CBPF-NT-002/97

# Introdução ao FORM

*Patrícia Macedo da C. Jorge & Patrícia Duarte Peres*

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

Endereço permanente:
Universidade Católica de Petrópolis (UCP)
Petrópolis - RJ

---

Notas do curso conjunto UCP–CBPF sobre "Aplicações do software FORM em Problemas de Teorias-de-Gauge"

**Abstract**

O programa FORM para manipulação simbólica é uma poderosa ferramenta para realizar grandes cálculos algébricos.É uma linguagem simples, de fácil aprendizado, que é complementar de grandes programas como o Maple e o Matematica e que fornecem as maneiras de se manipular grandes fórmulