo em um arquivo texto, e executar o FORM passando este arquivo como argumento na linha de comando do sistema operacional. Vamos examinar nosso primeiro exemplo: suponha que temos um arquivo **ex1.frm** com o seguinte conteúdo:

```
Symbols a, b;
Local exp = (a+b)^2;
Print;
.end
```

Para processar este programa devemos executar o FORM com o comando **form ex1.frm**. Feito isto, o seguinte resultado deverá ser visto na tela do computador:

```
FORM version 3.

Symbols a,b;
Local exp = (a+b)^2;
Print;
.end

 Time =       0.00 sec    Generated terms =          3
               exp        Terms in output =          3
                          Bytes used      =         52

 exp =
    2*a*b + a^2 + b^2;
```

Examinemos este exemplo e o resultado que nos fornece.

A primeira linha deste programa é uma declaração. **Symbols a, b** declara os objetos **a** e **b** como símbolos. Símbolos são os objetos mais simples em um programa FORM . Eles são entendidos como constantes e podem comutar com todos os outros tipos de objeto.

A segunda linha também é uma declaração. **Local exp** indica que **exp** é uma expressão local. A instrução **Local** é uma forma de declarar e de introduzir no cálculo a expressão que será trabalhada pelo FORM .

O comando **Print**, que aparece na terceira linha, fará com que o resultado seja mostrado ao usuário. Embora possa parecer, **Print** não tem efeito imediato. O resultado do cálculo só será exibido ao final do programa.

A última linha não é uma instrução, mas uma diretiva. **.end** indica que o programa termina neste ponto, e que a execução dos comandos anteriores pode começar. É importante observar que até este ponto nenhum comando é executado. Em um programa pode haver uma ou mais diferentes diretivas, que indicam a separação do programa em módulos.

Quando a execução do programa termina, as estatísticas são mostradas. Elas informam o nome da expressão, o número de termos que foram gerados, o número de termos que resultaram e os bytes que a expressão ocupa. Estas estatísticas podem ser suprimidas com o uso do comando **NWrite Statistics**.

Em seguida é mostrado o resultado final da expressão. Ao examinarmos o resultado que o FORM nos fornece, podemos observar que, embora nenhum comando tenha sido usado com este objetivo, a expressão local declarada como **exp** foi automaticamente expandida e simplificada. O FORM , sempre que possível, tentará eliminar todos os parênteses.

Este exemplo ilustrou a estrutura básica de um programa FORM : declarações, comandos e diretivas. Todavia, como veremos no exemplo que segue, estas estruturas podem se repetir:

```
FORM version 3.

NWrite Statistics;
Symbols a, b;
Symbols c, d;
Local exp1 = a+b;
Local exp2 = c+d;
Local [exp1+exp2] = exp1+exp2;
Local [exp1*exp2] = exp1*exp2;
Print;
.end

exp1 =
   a + b;

exp2 =
   c + d;

[exp1+exp2] =
   a + b + c + d;

[exp1*exp2] =
   a*c + a*d + b*c + b*d;
```

Examinemos também este programa.

As duas primeiras linhas repetem a instrução **Symbols**, caracterizando os objetos **a**, **b**, **c** e **d** como símbolos. As quatro instruções seguintes identificam quatro novas expressões locais. Observe que as duas primeiras, **exp1** e **exp2**, foram introduzidas como no primeiro exemplo. As duas seguintes, **[exp1+exp2]** e **[exp1*exp2]**, entretanto, foram construídas usando as duas primeiras. Instruções desse tipo são perfeitamente válidas e a sua compreensão é útil para os casos em que se faz necessário compor uma expressão com o resultado de outras.

Um novo fato que surge neste exemplo é o de que as expressões, **[exp1+exp2]** e **[exp1*exp2]**, foram declaradas entre colchetes. O ponto importante está no fato de que o FORM não verifica o que estiver dentro dos colchetes, de forma que seu conteúdo fica a critério do usuário. Em nosso exemplo, **[exp1+exp2]** é apenas o nome de uma expressão, não significa **exp1+exp2**.

Os comandos **Print** e **NWrite Statistics** desempenham as funções que discutimos acima. O primeiro mostra o resultado ao usuário e o segundo desabilita a apresentação das estatísticas.

A última diretiva indica o término do programa.

Considerações Gerais

Os itens que seguem enumeram alguns detalhes da estrutura geral de um programa FORM.

- Todo programa FORM deve possuir a extensão.**frm**
- Toda instrução FORM consiste de uma palavra-chave, como **Symbols** e **Print**, e uma lista de um ou mais parâmetros. Uma instrução pode se extender por várias linhas, mas deve sempre terminar com ponto-e-vírgula. O FORM ignora o que vier, na mesma linha, após o ponto-e-vírgula.

- O FORM não distingue entre declarações no singular ou plural. **Symbols** e **Symbol** são o mesmo comando, **Functions** e **Function** têm a mesma função.
- Para a maioria dos comandos há uma abreviação: **Symbols** pode ser abreviado para **S**, **Print** pode ser abreviado para **P**, e **Local** para **L**.
- Itens em uma instrução podem ser separados por vírgulas ou por espaços. Na maioria das vezes, o FORM não faz distinção entre eles.
- O FORM não é sensitivo com relação às palavras-chave e funções reservadas. Uso de letras maiúsculas ou minúsculas é indiferente nestes casos. Entretanto, para variáveis e expressões definidas pelo usuário haverá distinção.
- Você pode criar um arquivo com o resultado de um programa. Isto é feito através do parâmetro **-l** na linha de comando de execução do FORM a partir do sistema operacional. Por exemplo, **form -l ex1.frm** criará o arquivo **ex1.log** com exatamente o mesmo conteúdo que é mostrado na tela.

Tipos de Variáveis

Na seção anterior discutimos dois exemplos que fizeram uso de variáveis declaradas como símbolos. Mas, observem que, devido à sua simplicidade, símbolos não são úteis para a representação de grandezas mais complexas, como funções ou vetores. Para grandezas desta natureza, FORM tem instruções especiais para sua declaração.

Existem outros três tipos de variáveis em FORM : Funções, Vetores e Tensores. Cada uma delas, e a sua classificação, veremos a seguir.

Funções

Funções são variáveis mais eficientes porque podem ter uma lista de argumentos. Estes argumentos devem ser separados por vírgulas, e podem ser de qualquer tipo. Quando uma função não tem argumentos, ela se comporta semelhantemente aos símbolos.

Funções são classificadas como comutativas e não-comutativas. As instruções **CFunctions** e **Commuting** declaram funções comutativas, e a instrução **Functions** declara funções não-comutativas.

O exemplo seguinte ilustrará a diferença entre as duas classes de funções.

```
FORM version 3.

*
*Programa FORM explicativo
*
Functions F,G;
CFunctions f,g;
Symbols x,y;
nw stat;
Local exp1 = (F+G)^2;
Local exp2 = (f+g)^2;
print;
.sort
```

```
exp1 =
   F*F + F*G + G*F + G*G;


exp2 =
   f^2 + 2*f*g + g^2;


Local exp3 = F(x)*G(x)+G(x)*F(x)+G(f(x),y);
Local exp4 = f(x)*g(x)+g(x)*f(x) +g((x+1)*(x-1));
print;
.end

exp1 =
   F*F + F*G + G*F + G*G;


exp2 =
   f^2 + 2*f*g + g^2;


exp3 =
   F(x)*G(x) + G(f(x),y) + G(x)*F(x);


exp4 =
   2*f(x)*g(x) + g(-1+x^2);
```

Compare o resultado obtido na expressão **exp1**, com o obtido na **exp2**. Observe que o caracter de comutatividade ou não-comutatividade de uma função tem grande valia quando se faz um cálculo envolvendo, por exemplo, matrizes.

Estas expressões demonstram a possibilidade de utilização de funções com qualquer número e tipo de argumento.

Observe que no programa acima foram feitas algumas especificações para explicar um pouco mais sobre as características do FORM .

- Pode-se escrever comentários utilizando-se no início da linha um *.
- A impressão das mensagens de estatísticas de cálculos podem ser eliminadas pela especificação **Nwrite Statistics**.
- Há uma ordem fixa de declarações em um bloco de programa:

1. Declarações executáveis: Inicia-se com as palavras-chaves Symbol, Function,...
2. Especificações executáveis: inicia-se com as palavras-chaves nwrite, skip, drop, hide,...
3. Definições executáveis: inicia-se com as palavras-chaves local ou global.
4. Controle de saída: print.

A declaração .sort é uma diretiva do FORM para executar um bloco de programa, agrupa o resultado (i.e., traz para dentro da ordem padrão), e prepara para processamento futuro.

Vetores

Vetores são objetos hábeis e velozmente manipulados pelo FORM. O produto escalar e a contração de índices foram eficientemente implementadas.
Vetores são declarados pela instrução **Vectors**, e podem receber índices que devem ser simples números ou índices simbólicos, os quais são declarados pela instrução **Indices**. A dimensão padrão do espaço vetorial no FORM é 4, mas isto pode ser mudado através do comando `dimension`. Veja um exemplo de utilização de vetores:

```
FORM version 3.

NWrite Statistics;
Vectors u, v, x;
Indices i, j, k;
Local t = u + v + x;
Local w = v(1) + u(2) + x(5);
Local y = v(i) + u(i) + x(i);
Local z = v(i)*u(i) + v(j)*u(j) + x(k)*v(k) + v.x;
Print;
.end

t =
   u + v + x;

w =
   u(2) + v(1) + x(5);

y =
   u(i) + v(i) + x(i);

z =
   2*u.v + 2*v.x;
```

A ultima expressão mostra o produto escalar entre vetores. O produto **v(i)*u(i)** é entendido como $\sum_i v_i u_i$, o que significa o produto interno entre estes vetores, **v.u**.
O somatório dos índices é chamado de contração, que é automaticamente realizada. Mas é possível evitar este recurso. Isto é feito no programa abaixo, com a utilização da declaração **Index i=0**, a qual impõe que a dimensão do índice **i** seja zero, evitando a contração.

```
FORM version 3.

NWrite Statistics;
Vector u;
Index i=0;
Local v = u(i)*u(i);
Print;
.end

v =
  u(i)*u(i);
```

Tensores

Tensores são um tipo especial de funções. A única diferença está no fato de que os tensores aceitam como argumento apenas índices ou vetores. Tensores, entretanto, são manipulados mais rápida e eficientemente do que funções.

Tensores também se classificam como comutativos e não-comutativos. A declaração de tensores comutativos é feita pela instrução **Tensors** e a de não-comutativos é feita por **NTensors**.

```
FORM version 3.

NWrite Statistics;
Function F;
Tensors S, X, U;
NTensors A, B;
Vectors u, v;
Indices i, j, k, l=0;
Local T = S(i,j,k)*v(k) + X(i,k)*U(k,j) + F(i,k)*v(k);
Local W = S(i,j,l)*u(l)*v(j);
Local Z = A(i,j)*B(j,k) + B(j,k)*A(i,j);
Print;
.end

T =
  S(i,j,v) + X(i,k)*U(k,j) + F(i,v);

W =
   S(i,v,l)*u(l);

Z =
   A(i,j)*B(j,k) + B(j,k)*A(i,j);
```

Neste programa podemos verificar, também, a contração automática entre vetores e tensores com índices repetidos, e a forma de evitar a contração com a especificação de dimensão zero para o índice. Observe que a contração só acontece quando vetores estão envolvidos.

O FORM possui métodos convenientes para trocar tensores por um produto de componentes de vetores e vice-versa. Isto pode se feito através de `ToVector` e `ToTensor`, respectivamente. Os comandos têm dois argumentos, um tensor e um vetor. A ordem na qual estes argumentos ocorrem é irrelevante. Mudanças de vetores para tensores ocorrem não somente quando as componentes de um vetor são usadas, mas também quando o vetor é contraído com outros vetores ou tensores.

```
FORM version 3.

Tensor t;
Vector u,v;
Indices i,j,k;
nw stat;
Local F1 = v(i)*v(j)*v(k)*v(1);
Local F2 = v;
Local F3 = (u.v)^2*v.v;
ToTensor v,t;
print;
.sort
```

```
F1 =
   t(1,i,j,k);

F2 =
   v;

F3 =
   t(u,u,N1_?,N1_?);

Local F4 = t;
ToVector t,v;
print;
.end

F1 =
   v(1)*v(i)*v(j)*v(k);

F v;

F3 =
   u.v^2*v.v;

F4 =
   1;
```

Construindo Funções Matemáticas

Função Implementada	Significado
abs_	valor absoluto
bernoulli_	número de Bernoulli
binom_	coeficiente binomial
fac_	fatorial
invfac_	inverso do fatorial
max_ , min_	valor máximo e mínimo
mod_	módulo
root_	função raiz
sig_	função sinal
sign_	sinal para inteiros

Funções reservadas	Significado
*acos_, asin_, atan_, atan2_*	*funções trigonométricas inversas*
acosh_, asinh_, atanh	funções hiperbólicas inversas
cos_, sin_, tan_	funções trigonométricas
cosh_, sinh_, tanh_,	*funções hiperbólicas*
li2_	dilogaritmo
lin_	polilogaritmo
ln	*logaritmo natural*
sqrt_	raiz quadrada

Simetrização e Contração de Índices

Propriedade de Simetria	Significado
symmetric	$F(x_1,x_2,\ldots,x_n)=F(x_{\theta(1)},x_{\theta(2)},\ldots,x_{\theta(n)}$ para toda permutação θ
antisymmetric	$F(x_1,x_2,\ldots,x_n)=\mathrm{sgn}(\theta)\,F(x_{\theta(1)},x_{\theta(2)},\ldots,x_{\theta(n)}$ *para toda permutação θ, quando sgn(θ) denota o significado de θ*
cyclic	$F(x_1,x_2,\ldots,x_n)=F(x_{\theta(1)},x_{\theta(2)},\ldots,x_{\theta(n)}$ para toda permutação θ no grupo generado de um ciclo (1 2 3...n)
reyclic	$F(x_1,x_2,\ldots,x_n)=F(x_{\theta(1)},x_{\theta(2)},\ldots,x_{\theta(n)}$ para toda permutação θ no grupo generado de um ciclo (1 2 3...n) e um ciclo (1 n) (2 n-1) (3 n-2) ... $([\frac{n}{2}]\,[\frac{n}{2}+1])$

Matematicamente, tensores podem ser comparados a matrizes de ordem maior ou igual a dois. A grande maioria das propriedades das matrizes, como simetria e anti-simetria, podem ser estendidas aos tensores. E, justamente estas propriedades podem ser aplicadas em um programa FORM .

Considere, primeiro, o seguinte exemplo:

```
FORM version 3.

NWrite Statistics;
Tensors A, B;
Indices i, j, k, l, m, n;
Local T = A(i,j)*B(k,l) + A(j,i)*B(k,l);
Local W = A(i,j)*B(k,l) + A(i,j)*B(l,k);
Local U = A(i,m)*B(m,j) + A(i,n)*B(n,j);
Print;
.end

T =
```

```
   A(i,j)*B(k,l) + A(j,i)*B(k,l);

W =
   A(i,j)*B(k,l) + A(i,j)*B(l,k);

U =
   A(i,m)*B(m,j) + A(i,n)*B(n,j);
```

Suponhamos que o tensor **A** seja simétrico e o tensor **B** seja anti-simétrico. Observe, o resultado que obteremos agora, aplicando as propriedades de simetria de **A** e de anti-simetria de **B**, no exemplo seguinte:

```
FORM version 3.

NWrite Statistics;
Tensors A(Symmetric), B(Antisymmetric);
Indices i, j, k, l, m, n;
Local T = A(i,j)*B(k,l) + A(j,i)*B(k,l);
Local W = A(i,j)*B(k,l) + A(i,j)*B(l,k);
Local U = A(i,m)*B(m,j) + A(i,n)*B(n,j);
Print;
.end

T =
   2*A(i,j)*B(k,l);

W = 0;

U =
    - A(i,m)*B(j,m) - A(i,n)*B(j,n);
```

O comando **Symmetric** é usado para indicar que o tensor que segue é simétrico, e o comando **Antisymmetric** é usado para indicar que o que segue é anti-simétrico.

Dê, agora, atenção a expressão **U** de ambos os exemplos. Embora tenhamos visto que índices repetidos se contraem, representando um somatório, isto não acontece quando as variáveis envolvidas são funções ou tensores. Para forçar que a contração aconteça, utilizamos o comando **Sum**. Vejamos como ele atua nesta expressão **U**.

```
FORM version 3.

NWrite Statistics;
Tensors A(Symmetric), B(Antisymmetric);
Indices i, j, k, l, m, n;
Local T = A(i,j)*B(k,l) + A(j,i)*B(k,l);
Local W = A(i,j)*B(k,l) + A(i,j)*B(l,k);
Local U = A(i,m)*B(m,j) + A(i,n)*B(n,j);
Sum m, n;
Print;
.end

T =
   2*A(i,j)*B(k,l);
```

```
W = 0;

U =
    - 2*A(i,N1_?)*B(j,N1_?);
```

O comando **Sum m, n** diz que os índices **m** e **n** estão sendo somados, ou seja, representam uma contração. Estes são então renomeados com índices internos do FORM , e a simplifição da expressão pode ser feita. O índice contraído gerado pelo FORM é representado por um nome e um símbolo de sublinhado _ .

Padrão de Comparação

Substituição

Nos exemplos anteriores nós vimos a definição de expressões, mas nada foi feito com elas. O FORM não seria muito útil se nós não pudéssemos realizar operações sobre estas expressões. Nesta seção, discutiremos como realizar substituições com variáveis, utilizando-se o comando **id**. Um exemplo:

```
FORM version 3.

Nw Stat;
Symbols x,y,z,h,a,b;
Local expr = 2*x*y + y + x + 1 ;
id x = z + h;
id y = a + b;
print;
.end

expr =
   1 + 2*z*a + 2*z*b + z + 2*h*a + 2*h*b + h + a + b;
```

A substituição **x** → **z** + **h** não representa um loop infinito, pois a instrução **id** atua apenas uma vez sobre cada termo. O comando **id y = a + b** faz a substituição de **y** por **a** + **b**.
Vejamos um novo exemplo:

```
FORM version 3.

Nw Stat;
Symbols x,y,a,b,z;
Local expr = x^2 + x^3 + y^-2 - y^-1 + x - y;
id x = z;
```

```
id y = a*b;
print;
.end

expr =
   y^-2 - y^-1 - a*b + z + z^2 + z^3;
```

Podemos verificar que o FORM não atuou sobre as potências negativas. Vejamos na próxima seção como resolvemos este problema.

Variáveis Wildcard

Wildcards são denotados por variáveis seguidas de interrogação, e representam um objeto simples. Por exemplo, se **x** é um símbolo, então **x?** é um wildcard que representa qualquer símbolo e **x?^2** representa qualquer símbolo elevado ao quadrado. Se **f** é uma função, então **f?** representa qualquer função. Wildcards são ferramentas muito poderosas em manipulação simbólica. São usados como padrões em substituições nas expressões.
Vamos corrigir o programa visto na seção anterior para a substituição de potências negativas:

```
FORM version 3.

Nw Stat;
Symbols x,y,a,b,z,n;
Local expr = x^2 + x^3 + y^-2 - y^-1 + x - y;
id x = z;
id y^n? = a*b;
print;
.end

expr =
    a*b*z + a*b*z^2 + a*b*z^3 - a*b;
```

Existem duas maneiras distintas para a utilização de wildcards em FORM. A primeira delas é colocando o ponto de interrogação à direita da variável. A segunda forma é colocando o ponto de interrogação à esquerda da mesma. Na próxima seção, veremos exemplos destas duas formas, com explicações que diferenciam a sua utilização.

Regras de substituição

Substituição polinomial

Um exemplo de substituição polinomial é o que se segue.

```
FORM version 3.

Nw Stat;
Symbol x,y,z,n;
```

```
Local F = x^2 + y^3 + 1;
id x? = z;
print;
.sort

F =
  1 + z^2 + z^3;

id z^n? = x;
print;
.sort

F =
  3*x;

Local G = F +y^2 + 1;
id x?^n? = z;
print G;
.end

G =
  1 + 4*z;
```

Vamos analisar detalhadamente o programa acima. Como x foi declarado como um símbolo a primeira identificação lê-se como "substitua todos os símbolos por z". Portanto, a expressão $F = x^2 + y^3 + 1$ é substituída por $F = z^2 + z^3 + 1$ e reestruturada como $1 + z^2 + z^3$. A próxima identificação lê-se como "substitua qualquer potência inteira de z por x". O resultado $3x$ torna-se *menos obscuro quando você nota que 1 pode ser descrito como* z^0. *Finalmente nós podemos* construir a expressão $G = 3x + y^2 + 1$ e aplicar a regra de substituição "qualquer símbolo para qualquer potência será substituída por z". Agora o FORM deixa o termo constante intacto devido a ele esperar, no mínimo, um símbolo.

Vejamos agora um exemplo que mostra a utilização dos dois tipos de wildcards:

```
FORM version 3.

Nw Stat;
Function F;
Symbols a,b,c,m,n;
Local expr = F(a+1,b,c+2,a-b,4);
id F(n?,m?,?z) = F(n) + F(m,?z);
print;
.sort

expr =
   F(1 + a) + F(b,2 + c,a - b,4);

repeat;
   id F(n?,m?,?z) = F(n) + F(m,?z);
endrepeat;
print;
.end

expr =
   F(1 + a) + F(2 + c) + F(a - b) + F(b) + F(4);
```

O símbolo **n**? e **m**? representam qualquer elemento que estejam na posição em que eles foram utilizados. Note que **n** e **m** foram declarados como símbolos no programa. No entanto, temos que ?**z** representa nenhum elemento ou vários elementos a partir dele. No programa acima, ele representa os argumentos **c** + **2**, **a-b** e **4**, que são os argumentos restantes. Note que o ?**z** não precisou ser declarado e ele tem que ser repetido no lado direito da expressão, o que não acontece com **n** e **m**.

Números de Fibonacci

No próximos dois exemplos iremos usar regras de substituição para calcular o décimo nono número de Fibonacci F_9.

Note que os números de Fibonacci F_n são recursivamente definidos como

$$F_n = F_{n-1} + F_{n-2}, \quad F_1 = 1, \quad F_2 = 1.$$

```
FORM version 3.

Symbols ultimo, penultimo, indice;
Function F;
Local Fibonacci = F(1,1)*indice^17;
repeat;
  id F(ultimo?,penultimo?)*indice = F(ultimo + penultimo,ultimo);
endrepeat;
id F(ultimo?,penultimo?) = ultimo;
print;
.end

Time = 0.43 sec Generated terms = 1
      Fibonacci Terms in output = 1
                Bytes used      = 10

Fibonacci =
           4181;
```

As declarações que devem ser repetidas são colocadas entre as instruções repeat e endrepeat. As declarações são repetidas até não surtirem mais efeitos sobre a expressão. Nós usamos no programa abaixo a variável **índice** para restringir a repetição.

Uma recursão descendente pode resultar em um programa menos eficiente.

```
FORM version 3.

Symbols n;
Function F;
Local Fibonacci = F(19);
repeat;
  id F(1) = 1;
  id F(2) = 1;
  id F(n?) = F(n-1) + F(n-2);
endrepeat;
```

```
print;
.end

Time = 2.08 sec Generated terms = 4181
      Fibonacci    Terms in output = 1
                       Bytes used = 10

 Fibonacci =
            4181;
```

Derivação de Polinômios

Um exemplo de derivação de polinômios no FORM é o que se segue.

```
FORM version 3.

NWrite Statistics;
Functions dx, sin, cos, ln;
Symbol x;
Local exp = sin(x)*cos(x)*ln(x) + x^2;
Multiply Left dx;
print;
.sort

exp =
   dx*x^2 + dx*sin(x)*cos(x)*ln(x);

 id dx*sin(x) = cos(x) + sin(x)*dx;
 id dx*cos(x) = -sin(x) + cos(x)*dx;
 id dx*ln(x) = 1/x;
 id dx*x^2 = 2*x;
 print;
 .end

exp =
   2*x - sin(x)*sin(x)*ln(x) + sin(x)*cos(x)*x^-1 + cos(x)*cos(x)*ln(x);
```

A instrução **Multiply** atua multiplicando as expressões correntes pelo seu argumento. Pode ter três formas: **Multiply Left**, **Multiply Right** e apenas **Multiply**. No primeiro caso a multiplicação será feita pela esquerda e no segundo pela direita. No terceiro caso, FORM aplicará o que for mais rápido.
Em nosso exemplo, todos os termos da expressão **exp** foram multiplicados a esquerda pela função não-comutativa **dx**.

Tensor de Levi-Civita

Vamos ilustrar as regras sobre repetição de wildcards para tensores pré-construídos, o Tensor de Levi-Civita, ϵ, e o delta de Kronecker, δ, com várias contrações.

```
FORM version 3.
```

```
Nw Stat;
dimension 3;
Tensors eps, delta;
Indices i,j,k,l,m,n;
*
*tres indices repetidos
*
Local F1 = eps(i,j,k) * eps(i,j,k);
id eps(i?,j?,k?) * eps(i?,j?,k?) = 6;
print F1;
.sort

F1 =
   6;

*
* dois indices repetidos
*
Local F2 = eps(i,j,k) * eps(i,j,l);
id eps(i?,j?,k?) * eps(i?,j?,l?) = delta(k,l);
print F2;
.sort

F2 =
   delta(k,l);

*
* um indice repetido
*
Local F3 = eps(i,j,k) * eps(i,l,m);
id eps(i?,j?,k?) * eps(i?,l?,m?) = delta(j,l) * delta(k,m) -
                 delta(j,k) * delta(l,m);
print F3;
.sort

F3 =
    - delta(j,k)*delta(l,m) + delta(j,l)*delta(k,m);

*
* nenhum indice repetido
*
Local F4 = eps(i,j,k) * eps(l,m,n);
id eps(i?,j?,k?) * eps(l?,m?,n?) = eps(i,j,k) * eps(l,m,n);
print;
.end

F1 =
   6;


F2 =
   delta(k,l);


F3 =
    - delta(j,k)*delta(l,m) + delta(j,l)*delta(k,m);
```

```
F4 =
   eps(i,j,k)*eps(l,m,n);
```

Como uma observação, podemos falar sobre o comando **renumber**. Ele tenta uma renumeração dos índices mudos e possui duas opções distintas para utilização:

- **renumber**; (ou **renumber 0**;): Tenta a troca de pares até não acontecer mais nenhum tipo de melhora;
- **renumber 1**; : Tenta todas as permutações. Isto pode ser extremamente lento.

Se como resultado de uma renumeração for obtido o mesmo termo com coeficientes diferentes, o termo é anulado. Por exemplo:

```
e_(p1,p2,N1?)*e_(p1,p2,N2?)*e_(p3,N1?,N2?) -> 0
```

Wildcards e Funções

Para funções são utilizados basicamente quatro tipos de padrões:

1. um ou mais parâmetros de wildcards sobre uma função;
2. wildcards sobre funções;
3. uma combinação das duas acima;
4. wildcards para grupo de argumentos.

Wildcards sobre parâmetros

Exemplificaremos, nesta seção, situações em que se torna necessário operar sobre os argumentos de funções.
É interessante ressaltarmos que uma certa operação pode não ser desejada sobre todas as funções, mas apenas sobre uma específica. Ou, também, não com todos os argumentos, mas apenas com *alguns*.
Ilustraremos, portanto, como selecionar as funções e os parâmentros em cima dos quais queremos trabalhar.
Seja uma transformação, por translação, no sistema de coordenadas: $x \to x + \delta x$ e $y \to y + \delta y$.

```
FORM version 3.

NWrite Statistics;
```

```
CFunctions f, g;
Symbols x, y, z, dx, dy;
Local exp = x + y + z + f(x,y,z) + g(x,z,f(x,y));
id x = x + dx;
id y = y + dy;
print;
.sort

exp =
  x + y + z + dx + dy + f(x,y,z) + g(x,z,f(x,y));

argument;
    id x = x + dx;
    id y = y + dy;
endargument;
print;
.sort

exp =
   x + y + z + dx + dy + f(x + dx,y + dy,z) + g(x + dx,z,f(x,y));

argument;
 argument;
   id x = x + dx;
   id y = y + dy;
 endargument;
endargument;
print;
.end

exp =
   x + y + z + dx + dy + f(x + dx,y + dy,z) + g(x + dx,z,f(x + dx,y + dy));
```

Como havíamos discutido nos exemplos anteriores, uma substituição não atua em cima dos argumentos de uma função. Desta forma, o primeiro resultado apresentado foi o esperado.

Entretanto, quando se faz uma transformação de coordenadas, espera-se que as funções também sofram esta transformação, através de seus parâmetros.

Para realizar operações nos argumentos fazemos uso dos comandos **argument** e **endargument**. Estes dois comandos definem uma estrutura, de forma que, qualquer instrução no seu interior atuará apenas sobre argumentos de funções.

O segundo resultado, embora mais satisfatório, não foi o esperado. Podemos observar que **g** não sofreu a transformação por completo. Isto se deve ao fato de que o segundo argumento de **g** também é uma função que possui outros argumentos. Temos, então, que fazer uso de duas estruturas **argument-endargument** aninhadas para que a transformação seja efetuada por completo.

Consideremos um problema envolvendo álgebra de números complexos : $x + 1 * y$. Como implementaríamos, por exemplo, o cálculo do complexo conjugado? Eis uma proposta de solução:

```
FORM version 3.

NWrite Statistics;
CFunctions conjugado, sqrt;
```

```
Symbols x, I;
Local complex = 3 + 2*I + conjugado(2 + 5*I) + sqrt(5 - I);
argument conjugado;
   id I = -I;
endargument;
print;
.sort

complex =
   3 + 2*I + conjugado(2 - 5*I) + sqrt(5 - I);

id conjugado(x?) = x;
print;
.end

complex =
   5 - 3*I + sqrt(5 - I);
```

Wildcards para Funções

Não há nada especial sobre wildcards em funções ou combinação deste com o tipo anterior de substituição. Nós damos um exemplo que engloba tudo isso.

```
FORM version 3.

Nw Stat;
Symbols x,y,z;
Functions f,g;
Local F1 = f(x) + g(y);
id f?(x) = g(x);
print F1;
.sort

F1 =
   g(x) + g(y);

Local F2 = f(x) + g(y);
id f?(x?) = z;
print F2;
.end

F2 =
   2*z;
```

Substituição para grupos de parâmetros

A substituição em FORM permite que você se refira a um grupo de parâmetros. Vejamos o exemplo abaixo:

```
FORM version 3.
```

```
Nwrite statistics;
Symbols x,y;
Functions f,g;
Local F = f(x,x,x);
id f(?x) = g(0,?x,0);
print;
.end;

F =
   g(0,x,x,x,0);
```

Como visto, com as wildcards podemos especificar mais do que um campo no argumento para substituição.

Vamos demonstrar mais substituições de campos nos argumentos por um exemplo de cálculo tensorial. Consideramos o tensor métrico g, e os tensores Riemann e Ricci denotados por R. Iremos demonstrar como você pode implementar em FORM as contrações $g^{j}_{j}R_{jk} = R_{ik}$ e $g^{ij}R_{ikjl} = R_{kl}$. Será demonstrado também como trabalhar com índices contravariantes (acima) e covariantes (abaixo) em *FORM*.

```
FORM version 3.

Nw Stat;
Tensors g,R;
Indices i,j,k,l,m,n,low,up;
Local T1 = g(i,acima,j,abaixo) * R(j,acima,k,acima);
Local T2 = g(i,abaixo,j,abaixo) * R(i,acima,k,acima,j,acima,l,acima);
id g(?i,acima,?j,abaixo) * R?(?k,?j,acima,?m) = R(?k,?i,acima,?m);
id g(?i, abaixo,?j,abaixo) * R?(?k,?i,acima,?m,?j,acima,?n) = R(?k,?m,?n);
id g(?i, abaixo,?j,abaixo) * R?(?k,?j,acima,?m,?i,acima,?n) = R(?k,?m,?n);
print;
.end

T1 =
    R(i,abaixo,k,abaixo);


T2 =
   R(k,abaixo,l,abaixo);
```

Como você viu, nós simplesmente mantemos os índices anteriores adicionando um índice especial abaixo ou acima, e os distinguimos pelo tipo das identificações. A terceira identificação é adicionada somente para casos gerais onde índices repetidos podem ser alternados.

É claro, a implementação acima de **acima** e **abaixo** dos índices é algo incômodo. Você pode ser *instigado a usar uma notação menor como U(i) e L(j) para índice contravariante i e índice* covariante j, respectivamente. Analise o exemplo abaixo.

```
FORM version 3.

Nw Stat;
Functions g,R,L,U;
Indices i,j,k,l,m,n,low,up;
Local T1 = g(L(i),U(j)) * R(L(j),L(k));
```

```
Local T2 = g(U(i),U(j)) * R(L(i),L(k),L(j),L(l));
*
* modificando as funcoes
*
repeat;
  id R?(?k,L(?i),?m) = R(?k,?i,low,?m);
  id R?(?k,U(?i),?m) = R(?k,?i,up,?m);
endrepeat;
*
* aplicando as regras
*
id g(?i,low,?j,up) * R?(?k,?j,low,?m) = R(?k,?i,low,?m);
id g(?i, up,?j,up) * R?(?k,?i,low,?m,?j,low,?n) = R(?k,?m,?n);
id g(?i, up,?j,up) * R?(?k,?j,low,?m,?i,low,?n) = R(?k,?m,?n);
*
* voltando para a notacao original
*
repeat;
   id R?(?k,i?,low,?m) = R(?k,L(i),?m);
   id R?(?k,i?, up,?m) = R(?k,L(i),?m);
endrepeat;
*
* imprimindo os resultados
*
print;
.end


T1 =
   R(L(i),L(k));

T2 =
    R(L(k),L(l));
```

Ao longo desta apostila, mostraremos outros caminhos para manusear índices contravariantes e covariantes que não conduzem à duas representações de um mesmo objeto matemático.

Condições sobre Wildcards

As substituições que temos visto até agora envolvem wildcards que poderiam corresponder a qualquer variável de tipo básico. Além disso, nenhuma restrição sobre as regras de substituição tinham sido feitas. Nesta sessão, devemos ver como obter mais controle sobre as wildcards e substituições.

Conjuntos e Substituições

Um dos tipos de variáveis em FORM é "set" (que significa conjunto). Abaixo possui um exemplo que explica isto:

```
FORM version 3.
```

```
Symbol a1,a2;
Set a:a1,a2;
Local F = a[1] + a[2];
print;
.end

Time =       0.16 sec    Generated terms =          2
             F           Terms in output =          2
                         Bytes used      =         32

F =
   a1 + a2;
```

Como você vê, o FORM assume que conjuntos e vetores comecem com índice 1. Além disso, os elementos dos conjuntos devem ser do mesmo tipo.
Conjuntos são mais usados em substituições.

```
FORM version 3.

Symbols a,b,c,x;
Local F = a + b + c;
id x?{b,c} = 3;
print;
.end

Time =       0.49 sec    Generated terms =          3
         F               Terms in output =          2
                         Bytes used      =         26


F =
  6 + a;
```

O x? significaria "qualquer símbolo", mas no nosso exemplo nós o limitamos para o conjunto b, c.
Você pode também excluir símbolos por ?! .

```
FORM version 3.

Symbols a,b,c,x;
Local F = a + b + c;
id x?!{b,c} = 3;
print;
.end


Time =       0.49 sec    Generated terms =          3
         F               Terms in output =          3
                          Bytes used     =         40

 F =
    3 + b + c;
```

Conjuntos também são chamados de vetores porque seus elementos podem ser referenciados por números. Vejamos como isto e feito:

```
FORM version 3.

Symbols a1,a2,b1,b2,x,n;
Set a : a1,a2;
Set b : b1,b2;
Local F = a[1] + a[2];
id x?a[n] = b[n];
print;
.end

Time =       0.00 sec    Generated terms =          2
           F             Terms in output =          2
                         Bytes used      =         32

F =
   b1 + b2;
```

No exemplo acima, x deve pertencer ao conjunto a, e n torna-se o número de elementos no conjunto que x corresponde. O mesmo número é usado para o lado direito da identidade. Então, não pode ser feita operações aritméticas com n.
Existe um atalho para mudar nomes dos elementos dos conjuntos.

```
FORM version 3.

Symbols a1,a2,b1,b2,x,n;
Set a : a1,a2;
Set b : b1,b2;
Local F = a[1] + a[2];
id x?a?b = x;
print;
.end

Time =       0.00 sec    Generated terms =          2
           F             Terms in output =          2
                         Bytes used      =         32

F =
  b1 + b2;
```

O comando **id** lê-se como segue: x deve pertencer ao conjunto a, e no lado direito cada ocorrência de x poderá ser substituída pelo elemento correspondente do conjunto b.
Os conjuntos pré-construídos em FORM estão listados abaixo. Como todos são objetos pré-construídos, eles terminam com um underscore.

Conjunto	Significa um conjunto de
*int_*	*inteiros*
pos_	inteiros positivos
pos0_	inteiros não negativos
*neg_*	*inteiros negativos*
neg0_	inteiros não positivos
symbol_	símbolos formais
*fixed_*	*índices fixados*
index_	todos os índices

Com estes conjuntos pré-construídos podemos restringir wildcards para infinitos conjuntos, como no próximo exemplo.

```
FORM version 3.

Nw Stat;
Symbols x,n;
Local F = sum_( n, -3, 3, x^n );
print;
.sort

F =
  1 + x^-3 + x^-2 + x^-1 + x + x^2 + x^3;

id x^n?pos_ = 0;
print;
.end

F =
  1 + x^-3 + x^-2 + x^-1;
```

Aqui, nós substituímos todas as potências positivas de x por zero.

Restrições sobre as Substituições

Quando criamos um par de identidades, pode haver um conflito de qual regra será aplicada primeiro. No FORM , podemos influenciar a aplicabilidade de uma identificação por opções. Uma das opções é select . Ele é seguido pelos nomes de um ou mais conjuntos de símbolos. A regra de substituição somente poderá ser aplicada se após a correspondência do lado esquerdo nenhum elemento dos conjuntos mencionados estiver sobrando. Um exemplo explicará isto melhor.

```
FORM version 3.

Symbols a,b,c;
Set bcSet: b,c;
Local F = a*b*c;
id select bcSet a*b = b^2;
```

```
id select bcSet a*b*c = b^2*c^2;
print;
.end

Time =       0.98 sec    Generated terms =          1
                F        Terms in output =          1
                         Bytes used      =         22

F =
   b^2*c^2;
```

No exemplo acima, a primeira regra de substituição não é aplicada porque após corresponder o termo a*b no produto a*b*c o elemento c do conjunto {b,c} está sobrando. No segundo caso, após correspondermos o produto a*b*c em a*b*c, temos que todos os elementos do conjunto {b,c} estão sendo utilizados. Portanto, a regra de substituição coincide e foi aplicada.

Vejamos um outro exemplo:

```
FORM version 3.

Symbols a,b,c,d;
Indices i,j,k,l;
CFunctions f,g,h,ff;
Local expr = (f(i,a)*b + g(j,b)*c + h(k,c)*a);
id select,{a,b}, d? = ff(d);
print;
.end

Time =       0.10 sec    Generated terms =          3
               expr      Terms in output =          3
                         Bytes used      =         94

expr =
   f(i,a)*b + g(j,b)*c + h(k,c)*ff(a);
```

A primeira característica deste programa está no fato de que o conjunto foi implicitamente declarado. Neste caso, o conjunto não possui um nome e não é necessário a utilização do comando **Set**, mas os seus elementos devem obrigatoriamente estar entre chaves. Assim, termos {**a**,**b**} significa que estamos declarando um conjunto de componentes **a** e **b**, que não possui um nome.

Como já vimos, a wildcard **d**? representa qualquer símbolo. Observe os termos. O comando **id** tentará a substituição no primeiro termo de **expr**. No entanto, quando procurar dentro do argumento da função, verá que existe um elemento do conjunto, no caso **a**. Logo, a substituição não é feita. O mesmo se dá com o segundo termo de **expr**. Já com o terceiro termo, a substituição acontece porque foi encontrado o símbolo **a** e observando o argumento da função vimos que não possui nenhum elemento do conjunto.

Existem mais opções para restrição de substituição em FORM. Abaixo, colocamô-las em forma de tabela e ilustramos algumas delas.

Opção	Significado
once	*correspondência simples*
only	correspondência exata
multi	correspondência simples com multiplicidade
many	*correspondência múltiplas*
disorder	substituição em álgebra não-comutativa

```
FORM version 3.

Nw Stat;
Symbols a,b,x,y,z;
Local f0 = (x+y)^4;
print f0;
.sort

f0 =
   4*x*y^3 + 6*x^2*y^2 + 4*x^3*y + x^4 + y^4;

Local f1 = (x+y)^4;
id once x = z;
print f1;
.sort

f1 =
   6*x*y^2*z + 4*x^2*y*z + x^3*z + 4*y^3*z + y^4;

Local f2 = (x+y)^4;
id x*y = z;
print f2;
.sort

f2 =
   4*x^2*z + x^4 + 4*y^2*z + y^4 + 6*z^2;

Local f3 = (x+y)^4;
id only x*y = z;
print f3;
.sort

f3 =
   4*x*y^3 + 6*x^2*y^2 + 4*x^3*y + x^4 + y^4;

Local f4 = (a+b)^2 * (x+y)^2;     print f4;     .sort

f4 =
   4*a*b*x*y + 2*a*b*x^2 + 2*a*b*y^2 + 2*a^2*x*y + a^2*x^2 + a^2*y^2 + 2*
   b^2*x*y + b^2*x^2 + b^2*y^2;

id multi x?*y? = z;
print f4;
.end

f4 =
   2*a*y*z + 2*b*y*z + 4*x*y*z + 2*x^2*z + 2*y^2*z + 4*z^2;
```

```
Local f4 = (a+b)^2 * (x+y)^2;
print f4;
.sort

 f4 =
    4*a*b*x*y + 2*a*b*x^2 + 2*a*b*y^2 + 2*a^2*x*y + a^2*x^2 + a^2*y^2 + 2*
    b^2*x*y + b^2*x^2 + b^2*y^2;

id multi x?*y? = z;
print f4;
.end

f4 =
   2*a*y*z + 2*b*y*z + 4*x*y*z + 2*x^2*z + 2*y^2*z + 4*z^2;
```

A regra normal para comparação de termos no FORM é que o termos é pego tantas vezes quanto possível antes de ser inserido do lado direito (a mais importante exceção é com potências de uma wildcard). Nenhum esforço adicional de comparação de termos é feito. Na primeira identificação, no exemplo acima, a opção once anula o comportamento geral do sistema: somente uma única comparação de termos é feita. À propósito, o padrão é many . As próximas duas identificações no *exemplo são as substituições* $xy \to z$ *no polinômio* $(x+y)^4$. *A opção only implica que a* comparação do termo deve ser exata. Devido a isto, não há termos em $(x+y)^4$ com potências iguais de x e y, e, portanto, não haverá nenhuma substituição. Finalmente, a opção multi é usada: *ela significa uma única comparação com multiplicidade. No exemplo dado, ele significa* que quando ele é aplicado ao termo $4abxy$, somente ab é pego com uma multiciplidade. O outro produto, xy, não é tocado.

Devemos descrever a opção disorder com uma exemplo da álgebra de Dirac em dimensão 4. As matrizes de Dirac $\gamma_1,\gamma_2,\gamma_3$, e γ_4 tem as propriedades

$$\gamma_i\gamma_j + \gamma_j\gamma_i = 0, \quad \text{se } i \neq j$$
$$\gamma_i^2 = 1$$

Para $\gamma_5 = \gamma_1\gamma_2\gamma_3\gamma_4$, temos as duas equações seguintes

$$\gamma_5\gamma_i + \gamma_i\gamma_5 = 0, \quad \text{para todo } i$$
$$\gamma_5^2 = 1$$

É simples confirmar estas regras no FORM .

```
FORM version 3.

Nw Stat;
Functions g4,g3,g2,g1,g,h;
```

```
Local [g5] = g4 * g3 * g2 * g1;
Local [g5^2] = [g5]^2;
Local [g1*g5+g5*g1] = g1 * [g5] + [g5] * g1;
.sort
repeat;
id g?*g? = 1;
id disorder g? * h?= - h * g;
endrepeat;
print;
.end

[g5] =
   g1*g2*g3*g4;

[g5^2] =
   1;

[g1*g5+g5*g1] = 0;
```

Você pode ter esperado regras de substituição como:

```
id g2 * g1 = -g1 * g2;
id g3 * g1 = -g1 * g3;
id g3 * g2 = -g2 * g3;
id g4 * g1 = -g1 * g4;
id g4 * g2 = -g2 * g4;
id g4 * g3 = -g3 * g4 ;
```

Mas isto é um tanto quanto incômodo. É muito mais fácil confiar na ordem interna das funções e ter somente uma identificação como

```
id g? * h? = - h*g
```

Mas quando você repete cada regra de substituição, você poderia entrar dentro de um loop infinito. Para evitar isto, a opção disorder foi introduzido no FORM . Com esta opção, a identificação é aplicada quando a ordem normal dos termos na comparação mudaria a ordem das funções se eles forem comutativos.

Índices Contravariantes e Covariantes

Agora que conhecemos conjuntos, podemos analisar outra implementação no FORM de índices contravariantes e covariantes. Consideramos o mesmo exemplo da Seção de substituição para grupos de parâmetros. Índices covariantes ou índices inferiores como a e b são escritos como ia e ib , onde o i representa "inferior". Similarmente, contravariante ou índices superiores como a e b são escritos sa e sb , onde o s representa "superior". Definimos dois conjuntos , a saber, LU e UL , que permite-nos checar se um índice aparece duas vezes mas de tipos opostos. Isto torna fácil definir as regras de substituição para "levantar" e "abaixar" índices com o tensor métrico, como o exemplo abaixo mostra.

```
FORM version 3.

Nw Stat;
Tensors g,R;
Indices si,ii,sj,ij,sk,ik,sl,il;
Set UL: si,ii,sj,ij,sk,ik,sl,il;
Set LU: ii,si,ij,sj,ik,sk,il,sl;
Indices i,j,k,l;
Symbols m,n,z;
Local T1 = g(ii,sj)*R(ij,ik);
Local T2 = g(si,sj)*R(ii,ik,ij,il);
id g(i?UL[m],j?UL[n]) * R?(?m,k?LU[m],?n,l?LU[n],?z) = R(?m,?n,?z);
id g(i?UL[m],j?UL[n]) * R?(?m,k?LU[n],?n,l?LU[m],?z) = R(?m,?n,?z);
id g(?i,j?UL[n]) * R?(?m,k?LU[n],?n) = R(?m,?i,?n);
print;
.end

T1 =
   R(ii,ik);

T2 =
   R(ik,il);
```

Note que a ordem das regras de substituição torna-se importante: se a última tinha colocado o índice na parte superior, o segundo tensor poderia não estar completamente contraído. Teríamos obtido como resultado intermediário e final R(ii,ik,si,il) .

Em nosso exemplo, o melhor conjunto de regras é o que segue.

```
FORM version 3.

Nw Stat;
Tensors g,R;
Indices si,ii,sj,ij,sk,ik,sl,il;
Set UL: si,ii,sj,ij,sk,ik,sl,il;
Set LU: ii,si,ij,sj,ik,sk,il,sl;
Indices i,j,k,l;
Symbols m,n,z;
Local T1 = g(ii,sj)*R(ij,sk);
Local T2 = g(si,sj)*R(ii,ik,ij,il);
id g(?i,j?UL[n]) * R?(?m,k?LU[n],?n) = R(?m,?i,?n);
id R?(?m,i?UL[n],?n,j?LU[n],?z) = R(?m,?n,?z);
print;
.end

T1 =
   R(ii,ik);

T2 =
   R(ik,il);
```

Limitações na Substituição

Coeficientes e Wildcards

Podemos trabalhar com coeficientes utilizando a declaração collect. Com esta declaração você pode colocar dados que estão entre colchetes (tanto pela declaração bracket ou antibracket) dentro de uma função regular. Teríamos, então:

```
FORM version 3.

Nw Stat;
Symbol a,x,y;
CFunctions indice, sqrt;
Local F = 2 * sqrt(a);
antibracket a;
print;
.sort

F =
    + sqrt(a) * ( 2 );

collect indice;
print;
.sort

F =
   indice(2)*sqrt(a);

id indice(x?)*sqrt(y?) = sqrt(x^2*y);
print;
.end

F =
   sqrt(4*a);
```

Somatório de Wildcards

Outro tipo de comparação de termos que não é permitido em FORM é f(x?+y?), e suas variações. Por exemplo, a aditividade de uma função f não pode ser especificada como se segue:

```
FORM version 3.

Nw Stat;
Symbols x,y,z;
CFunction f;
Local expr = f(x + y + z);
id f(x? + y? + z?) = f(x) + f(y) + f(z);
print;
.end

expr =
```

```
f(x + y + z);
```

A razão para a não permissão deste tipo de substituição em argumentos de funçaõ é a eficiência: o termo f(x1? + x2? + ... + xn?) tem em geral $n!$ possíveis atribuições para as substituições. Eficiência é também a razão para a não-implementação de comparação de termos para denominadores compostos e para potências com expoentes não-inteiros.
No entanto, o problema acima de aditividade de uma função pode ser implementada através do seguinte truque. De fato, nós implementamos mais: a linearidade de uma função.

```
FORM version 3.

Nw Stat;
Vectors x,y,z;
Indice i;
CFunction f;
Local expr = f(2*x + y + z);
id f(i?) = f(i);
print;
.end

expr =
   2*f(x) + f(y) + f(z);
```

A explicação é a que segue: a wildcard na função é um índice. Se o FORM possui um vetor como argumento em uma função, ele assume que ele está lá devido a contração dos índices. Em outras palavras, ele assume que a função é linear neste argumento.

Tabelas

Tabelas são como matrizes, preenchidas com dados. As tabelas podem ter parâmetros formais do mesmo modo que funções podem estar em declarações **id**. Também como funções, temos duas opções para a declaração de tabelas em FORM :

- **NTable**: Declara a tabela automaticamente como uma função não-comutativa.
- **CTable**: Declara que a tabela automaticamente como uma função comutativa. O comando **Table** é o padrão para **CTable**.

Existem dois tipos de declarações de tabelas. Uma tabela comum possui a sintaxe

```
Table [check][strict|relax] name(low:hi,...,argumentos);
```

Em primeiro lugar, existem palavras-chaves que controla um conjunto limitado de erros. A tabela "nome" deve ter sido declarada anteriormente como uma função. Isto permite ao usuário definir

tabelas de objetos não-comutativos. Então, dentro dos campos de argumentos existe primeiro os intervalos dos índices da tabela. Deve haver no mínimo em cada intervalo. Um intervalo é indicado por dois números, separados por dois-pontos (:). Os vários intervalos estão separados tanto por espaço-em-branco tanto por vírgulas. Note que o tamanho da tabela será o produto dos tamanhos de cada intervalo. Após o intervalo poderão existir parâmetros formais extras. A sintaxe para eles é idêntica à sintaxe para os argumentos de funções no lado esquerdo de uma declaração **id**. O significado das palavras-chaves é:

check

Este parâmetro controla o que está acontecendo com elementos fora do intervalo da tabela. Com o parâmetro **check** um elemento pode resultar em um erro durante execução. Sem este parâmetro, elementos podem aparecer na saída.

strict

Todos os elementos da tabela devem ser definidos com declarações **fill** antes da primeira instrução executável no módulo.

relax

Se nem todos os elementos da tabela tiverem sido definidos e durante a execução um elemento dentro do intervalo da tabela é encontrado , nenhum erro será gerado, mas o elemento está sobrando e aparecerá na saída.

A declaração **fill** é usada para preecher a tabela com elementos. Normalmente, cada elemento necessita de uma declaração **fill**. Para inserir um elemento em uma tabela devemos nos referir somente às coordenadas dos elementos. Não devemos repetir os argumentos formais. A sintaxe é:

```
fill nome(num1,...,numn) = expressao;
```

No caso de uma tabela unidimensional deve haver, é claro, somente um número dentro do intervalo da declaração **table**. Todas as declarações **fill** devem estar antes da primeira declaração executável no módulo. A razão para esta regra é que o conteúdo das tabelas está armazenada dentro do buffer do compilador. Ele previne a ocorrência de buracos e ineficiências na tabela.

O segundo tipo de tabela é o tipo **sparse**. Esta é uma tabela cujo intervalos dos argumentos causariam a existência de muitos elementos. Mesmo que não utilizemos a opção **strict**, faz com que não tenhamos de definir todos os elementos, há ainda espaço alocado para cada elemento em potencial na tabela. Isto pode causar sérios problemas. Portanto, existe um tipo de tabelas na qual os elementos estão armazenados de uma maneira, que não é necessariamente especificar as limitações de matrizes. O preço a pagar é uma perfomance mais lenta. Descobrir se um elemento foi ou não definido involve testar todos os elementos definidos (= "preenchidos"). A sintaxe para este tipo de tabela é:

```
Table sparse [check] name(numargs, regular arguments);
```

Neste caso, não especificamos intervalos, mas somente a dimensão da tabela. O argumento extra é o mesmo de antes. A antiga definição da opção **strict** é inútil agora. A ação padrão é que nada é checado, nem durante o tempo de execução. Isto é equivalente a opção **relax** no primeiro tipo de tabela. Havendo problemas no programa, quando o elemento da tabela estiver indefinido durante o tempo de execução, podemos ativar a opção **check**. As opções **strict** e **relax** são ignoradas. Um exemplo é que segue:

```
FORM version 3.

Nw Stat;
CFunction x;
Table x(-2:1);
Fill x(-2) = 10;
Fill x(-1) = 5;
Fill x(-0) = -3;
Fill x(1) = 8;
Symbols a,b,n;
Local F = b/a + b^2/a^2 +  b^3/a^3 + a/b;
id a^n? = x(n);
print;
.end

 F =
    8*b^-1 + 5*b + 10*b^2 + x(-3)*b^3;
```

Note que o conteúdo da tabela foi substituído automaticamente. O elemento -3 permaneceu porque não foi definido na tabela anteriormente. Outro exemplo:

```
FORM version 3.

Nw Stat;
CFunction t;
Symbols x,y,a,b,c,ep;
Table t(1:2,x?);
Fill t(1) = 1 + ep*x + ep^2*x^2 + ep^3*x^3;
Fill t(2) = 1 + ep*x + ep^2*x^2/2 + ep^3*x^3/6;
.global
Local F = a*t(1,y) + a^2*t(2,b+c);
Bracket a;
Print +s;
.end

F =
   + a * (
   + 1
   + y*ep
   + y^2*ep^2
   + y^3*ep^3
```

```
)
+ a^2 * (
+ 1
+ b*c*ep^2
+ 1/2*b*c^2*ep^3
+ b*ep
+ 1/2*b^2*c*ep^3
+ 1/2*b^2*ep^2
+ 1/6*b^3*ep^3
+ c*ep
+ 1/2*c^2*ep^2
+ 1/6*c^3*ep^3
);
```

Derivado do comando **fill**, o comando **FillExpression** transfere uma expressão para uma tabela. A sintaxe é a seguinte:

```
FillExpression tab = expr(x1,...,xn)
```

Neste exemplo, **tab** deve ser declarado como uma tabela com **n** índices. **Expr** é uma expressão que foi anteriormente agrupada nas variáveis **x1**,...,**xn**. As potências de **x1**,...,**xn** referem-se aos elementos da tabela e o conteúdo de cada agrupamento é colocado no elemento da tabela correspondente. Agrupamentos que não possuem um elemento da tabela correpondente a eles são ignorados.

Temos, também, o comando **PrintTable**. Ele escreve o conteúdo de uma tabela na tela, em um arquivo .log, em ambos ou somente em um arquivo especificado. Sua sintaxe é:

```
PrintTable [+f][+s]nome_da_tabela [>[>]arquivo];
```

As opções **+f** e **+s** trabalham como uma declaração de impressão regular.

Se um redirecionamento para um arquivo foi especificado, a opção **+f** não terá efeito, pois a saída é feita somente para um arquivo.

Com > **nome_do_arquivo**, o antigo conteúdo do arquivo será reescrito (se houver algum).

Com >> **nome_do_arquivo**, a tabela será anexada. Isto permite a escrita de mais de uma tabela no mesmo arquivo.

Na escrita, as declarações **Fill** apropriadas são escritas para tornar o arquivo útil como uma nova entrada de dados para o FORM .

Tabelas podem podem ser usados em regras de recursão:

```
FORM version 3.

Nw Stat;
#define MAXTAB "6"
CFunction t;
Symbols x,y;
```

```
Table t(1:'MAXTAB',x?);
Fill t(1) = 1 + x;
#do i = 2, 'MAXTAB'
Fill t('i') = ('i'+x)*t({'i'-1},x);
#enddo
Local F = t(6,y);
Print;
.end

F =
   720 + 1764*y + 1624*y^2 + 735*y^3 + 175*y^4 + 21*y^5 + y^6;
```

Tabelas do tipo **sparse** são, freqüentemente, utilizadas quando a tabela contém simetrias. Isto é mostrado no exemplo abaixo:

```
FORM version 3.

Nw Stat;
Function tt;
Table sparse tt(4);
Fill tt(1,1,1,2) = 10;
Local F = tt(2,1,1,1) + tt(1,2,1,1) + tt(1,1,2,1) + tt(1,1,1,2);
Print;
.sort

F =
   10 + tt(1,1,2,1) + tt(1,2,1,1) + tt(2,1,1,1);

Symmetrize tt;
Print;
.end

F =
   40;
```

Funções Especiais

Já citamos algumas delas no **Capítulo 1** e estudamos, neste Capítulo, as funções pré-definidas **d_** e **e_**. Agora, iremos citar outras funções especiais que são muito úteis na manipulação de expressões no FORM .

- **delta_(a)**: Resultado igual a 1 se $a = 0$; 0 se $a \neq 0$ mas numérico.
- **theta_(a)**: Resultado igual a 1 se $a \geq 0$; 0 se $a < 0$. No entanto, **theta_(x^2)** não é simplificado.

- **fac_(n)**: $n!$.

- **sum_(i,m,n,expressão)**: $\Sigma_{i=m}^{n}$ *expressão*.

- **bernouilli_** (incluindo o $n!$): Fornece os coeficientes de Bernouilli.

- **sig_(x)** = Resultado igual a 1 se $x \geq 0$; -1 se $x < 0$ e retorna **sig_(x)** se **x** for simbólico.

- **abs_(x)**: Fornece como resultado o módulo de **x** ($|x|$) se **x** for numérico, retorna **abs_(x)** se **x** for simbólico.

- **root_(n,x)**: Se **n** é um positivo inteiro e **x** é um número racional e a enésima raiz de **x** também um número racional **y**, então **root_(n,x)** será substituído por **y**. Em todos os outros caso, no entanto, ele não será tocado. Se existir uma escolha em relação ao sinal, esta função sempre escolherá o valor positivo. Portanto: **root_(2,9)** => **+3**. Principalmente para uso interno, ela será extremamente útil para simplificações de expressões (como em **(a+b)^3**, por exemplo). No entanto, ela não tenta separar raízes como **root_(2,8)** = **2*root_(2,2)** porque, para isso, necessitaríamos adicionar uma teoria numérica sobre fatorização.

O FORM possui ainda outros nomes que estão reservados. São eles:

- Para **símbolos**: **pi_** e **coeff_**.

O comando reservado **coeff_**, é substituído imediatamente pelo coeficiente do termo. No entanto, seu uso é restrito para variáveis $, que serão explicadas quando falarmos sobre o pré-processador.

- Para **CFunctions**: **sqrt_ ln_, sin_, cos_, tan_, asin_, acos_, atan_, atan2_, sinh_, cosh_, tanh_, asinh_, cosh_, atanh_, li2_,li_**.

Comandos Especiais

Esta seção abrange comandos não disponíveis em FORM 2.1. São eles:

- **SplitArg**, que ignora termos de acordo com suas opções.

1. a. **splitarg,(termo), outras_opções**: Ignora somente o termo no argumento que é um múltiplo numérico do termo especificado entre parênteses.
   b. **splitarg,((termo)), outras_opções**: Ignora todos os termos que contém "termo" especificado. Portanto, ignora também múltiplos não -numéricos. Wildcards não são permitidas.

- **Normalize**, que trabalha com fatores de normalização:

1. a. **Normalize, (0),...** : Faz o mesmo que **Normalize,...** mas não multiplica os coeficientes globais com o fator de normalização. Portanto:

Normalize, **f1**; muda **f1(2*x+3*y)** em **2*f1(x+3/2*y)**, mas
Normalize, **f1**; muda **f1(2*x+3*y)** em **f1(x+3/2*y)**.

- **FactArg**, que trabalha com os coeficientes de uma expressão:

1. a. **FactArg,(0)**: Elimina coeficientes (como **Normalize**, **(0)**,...)
   a. **FactArg,(1)**: Coloca em evidência o coeficiente e sinal (separadamente).
   b. **FactArg,(-1)**: Coloca em evidência o coeficiente com sinal.

No caso de **(1)** e **(-1)**, um coeficiente de valor **1** é também colocado em evidência para tornar o cálculo mais uniforme.

- **FunPowers**, que controla a impressão de potências de funções:

1. a. **FunPowers nofunpowers**: As funções são impressas da maneira que apareceriam no FORM 2.1, o que significa **f(a)*f(a)*f(a)**.
   b. **FunPowers commutingonly**: É o padrão. As funções comutativas são impressas com potências *(como* **f(a)^3***)* e as funções não-comutativas permanecem escritas normalmente.
   c. **FunPowers allfunpowers**: Ambas as funções são escritas com potências.

Note que tensores nunca são impressos como potências, e que qualquer escrita da parte principal do comando escolhido é suficiente para ativá-lo.

- **AutoDeclare**, que faz declarações genéricas:

Utilização:

```
AutoDeclare Vector v;
AutoDeclare Symbol x(:10), y(:5);
AutoDeclare Symbol x1(:9);
```

Todas as variáveis não declaradas que comecem com o caracter **v** serão automaticamente declaradas como vetores. Todas as variáveis não-declaradas que comecem com a letra **x** possuirão, no máximo, potências de 10, exceto para aquelas que comecem com **x1**, que possuem a restrição de potências até 9.

- Substituição de variáveis $ do lado esquerdo, com o comando **count_**:

```
$a = count_(x,1)
id F($a) = ...\newline{}
```

A variável $ somente será substituída quand os termos forem comparados. Se na substituição

ocorrer de um coeficiente necessitar ser desenvolvido; somente o sinal é retido. se nenhum termo ou mais do que um termo necessitar disto -> problema.

O comando **count** trabalha somente com as variáveis $, como em

```
$a = 3*count_(x,1,y,2)-1+x;
```

Neste exemplo, a contagem é feita a partir do termo que é o principal dado. Nem todas as opções de contagem regular são válidas. Notamos a diferença quando trabalhamos com vetores.

Manipulação de Expressões

Expressões Globais

No FORM, quase não há diferença entre expressões locais e globais. No entanto, esta última leva algumas vantagens sobre a primeira.

A primeira vantagem que podemos citar está no fato de que elas podem receber uma lista de argumentos. Vejamos um exemplo:

```
FORM version 3.

Nw Stat;
Symbols a, b;
CFunctions sin, cos;

* Expressoes Globais

Global SIN(a,b) = sin(a)*cos(b) + sin(b)*cos(a);
Global COS(a,b) = cos(a)*cos(b) - sin(a)*sin(b);
Print;
.store


SIN(a,b) =
    sin(a)*cos(b) + sin(b)*cos(a);

COS(a,b) = - sin(a)*sin(b) + cos(a)*cos(b);


Symbols a, b, c;
Local trig = SIN(a,c) + SIN(b,b) + COS(c,b);
Print +s;
.end


trig =
   sin(a)*cos(c) - sin(b)*sin(c) + 2*sin(b)*cos(b)
   + sin(c)*cos(a) + cos(b)*cos(c);
```

Como visto acima, uma expressão global é declarada pela instrução **global**; no entanto, só podemos utilizar os efeitos de suas vantagens após a diretiva .**store**, quando, então, elas são armazenadas em um arquivo temporário.

A diretiva .**store** realiza as seguintes operações:

- indica o término de um módulo;
- remove todas as expressões locais;
- armazena todas as expressões globais;
- desfaz todas as definições de variáveis feitas até então.

Após esta diretiva, não podemos mais modificar as expressões globais. No entanto, podemos utilizá-las na construção de novas expressões.

Para evitar a perda das definições das variáveis, podemos utilizar a diretiva .**global**.

```
FORM version 3.

Nw Stat;
Symbols x, n;
.global
Global exp(x) = sum_(n,0,8,x^n/fac_(n));
Print;
.store


exp(x) =
   1 + x + 1/2*x^2 + 1/6*x^3 + 1/24*x^4 + 1/120*x^5 + 1/720*x^6 +
   + 1/5040*x^7 + 1/40320*x^8;


Local F = exp(x);
id x = 1;
Print;
.end


F =
    109601/40320;
```

Observe que, desta vez, não tivemos de redeclarar o símbolo **x** depois da utilização de .**store**, como aconteceu no exemplo anterior.

Salvando e Carregando Expressões

A segunda vantagem das expressões globais sobre as expressões locais, já citada rapidamente no início deste texto, está no fato de que as primeiras podem ser salvas em um arquivo temporário, e depois podem ser utilizadas em outros programas.

Para salvá-las, utilizamos a instrução **save**. Sua sintaxe é muito simples, consistindo apenas do nome do arquivo e dos nomes das expressões. Se nenhum nome de expressão for mencionado, todas as expressões do programa serão salvas.

```
FORM version 3.

Nw Stat;
Symbol x;
CFunction sqrt;
Global e(x) = 1 + x + x^2/2 + x^3/6 + x^4/24;
Global abs(x) = sqrt(x^2);
Print;
.store


e(x) =
   1 + x + 1/2*x^2 + 1/6*x^3 + 1/24*x^4;

abs(x) =
   sqrt(x^2);


Save exemplo.sav;
.end
```

Note que é somente após a diretiva .**store** que as expressões globais são salvas, o que faz sentido, já que é através desta diretiva que as expressões globais são armazenadas.

Podemos carregar as expressões salvas com a instrução **load**. Sua sintaxe é semelhante a do comando **save**. Quando as expressões são carregadas, podem ser utilizadas normalmente, como se tivessem sido definidas dentro do próprio programa.

```
FORM version 3.

Nw Stat;
Load exemplo.sav;
e loaded
abs loaded
Symbols a, b;
Local exp = abs(a) + e(b);
Print;
.end


exp =
   1 + b + 1/2*b^2 + 1/6*b^3 + 1/24*b^4 + sqrt(a^2);
```

Abandonando expressões

A cada novo módulo de nosso programa, podemos definir novas expressões, as quais podem ser, ou não, construídas a partir das anteriores. Um exemplo:

```
FORM version 3.

Symbols a, b;
NWrite Statistics;
Local E1 = a+b;
Print;
.sort

E1 =
   a + b;

Local E2 = E1*E1;
Print;
.sort

E1 =
   a + b;

E2 =
   2*a*b + a^2 + b^2;

Local E3 = E2*E1;
Print;
.end

E1 =
   a + b;

E2 =
   2*a*b + a^2 + b^2;

E3 =
   3*a*b^2 + 3*a^2*b + a^3 + b^3;
```

Ao definirmos uma expressão, estamos definindo-a para o resto do programa. Embora o programa seja dividido em vários módulos, e a cada módulo uma nova expressão seja definida, as expressões definidas em módulos anteriores continuam a existir.

Contudo, podemos, a cada módulo, excluir as expressões que não desejamos mais. Isto é feito com comando **drop** seguido do nome das expressões. Vejamos como fazê-lo:

```
FORM version 3.

Symbols x, y;
NWrite Statistics;
Local F1 = x+y;
Print;
.sort
```

```
F1 =
   x + y;

Drop F1;
Local F2 = F1^2;
Print;
.sort

F2 =
   2*x*y + x^2 + y^2;

Drop F2;
Local F3 = F2^2;
Print;
.end

F3 =
   4*x*y^3 + 6*x^2*y^2 + 4*x^3*y + x^4 + y^4;
```

Observe que, dentro do módulo em que a expressão é excluída, pode-se ainda utilizá-la, mas, no módulo seguinte, ela não existirá mais.

Mas, muitas vezes, não queremos excluir uma expressão, e sim, deixar de aplicar um ou mais comandos sobre ela. Para fazê-lo, utilizamos o comando **skip** seguido do nome das expressões. Este comando irá "esconder" as expressões definidas da aplicação dos comandos do módulo corrente, sem excluí-las. Um exemplo:

```
FORM version 3.

Symbol a, b, x, y;
NWrite Statistics;
Local exp1 = (a+b)^2;
Local exp2 = (a+b)^4;
id a = x;
Print;
.sort

exp1 =
   2*b*x + b^2 + x^2;

exp2 =
   4*b*x^3 + 6*b^2*x^2 + 4*b^3*x + b^4 + x^4;

Skip exp1;
id b = y;
Print;
.sort

exp2 =
   4*x*y^3 + 6*x^2*y^2 + 4*x^3*y + x^4 + y^4;

id x = a;
Print;
.end

exp1 =
```

```
   2*a*b + a^2 + b^2;

exp2 =
   4*a*y^3 + 6*a^2*y^2 + 4*a^3*y + a^4 + y^4;
```

Utilizando arquivos separados

Certos programas podem tornar-se demasiadamente grandes ou, às vezes, um trabalho pode vir a exigir inúmeros programas, nos quais se repetem os mesmos comandos.

Para organizar melhor nosso trabalho, poderíamos escrever em um arquivo separado o conjunto das declarações ou dos comandos que repetem-se várias vezes.

Suponhamos que temos um arquivo, chamado **derivada.txt**, com o seguinte conteúdo:

```
id dx*sin(x?) = cos(x) * dx*x;
id dx*cos(x?) = -sin(x) * dx*x;
id dx*x^n?= n*x^(n-1);
```

Podemos incluí-lo em um outro programa com o uso do comando **#include**:

```
Nw Stat;
Functions sin, cos, dx;
Symbols n, x;
Local exp = sin(2*x) + cos(3*x) + x^4;
Multiply Left dx;
#include derivada.txt
print;
.end
```

E, após executá-lo, teríamos o seguinte resultado:

```
FORM version 3

Nw Stat;
Functions sin, cos, dx;
Symbols n, x;
Local exp = sin(2*x) + cos(3*x) + x^4;
Multiply Left dx;
#include derivada.txt
id dx*sin(x?) = cos(x) * dx*x;
id dx*cos(x?) = -sin(x) * dx*x;
id dx*x^n?= n*x^(n-1);
Print;
.end

exp =
   4*x^3 - 3*sin(3*x) + 2*cos(2*x);
```

Manipulação de coeficientes

Para agrupar coeficientes de uma dada expressão, o FORM possui o comando **bracket**. Vejamos como ele atua:

```
FORM version 3.

Nw Stat;
Symbols a, b, c, x;
Local poly = (a + 3)*x^2 + (b + 2)*x + c + 7;
Print;
.sort


poly =
   7 + a*x^2 + b*x + c + 2*x + 3*x^2;


Bracket x;
Print;
.end


poly =
    + x * ( 2 + b )+ x^2 * ( 3 + a )+ 7 + c;
```

O comando **Bracket** precisa possuir um ou mais argumentos. Este argumento é a variável cujo os coeficientes são os elementos que desejamos colocar em evidência.

Podemos, agora, construir uma nova expressão, fazendo uso destes coeficientes. Vamos tentar, então, através de um exemplo, encontrar as raízes do polinômio abaixo:

$$(a+3)x^2 + (b+2)x + c + 7$$

```
FORM version 3.

Nw Stat;
Symbols a, b, c, x;
CFunction sqrt;
Local polinomio = (a + 3)*x^2 + (b + 2)*x + c + 7;
Bracket x;
Print;
.sort


polinomio =
    + x * ( 2 + b )+ x^2 * ( 3 + a )+ 7 + c;


Local A = polinomio[x^2];
Local B = polinomio[x];
Local C = polinomio[1];
```

```
Print A, B, C;
.sort


A =
   3 + a;

B =
   2 + b;

C =
   7 + c;

Local R1 = ( -B + sqrt(B^2 - 4*A*C) ) / ( 2*A );
Local R2 = ( -B - sqrt(B^2 - 4*A*C) ) / ( 2*A );
Print R1, R2;
.end


R1 =
    - 2/(6 + 2*a) - 1/(6 + 2*a)*b + 1/(6 + 2*a)*sqrt( - 80 -
    4*a*c - 28*a+ 4*b + b^2 - 12*c);

R2 =
    - 2/(6 + 2*a) - 1/(6 + 2*a)*b - 1/(6 + 2*a)*sqrt( - 80 -
    4*a*c - 28*a + 4*b + b^2 - 12*c);
```

Note que no programa acima, o comando **print** aceita a lista das expressões que desejamos que sejam impressas como argumentos.

Quando os coeficientes de uma variável são colocados em evidência através do comando **bracket**, eles podem possuir um nome, que no exemplo acima é o nome **polinomio** com a respectiva variável entre colchetes. Assim, **polinomio[1]** é o coeficiente de **x^0**, **polinomio[x]** é o coeficiente de **x^1** , e assim sucessivamente. Note que este recurso só pôde ser utilizado no módulo seguinte ao uso do comando **bracket**.

Estruturas do pré-processador

Macro substituição

Com a instrução **#define** definimos uma variável do pré-processador, e atribuímos a ela um valor. Sua sintaxe é:

```
#define nome "valor"
```

Vejamos um exemplo:

```
FORM version 3.
Nw Stat;
```

```
#define MAX "3"
Symbols x, n;
Local exp = sum_(n,0,'MAX',x^n);
Print;
.end

exp =
   1 + x + x^2 + x^3;
```

O nome da variável pode ser como o nome das variáveis do FORM . Para o valor da variável, que deve ser colocado entre aspas, é permitida qualquer palavra.
Com o comando **#redefine**, ao contrário do comando **#define**, redefinimos uma variável do pré-processador. Se a variável ainda não existir, o FORM criará esta variável automaticamente. Sua sintaxe é:

```
#redefine nome "valor"
```

Quando utilizadas dentro do programa, as variáveis do pré-processador devem ser colocadas entre aspas simples.
Sua versatilidade permite que as variáveis sejam concatenadas:

```
FORM version 3.

Nw Stat;
#define fatorial "fac_"
#define N "4"
Local f'N' = 'fatorial'('N");
Print;
.end

f4 =
   24;
```

No entanto, uma melhor notação para a concatenação de variáveis do pré-processador está em se utilizar

```
'nome'
```

ao invés de

```
'nome'
```

como utilizado anteriormente. Com esta nova notação, é possível utilizar estruturas do tipo

```
'nome'num"
```

para ter `'num'` substituído por um valor (por exemplo, **1**), e o pré-processador procurar automaticamente por `'nome1'`. Vejamos novamente o exemplo utilizando-se a nova notação

```
FORM version 3.

Nw Stat;
#define fatorial "fac_"
#define N "4"
Local f'N' = 'fatorial'('N');
print;
.end

f4 =
    24;
```

A primeira (e mais externa) ' pode ainda ser substituída por ', pois isto não causará ambiguidades.

Com o comando #**show**, podemos ver as atribuições atuais das variáveis do pré-processador.

Além de realizar substituições, o pré-processador pode fazer operações extras, como citadas abaixo

- ( ) { }
- + − * / % ^ (significados normais)
- & | (**e** ou **ou** lógicos)
- ! ^% ^/ (para fatorial, log2 e raiz quadrada).

Todos os cálculos são realizados sobre inteiros com sinal de 32 bits (ou mais se a máquina for de 64 bits) e a operação escolhida, para ser executada, deve ser encerrada por chaves:

```
FORM version 3.

Nw Stat;
#define MAX "3"
Symbols x, n;
Local poly'MAX' = sum_(n,0,'MAX',x^n);
Local poly{'MAX'+1} = poly'MAX' + x^{'MAX'+1};
Print;
.end

poly3 =
   1 + x + x^2 + x^3;

poly4 =
   1 + x + x^2 + x^3 + x^4;
```

O pré-processador possui também o operador ..., que auxilia na construção de comandos que anteriormente necessitavam de loops #**do**. Temos dois tipos de utilização:

a. **str#1**[?]**01**...**02str**[?]

Aqui, **str** é uma palavra que se inicia com um caracter alfabético. Ambas as palavras devem ser idênticas. **#1** é um número e **#2** é um segundo núumero que não precisam ser, necessariamente, os mesmos. O ? é opcional, mas, se estiver presente, deve estar em ambos os locais. **01** e **02** são dois operadores. As chances para o par são: (´ ´), (++),(--)(- +),(**),(//). Outros pares não são permitidos. O efeito desta construção é equivalente a

```
str{#1}[?]01str{#1+1}02str{#1+2}01 etc ate #2 se #2 > #1
```

e

```
str{#1}[?]01str{#1-1}02str{#1-2}01 etc ate #2 se #2 < #1
```

b. A utilização mais geral

<termo1>01...02<termo2>

Os < > servem como um tipo de parênteses que delimitam os termos. Os termos só são permitidos se possuírem partes numéricas diferentes. No entanto, a condição de diferença destes termos é a de que devem ter a mesma diferença numérica em valor absoluto. Como exemplo de construções equivalentes, temos

```
<f3(i1,i5)>*...*<f6(i4,i2)> ->

              f3(i1,i5)*f4(i2,i4)*f5(i3,i3)*f6(i4,i2)
```

Decisão

A estrutura de decisão desvia o fluxo de processamento do programa. Sua construção é feita com:

```
#if condicao
   [instrucoes
#else
    instrucoes]
#endif
```

ou

```
#if condicao
```

```
    instrucoes
[#elseif
    instrucoes]
[#else
    instrucoes]
#endif
```

A parte entre colchetes é opcional.

Se a condição for verdadeira, então o primeiro bloco de instruções é processado. Se a condição for falsa, então a instrução **#else** é usada, e o segundo bloco de instruções é processado.

Um exemplo:

```
FORM version 3.

Nw Stat;
#define N "-3"
Symbols n, x;
Local S =
#if 'N' < 0
     sum_(n,'N',0,x^n);
#else
    sum_(n,0,'N',x^n);
#endif
Print;
.end

S =
    1 + x^-3 + x^-2 + x^-1;
```

Na segunda opção, se a primeira condição é verdadeira, o primeiro bloco de instruções é processado. No entanto, se for falsa, entramos em um outro bloco de condições com a instrução **#elseif**. Esta opção somente é utilizada quando temos várias condições e vários blocos de instruções diferentes.

Podemos, também, compor condições (como em `#if('PAR'>=5) && ('PAR'!=10))`. A comparação não precisa ser mais, necessariamente, somente com números, mas pode ser executada com qualquer palavra, como em `#if 'PAR' == NOVAVERSAO`.

Se a palavra contiver espaços em branco, estes devem ser preenchidos com o caracter \. O mesmo deve ser feito com os operadores que possam interferir com o trabalho da instrução `#if`.

Além da construção `#if`, existe uma construção `#switch`, como a que existe na linguagem C. Sua sintaxe é:

```
#switcht objeto
#case valor
#break
#case outrovalor
#break
#default
#endswitch
```

Se um #**break** for omitido, pode-se entrar no próximo caso. O objeto e o valor podem ser qualquer palavra ou número.

Quanto aos loops, sabemos que são definidos como contrações cíclicas de índices somáveis em funções (anti) simétricas de tensores. Um exemplo está a seguir, que representa um loop de três vértices.

```
f(i1,i2)*f(i2,i3)*f(i3,i1)
```

Como um loop é trivial, não necessitamos de uma declaração especial. No entanto, para casos mais complicados, temos os comandos **ReplaceLoop** e **FindLoop**. Outro exemplo:

```
ReplaceLoop,f,arguments = 3, loopsize =3, outfun = ff
```

Se **f** é uma função antisimétrica, o resultado seria

```
-ff(i4,i5,i6)*f(i4,i7,i8)*...
```

Portanto, é mais aconselhável que **ff** seja ciclosimétrico, já que o FORM não recomenda o início a partir de uma função específica.
O primeiro argumento deve ser o nome de uma função ou tensor que é tanto simétrico quanto anti-simétrico. Também temos opções:

a. **arguments = number**

Somente ocorrências da função onde um certo número de argumentos são considerados.

b. **loopsize = number**

Somente loops onde este tamanho for correspondido.

c. **loopsize = all**

Todos os tamanhos são considerados em ordem crescente.

d. loopsize < number

Todos os tamanhos de loop menores do que o número são considerados.

e. outfun = name

Nome de uma saída de função ou tensor. Deve ser especificado.

f. include = name

Nome de um índice somável que deve ser incluído no loop. Isto é útil após a modificação que inclui este índice. Note que todas as estruturas devem ser analisadas desse modo. Isto é opcional.
A ordem destas "opções" é irrelevante, mas se a opção relevante estiver faltando haverá uma mensagem de erro.

Similarmente, temos a opção **FindLoop**, como em

```
if (FindLoop(f,arg = 3,loop < 5)) discard;
```

Aqui a opção **outfun** é ilegal porque não existe substituição.
A função **FindLoop** fornece o valor **1** se tal loop existir, e **0** caso contrário.
Existindo conflito, devido a função loop ser uma função com argumentos que não se encaixam dentro do tensor (isto também se aplica a índices), e **outfun** for um tensor, estas ocorrências da função loop que causariam conflito não são consideradas.

Repetição

A estrutura de repetição implementa um loop. Instruções no seu interior são processadas repetidas vezes. Sua sintaxe é:

```
#do nome = inicio, fim [,passo]
    instrucoes
#enddo
```

A variável **nome** recebe o valor inicial de **início**, e será incrementada em **passo**. Se o valor do

incremento não é especificado, é assumido o valor 1. O laço se encerrará quando **nome** for maior que **fim**.
Um exemplo:

```
FORM version 3.

Nw Stat;
Symbol x;
Local F1 =
#do N = 0, 4
  + x^'N'
#enddo;
Local F2 =
#do N = 0, 4, 2
   + x^'N'
#enddo;
Print;
.end


F1 =
   1 + x + x^2 + x^3 + x^4;

F2 =
   1 + x^2 + x^4;
```

Numa outra forma de definir loops, a variável não é incrementada. Seus valores são enumerados, separados por barras verticais:

```
#do nome = { valor1 | valor2 | ... | valorn }
    instrucoes
#enddo
```

Um exemplo:

```
FORM version 3.

Nw Stat;
CFunction f;
Symbol x;
Local F =
#do N = { 1 | 2 | 4 | 8 }
    + x^'N'
#enddo ;
Local G =
#do N = { 1 | 1\,x | x | x^2+1 }
    + f('N')
#enddo ;
Print;
.end

F =
  x + x^2 + x^4 + x^8;
```

```
G =
  f(1 + x^2) + f(x) + f(1) + f(1,x);
```

Procedimentos

A definição de procedimentos em FORM segue à seguinte sintaxe:

```
#procedure nome ( arg1, arg2, ... argn )
    instrucoes
#endprocedure
```

Na definição do procedimento, os argumentos são separados por vírgulas. As variáveis que fazem o papel de argumento devem ser usadas como variáveis do pré-processador.

A chamada a um procedimento é feita pelo comando #call . Sua sintaxe é:

```
#call nome { arg1 | arg2 | ... | argn }
```

Na chamada ao procedimento, os argumentos são separados por barras verticais, e podem ser de qualquer tipo. Argumentos em procedimentos são opcionais. Podem ser omitidos se não forem necessários.

Vejamos um exemplo:

```
FORM version 3.

Nw Stat;
#procedure diff(x,dx)
Multiply 'dx';
  id 'dx'*'x'^n? = n*x^(n-1);
#endprocedure
Symbols n, x, y, dx, dy;
Local exp =  4*x^3*y^2 + 2*x^3*y + x^2 + y;
#call diff{y|dy}
#call diff{x|dx}
Print;
.end

exp =
    6*x^2 + 32*x^3;
```

Procedimentos podem ser definidos em arquivos separados. Para isto o arquivo deve ter o mesmo nome do procedimento, e a extensão .**prc**.

Considere que temos um arquivo de nome **diff.prc** com o seguinte conteúdo:

```
#procedure diff(x,dx)
Multiply 'dx';
 id 'dx'*'x'^n? = n*x^(n-1);
#endprocedure
```

Examinemos agora o exemplo anterior sem a declaração do procedimento **diff**:

```
FORM version 3.

Symbols n, x, y, dx, dy;
Local exp =  4*x^3*y^2 + 2*x^3*y + x^2 + y;
#call diff{y|dy}
#call diff{x|dx}
Print;
.end

exp =
   24*x^2*y + 6*x^2;
```

Quando o FORM procura pelo procedimento e não o encontra no programa, irá procurar um arquivo com o mesmo nome do procedimento e com extensão .**prc**. Neste caso, **diff.prc**.

Estruturas de controle

Diferente das estruturas do pré-processador, que apenas realizam substituições, as estruturas de controle são avaliadas na execução do programa.

Decisão

A função desta estrutura é condicionar a atuação dos comandos no seu interior. Tem a seguinte sintaxe:

```
if (condicao);
   instrucoes
[else;
   instrucoes]
endif;
```

Neste caso, a condição é testada para cada termo das expressões. Vejamos como funciona:

```
FORM version 3.

Nw Stat;
Symbols a, b, x;
Local F = a + a*x + a*x^2;
if (match(x)=1);
   id a = 1;
else;
  id a = 2;
endif;
```

```
Print;
.end

F =
   2 + x + 2*x^2;
```

A função **match** retorna o número de ocorrências de seu argumento, no termo que esta sendo avaliado. A condição **match(x)=1** é verdadeira se no termo avaliado houver apenas uma ocorrência de **x**. Observe que a estrutura é executada para cada termo de **F**.
Outro exemplo:

```
FORM version 3.

Nw Stat;
Symbols n, x;
Local F = sum_(n,0,5,x^n);
if (match(x)>3);
discard;
endif;
Print;
.end

F =
   1 + x + x^2 + x^3;
```

O comando **discard** simplesmente discarta o termo em questão.
Além da função **match**, existe uma outra que pode ser usada como condição. Um exemplo:

```
FORM version 3.

Nw Stat;
Symbols a, b, c, d, x;
Local exp = a + b*x + c*x^2 + d*x^3;
if (count(x,1) < 3);
  multiply x;
endif;
Print;
.end

exp =
   a*x + b*x^2 + c*x^3 + d*x^3;
```

A função **count** conta o número de ocorrências de seu primeiro argumento, e multiplica pelo segundo. Assim, apenas as três primeiras parcelas de **exp** atendem à condição.
Outro exemplo:

```
FORM version 3.

Nw Stat;
Symbols a, b, c, d, x;
Local exp = a + b*x + c*x^2 + d*x^3;
if (count(x,2)=4);
```

```
multiply 2;
endif;
Print;
.end

exp =
   a + b*x + 2*c*x^2 + d*x^3;
```

Note que, neste exemplo, apenas o terceiro termo de **exp** atende à condição.
Um último recurso que pode ser usado como condição na decisão é o coeficiente do termo avaliado:

```
FORM version 3.

Nw Stat;
CFunctions f, g;
Symbol n;
Local exp1 = sum_(n,0,3,f(n)/fac_(n));
Local exp2 = sum_(n,0,3,g(n)/fac_(n));
Local exp = exp1*exp2;
if (coefficient < 1/10);
    discard;
endif;
Print;
.end

exp1 =
   f(0) + f(1) + 1/2*f(2) + 1/6*f(3);

exp2 =
   g(0) + g(1) + 1/2*g(2) + 1/6*g(3);

exp =
   f(0)*g(0) + f(0)*g(1) + 1/2*f(0)*g(2) +
   + 1/6*f(0)*g(3) + f(1)*g(0) + f(1)*g(1)+
   + 1/2*f(1)*g(2) + 1/6*f(1)*g(3) + 1/2*f(2)*g(0) +
   + 1/2*f(2)*g(1) + 1/4*f(2)*g(2)
   + 1/6*f(3)*g(0) + 1/6*f(3)*g(1);
```

O comando **Coefficient** representa o valor do coeficiente do termo que está sendo avaliado. Neste exemplo, se seu coeficiente é menor que 1/10, então o termo é discartado.

Repetição

Ao contrário do pré-processador, onde o laço é controlado por uma variável contador, a repetição é governada por uma condição. O laço é encerrado quando esta condição passa a ser falsa. Lembrando, novamente, que o estrutura é aplicada a cada termo da expressão. Sua sintaxe é:

```
while (condicao);
instrucoes
endwhile;
```

```
FORM version 3.
Nw Stat;
Symbols a, b, c, x;
Local exp = a*x^5 + b*x^6 + c*x^2;
while (match(x)>3);
id x^2 = x;
endwhile;
Print;
.end

exp =
    a*x^3 + b*x^3 + c*x^2;
```

A wildcard é aplicada sobre cada termo, até que a condição **match(x)>3** se torne falsa. Note que o terceiro termo da expressão não é atingido.
Uma outra forma de criar loops é através de saltos:

```
FORM version 3.

Nw Stat;
Symbol x;
Local F = x^10;
Label 1;
Multiply 1/x;
if (match(x) > 2);
  goto 1;
endif;
Print;
.end

F =
   x^2;
```

A instrução **label** declara um rótulo, um número, que será o alvo para o salto que é realizado pela instrução **goto**.

Variáveis-$

As variáveis-$ são uma ligação entre o processo algébrico e o pré-processador, e sua função é guardar informações por um pequeno período de tempo. Por exemplo, se tivéssemos a instrução abaixo executada,

```
id x ^n$a = f(n);
```

o processador colocaria a potência de **x** (portanto o valor de **n**) na variável-$ **a**, toda vez que o termo passar pela substituição. No resto do programa é permitida a utilização daquele valor. Por exemplo, se após o comando acima tivéssemos

```
id g($a)= algo;
```

o argumento de **g** seria substituído por um valor que seria idêntico a potência de **x** da declaração anterior.

Após o módulo ter sido executado por completo, o valor da variável-$ é o valor da potência do último termo que foi executado. Note que em uma expressão vazia **$a** não temos qualquer valor.

Após um .**sort** podemos novamente usar este valor como **$a**, ou também como

```
'$a'
```

pois o pré-processador substitui este valor. Um exemplo:

```
#assign
$b=-100;
if (count(x,1)>$b) id x^n$b=x^n;
.sort
```

Após este módulo, a potência mais alta de **x** está em **$b**. Dessa forma, podemos, por exemplo, controlar um número de recursões via pré-processador, como abaixo:

```
#do i =  '$b',1,-1
```

No exemplo abaixo

```
I m1,m2,m3,m4,n1,n2;
L F = g_(3,m1)*g_(3,m2)*g_(n2,m3)*g_(n2,m4);
L G = gi_(5);
repeat;
      if (match(g_(m1$n,m2?)) || match(gi_(m1$n));
          Tracen,$n;
      endif;
endrepeat;
Print;
.end
```

temos um "índice" na matriz gamma e calculamos o traço sobre as matrizes com a ajuda deste índice. Deste modo, não precisamos conhecer a rotina de traço na qual está o índice. O bloco de

comando

```
repeat;
      if (match(g_(m1$n,m2?)) || match(gi_(m1$n)) Tracen,$n;
endrepeat;
```

faz todos os traços. O comando **gi_** deve estar presente, pois ele é similar a 1. Sobre o **1**, não podemos fazer normalmente nenhuma substituição ou nenhuma determinação de multiplicidade. Portanto, somente podemos compará-lo quando ele estiver explícito. O mesmo se aplica para **g5_**, **g6_**, e **g7_**, com a exceção de que em um traço n-dimensional eles não podem aparecer. Para um traço quadri-dimensional devemos adicionar três comparações.

Além desta característica, um dos objetivos das variáveis-$ é auxiliar na manipulação de expressões. Devido a isto, tal variável pode conter um número, um índice, um símbolo, um termo completo, uma seqüência de argumentos de uma função ou mesmo uma expressão completa. No entanto, ao utilizá-la, certifique-se que está utilizando tipos compatíveis de dados, pois, caso contrário, surgirá uma mensagem de erro.

Para operações com variáveis-$ temos, além de outros já citados nos capítulos anteriores, o comando **Inside**. Sua sintaxe é:

```
Inside,$var1,$var2,...,$varn;
    Declaracoes
EndInside;
```

Este comando trabalha como o comando **Argument**. Todas as variáveis-$ que são mencionadas serão processadas. A ação de processamento está descrita abaixo:

1. a primeira variável será copiada para dentro de um buffer;
2. termo por termo o conteúdo deste buffer é tratado;
3. o resultado é ordenado e,
4. finalmente o conteúdo da primeira variável é sobrescrito;
5. a próxima variável é tratada, e assim por diante.

Cuidado com redefinições de qualquer destas variáveis dentro do local de declarações, pois isto surte efeito na ordem das variáveis. Do mesmo modo, utilizar qualquer destas variáveis dentro deste local também não é aconselhado.

Tome extremo cuidado também com o manuseio de expressões excessivamente grandes, pois as variáveis-$ não são armazenadas em disco. Com isso, pode haver muita atividade de *swap*.

Exercícios sobre o FORM 3.0

1. Considere que a e b são vetores e que T é um tensor. Utilize o FORM para calcular as expressões dadas abaixo:

a. $(a_i b_j + a_j b_i)(a_i b_j + a_j b_i)$

b. $(a_i b_j + a_j b_i)(a_i b_j - a_j b_i)$

c. $(T_{ij} + T_{ji})(a_i a_j)$

d. $(T_{ij} - T_{ji})(a_i a_j)$

2. Considere que A e T são tensores. Utilize o FORM para demonstrar as identidades abaixo:

a. $(T_{ij} + T_{ji})(A_{ij} + A_{ji}) = 2T_{ij}A_{ij} + 2T_{ij}A_{ji}$

b. $(T_{ij} + T_{ji})(A_{ij} - A_{ji}) = 0$

c. $(T_{ij} + T_{ji} + A_{ij} - A_{ji})^2 = (T_{ij} + T_{ji})^2 + (A_{ij} - A_{ji})^2$

3. Suponha que A seja um tensor antisimétrico e que S seja um tensor simétrico. Demonstre as igualdades abaixo.

a. $A_{ij} - A_{ji} + S_{ij} + S_{ji} = 2A_{ij} + 2S_{ij}$

b. $A_{ij}S_{ij} + A_{ji}S_{ji} = 0$

c. $A_{ij}a_j - A_{ji}a_j = 2A_{ij}a_j$

4. Escreva um programa FORM que ordene qualquer lista de números.

5. Escreva um programa FORM que elimine os n primeiros elementos de uma lista.

6. Escreva um programa FORM que calcule as duas raízes de qualquer polinômio do segundo grau.

7. Implemente, em FORM, a função mod(a,b) do Pascal, sabendo que:

$$\mathrm{mod}(5,2) = 1 \text{ (resto de 5/2)}$$

8. Implemente, em FORM, a função div(a,b) do Pascal, sabendo que:

$$div(7,3) = 2 \text{ (quociente de 7/3)}$$

9. Desenvolva o procedimento sumargs, que tem por objetivo somar os argumentos da função passada por parâmetro.

10. Escreva um programa FORM para que, dado um polinômio do quarto grau e uma de suas raízes,

$$a_1x^4 + b_1x^3 + c_1x^2 + d_1x + e_1 = 0,\ r_1 = dado$$

este seja capaz de encontrar o polinômio de terceiro grau,

$$a_2x^3 + b_2x^2 + c_2x + d_2$$

tal que

$$(a_2x^3 + b_2x^2 + c_2x + d_2)(x - r_1) = a_1x^4 + b_1x^3 + c_1x^2 + d_1x + e_1$$

11. Escreva um programa FORM que faça a união de dois conjuntos.

12. Escreva um programa FORM que faça a interseção de dois conjuntos.

13. Desenvolva o procedimento sqr em FORM.

14. Escreva um programa que resolva um sistema de equações da forma:

$$\begin{aligned} a_1x + b_1y + c_1 &= 0 \\ a_2x + b_2y + c_2 &= 0 \end{aligned}$$

15. Escreva um programa FORM que converta números em binário para números em decimal.

16. Escreva um programa FORM para que, dada a seguinte transformação

$$\begin{cases} x = (\cos\alpha)x' - sin(\alpha)y' \\ y = (\sin\alpha)x' + \cos(\alpha)y' \\ z = z' \end{cases}$$

possamos mostrar que $F(x,y,z) = F(x',y',z')$.

17. Levando-se em conta a Série de Taylor, calcule e^x até a terceira potência e calcule um valor aproximado para $e^{0.1}$.

18. Sabendo-se que a derivada de $f(n)$ é $f(n+1)$, calcule a derivada de $f(2)f(3)f(4)$.

Resolução dos Exercícios

1. Considere que a e b são vetores e que T é um tensor. Utilize o FORM para calcular as expressões dadas abaixo:

a. $(a_ib_j + a_jb_i)(a_ib_j + a_jb_i)$

b. $(a_ib_j + a_jb_i)(a_ib_j - a_jb_i)$

c. $(T_{ij} + T_{ji})(a_ia_j)$

d. $(T_{ij} - T_{ji})(a_ia_j)$

```
FORM version 3.

Nw Stat;
Tensor T;
Vectors a,b;
Indices i,j;
```

```
Local exp = (a(i)*b(j) + a(j)*b(i))*(a(i)*b(j) + a(j)*b(i));
Local exp1 = (a(i)*b(j) + a(j)*b(i))*(a(i)*b(j) - a(j)*b(i));
Local exp2 = (T(i,j) + T(j,i))*(a(i)*a(j));
Local exp3 = (T(i,j) - T(j,i))*(a(i)*a(j));
print;
.end

exp =
   2*a.a*b.b + 2*a.b^2;

exp1 = 0;


exp2 =
   2*T(a,a);


exp3 = 0;
```

2. Considere que A e T são tensores. Utilize o FORM para demonstrar as identidades abaixo:

a. $(T_{ij} + T_{ji})(A_{ij} + A_{ji}) = 2T_{ij}A_{ij} + 2T_{ij}A_{ji}$

b. $(T_{ij} + T_{ji})(A_{ij} - A_{ji}) = 0$

c. $(T_{ij} + T_{ji} + A_{ij} - A_{ji})^2 = (T_{ij} + T_{ji})^2 + (A_{ij} - A_{ji})^2$

```
FORM version 3.

Nw Stat;
Tensors A,T;
Indices i,j;
Local exp = (T(i,j) + T(j,i))*(A(i,j) + A(j,i));
Local exp1 = (T(i,j) + T(j,i))*(A(i,j) - A(j,i));
Local exp2 = (T(i,j) + T(j,i) + A(i,j) - A(j,i))^2;
sum i,j;
print;
.end


exp =
   2*A(N1_?,N2_?)*T(N1_?,N2_?) + 2*A(N1_?,N2_?)*T(N2_?,N1_?);

exp1 = 0;

exp2 =
   2*A(N1_?,N2_?)*A(N1_?,N2_?) - 2*A(N1_?,N2_?)*A(N2_?,N1_?) + 2*T(N1_?,
   N2_?)*T(N1_?,N2_?) + 2*T(N1_?,N2_?)*T(N2_?,N1_?);
```

3. Suponha que A seja um tensor antisimétrico e que S seja um tensor simétrico. Demonstre as igualdades abaixo.

a. $A_{ij} - A_{ji} + S_{ij} + S_{ji} = 2A_{ij} + 2S_{ij}$

b. $A_{ij}S_{ij} + A_{ji}S_{ji} = 0$

c. $A_{ij}a_j - A_{ji}a_j = 2A_{ij}a_j$

```
FORM version 3.

Nw Stat;
```

```
Tensor A,S,a;
Indices i,j;
Local exp = A(i,j) - A(j,i) + S(i,j) + S(j,i);
Local exp1 = A(i,j)*S(i,j) + A(j,i)*S(j,i);
Local exp2 = A(i,j)*a(j) - A(j,i)*a(j);
Symmetrize S;
Antisymmetrize A;
print;
.end


exp =
   2*A(i,j) + 2*S(i,j);

exp1 = 0;


exp2 =
   2*A(i,j)*a(j);
\newline
```

4. Escreva um programa FORM que ordene qualquer lista de números.

```
FORM version 3.

Nw Stat;
Cfunction lista;
Symbols x,y;
Local exp=lista(1,3,2,6,5,4);
repeat id lista(x?,y?,?m) = lista(x)*lista(y,?m);
repeat id lista(x?,?m)*lista(y?,?n) = lista(x,?m,y,?n);
print;
.end

exp =
  lista(1,2,3,4,5,6);
```

5. Escreva um programa FORM que elimine os n primeiros elementos de uma lista.

```
FORM version 3.

Nw Stat;
Cfunction elimina,lista;
Symbols x,y;
Local exp1=elimina(2,lista(5,2,7,11,15,1));
Local exp2=elimina(5,lista(2,8,7,1,5,21,3,-1));
repeat;
 id elimina(0,lista(?m)) =  lista(?m);
 id elimina(x?,lista(y?,?m)) = elimina(x-1,lista(?m));
endrepeat;
print;
.end

exp1 =
    lista(7,11,15,1);
```

```
exp2 =
   lista(21,3,-1);
```

6. Escreva um programa FORM que calcule as duas raízes de qualquer polinômio do segundo grau.

```
FORM version 3.

Nw Stat;
Symbols x,a,b,c;
cfunction sqrt;
local poly = (a)*x^2+(b)*x+c;
bracket x;
print;
.sort

poly =

   + x * ( b )

   + x^2 * ( a )

   + c;


Local A = poly[x^2];
Local B = poly[x];
Local C = poly[1];
Local R1 = (-B+sqrt(B^2-4*A*C))/(2*A);
Local R2 = (-B-sqrt(B^2-4*A*C))/(2*A);
print;
.end

poly =
   x*b + x^2*a + c;

A =
   a;

B =
   b;

C =
   c;

R1 =
    - 1/2*a^-1*b + 1/2*sqrt( - 4*a*c + b^2)*a^-1;

R2 =
    - 1/2*a^-1*b - 1/2*sqrt( - 4*a*c + b^2)*a^-1;
```

7. Implemente, em FORM, a função mod(a,b) do Pascal, sabendo que:

$$\mathrm{mod}(5,2) = 1 \text{ (resto de 5/2)}$$

```
FORM version 3.
```

```
Nw Stat;
Function mod,resp;
symbols a,b;
local exp=resp(mod(5,2));
repeat;
id resp(mod(a?pos_,b?pos_)) = resp(mod(a-b,b));
endrepeat;
id resp(mod(a?,b?)) = mod(a+b);
print;
.end

exp =
   mod(1);
```

8. Implemente, em FORM, a função div(a,b) do Pascal, sabendo que:

$$div(7,3) = 2 \text{ (quociente de 7/3)}$$

```
FORM version 3.

Nw Stat;
Function div;
symbols a,b,c;
local exp=div(7,3,0);
repeat;
id div(a?pos_,b?pos_,c?) = div(a-b,b,c+1);
endrepeat;
id div(a?,b?,c?) = div(c-1);
print;
.end

exp =
   div(2);
```

9. Desenvolva o procedimento sumargs, que tem por objetivo somar os argumentos da função passada por parâmetro.

```
FORM version 3.

Nw Stat;
#procedure sumargs(x)
  id 'x'(n?,m?,?i) = 'x'(n) + 'x'(m) + 'x'(?i);
  id 'x'(n?) = n;
#endprocedure
function f,g,x;
symbols a,b,c,d,n,m;
local exp = f(a+b,a+c,a+d) + g(1+b,1+c,1+d);
#call sumargs(f)
print;
.sort

exp =
```

```
   3*a + b + c + d + g(1 + b,1 + c,1 + d);


#call sumargs(g)
print;
.end

exp =
   3 + 3*a + 2*b + 2*c + 2*d;
```

10. Escreva um programa FORM para que, dado um polinômio do quarto grau e uma de suas raízes,

$$a_1x^4 + b_1x^3 + c_1x^2 + d_1x + e_1 = 0,\ r_1 = dado$$

este seja capaz de encontrar o polinômio de terceiro grau,

$$a_2x^3 + b_2x^2 + c_2x + d_2$$

tal que

$$(a_2x^3 + b_2x^2 + c_2x + d_2)(x - r_1) = a_1x^4 + b_1x^3 + c_1x^2 + d_1x + e_1$$

```
\vspace{1pt}     FORM version 3.

   Nw Stat;
   Symbols x;
   Local poly = 2*x^4-20*x^3 + 70*x^2 - 100*x+ 48;
   Local r1 = 2;
   bracket x;
   print;
   .sort

   poly =
       + x * (  - 100 )

       + x^2 * ( 70 )

       + x^3 * (  - 20 )

       + x^4 * ( 2 )

       + 48;

   r1 =
       + 2;


   Local A = poly[x^4];
   Local B = poly[x^3];
   Local C = poly[x^2] ;
   Local D = poly[x];
   Local Exp3 = A*x^3;
   Local Exp2 =((A*r1)+B)*x^2;
   Local Exp1 =(((A*r1)+B)*r1)+C)*x;
```

```
Local Exp0 =(((((A*r1)+B)*r1)+C)*r1)+D);
Local Poly3 = Exp3+Exp2+Exp1+Exp0;
print Poly3;
.end

Poly3 =
    - 24 + 38*x - 16*x^2 + 2*x^3;
```

11. Escreva um programa FORM que faça a união de dois conjuntos.

```
FORM version 3.

Nw Stat;
CFunction uniao,conjunto;
Symbols x,y,a,b,n,m;
Local exp = uniao(conjunto(1,3,5,2),conjunto(2,4,1,5));          repeat;

exp =
   conjunto(1)*conjunto(2)*conjunto(3)*conjunto(4)*conjunto(5);

repeat;
  id conjunto(a?,?m)*conjunto(b?,?n) = conjunto(a,?m,b,?n);
endrepeat;
print;
.end

exp =
    conjunto(1,2,3,4,5);
```

12. Escreva um programa FORM que faça a interseção de dois conjuntos.

```
FORM version 3.

Nw Stat;
CFunction intersecao, conjunto,f;
Symbol x,y,i,j,k,l;
Local exp = intersecao(conjunto(3,7,2,1), conjunto(1,3,4,5));
id intersecao(conjunto(?m), conjunto(?n)) = conjunto(?m)*conjunto(?n);
repeat;
  id conjunto(i?,j?,?m) = conjunto(i) * conjunto(j,?m);
endrepeat;
id conjunto(i?)*conjunto(i?) = intersecao(i);
id conjunto(?i) = 1;
repeat;
  id intersecao(?m)*intersecao(?n) = intersecao(?m,?n);
endrepeat;
id intersecao(?i) = conjunto(?i);
print;
.end

exp =
   conjunto(1,3);
```

13. Desenvolva o procedimento sqr em FORM.

```
FORM version 3.

Nw Stat;
#procedure sqr(exp)
local exp3 = f(z);
argument;
id z='exp';
endargument;
drop 'exp';
.sort
local 'exp' = exp3;
drop exp3;
.sort
id f(c?) = c^2;
#endprocedure
Function f;
Symbols a,b,c,d,z;
local exp1=a+b;
local exp2=1+b;
#call sqr(exp1)


print;
.sort

exp2 =
   1 + b;

exp1 =
   2*a*b + a^2 + b^2;


#call sqr(exp2)


print;
.end

exp1 =
   2*a*b + a^2 + b^2;

exp2 =
   1 + 2*b + b^2;
```

14. Escreva um programa que resolva um sistema de equações da forma:

$$a_1 x + b_1 y + c_1 = 0$$
$$a_2 x + b_2 y + c_2 = 0$$

```
FORM version 3.

Nw Stat;
```

```
Symbols x,y;
Local eq1 = 2*x+3*y -2 ;
Local eq2 = 6*x-6*y -1 ;
bracket x;
print eq1;
.sort

eq1 =

   + x * ( 2 )

   - 2 + 3*y;


Local exp = eq1[x];
Local exp1 =eq1[1]*(-1);
Local X = exp1/exp;
id x=X;
print;
.sort

eq1 = 0;

eq2 =
   5 - 15*y;

exp =
   2;

exp1 =
   2 - 3*y;

X =
   1 - 3/2*y;

bracket y;
print;
.sort

eq1 = 0;

eq2 =
   + y * (  - 15 )

   + 5;

exp =
   + 2;

exp1 =
   + y * (  - 3 )

   + 2;

X =
   + y * (  - 3/2 )

   + 1;
```

```
Local exp2 = eq2[y] ;
Local exp3 = eq2[1]*(-1);
Local Y = exp3/exp2;
id y = Y;
print X,Y;
.end


X =
   1/2;

Y =
   1/3;
```

15. Escreva um programa FORM que converta números em binário para números em decimal. Dado: número binário 10010.

```
FORM version 3.

Nw Stat;
Cfunction bin,decimal;
Symbol x,y;
Local exp=bin(10010);
repeat;
id bin(0,?m) = decimal(?m);
id bin(x?,?m) = bin(x/10,0,(x-1)/10,1,?m);
id bin(?m,x?!int_,y?,?n) = bin(?m,?n);
endrepeat;
print;
.sort

exp =
   decimal(1,0,0,1,0);

repeat;
id decimal(x?,y?,?m) = decimal(x*2+y,?m);
endrepeat;
print;
.end


exp =
   decimal(18);
```

16. Escreva um programa FORM para que, dada a seguinte transformação

$$\begin{cases} x = (\cos\alpha)x' - \sin(\alpha)y' \\ y = (\sin\alpha)x' + \cos(\alpha)y' \\ z = z' \end{cases}$$

possamos mostrar que $F(x,y,z) = F(x',y',z')$.

```
FORM version 3.

Nw Stat;
CFunction F,sin,cos,xl,yl,zl;
Symbols x,y,z;
Indices alpha;
Local exp = F(x,y,z);
print;
.sort


exp =
   F(x,y,z);


id F(x,y,z) = x^2+y^2+z^2;      print;
.sort


exp =
   x^2 + y^2 + z^2;


id x = cos(alpha)*xl - sin(alpha)*yl;
id y = sin(alpha)*xl + cos(alpha)*yl;
id z = zl;
print;
.sort


exp =
   sin(alpha)^2*xl^2 + sin(alpha)^2*yl^2 + cos(alpha)^2*xl^2 +
   cos(alpha)^2*yl^2 + zl^2;


id sin(alpha)*sin(alpha) = 1 - cos(alpha)*cos(alpha);
print;
.end


exp =
   xl^2 + yl^2 + zl^2;
```

17. Levando-se em conta a Série de Taylor, calcule e^x até a terceira potência e calcule um valor aproximado para $e^{0.1}$.

```
FORM version 3.

Nw Stat;
Symbol x;
Local exp = 1 + x + 1/2*x^2 + 1/6*x^3;
id x = 1/10;
print;
.end
```

```
exp =
   6631/6000;
```

18. Sabendo-se que a derivada de $f(n)$ é $f(n+1)$, calcule a derivada de $f(2)f(3)f(4)$.

```
FORM version 3.

Nw Stat;
NFunction f,dx;
Symbols x;
Local exp = f(2)*f(3)*f(1);
Multiply left dx;
.sort
repeat;
id dx*f(x?) = f(x+1) + f(x)*dx;
endrepeat;
id dx = 0;
print;
.end


 exp =
    f(2)*f(3)*f(2) + f(2)*f(4)*f(1) + f(3)*f(3)*f(1);
```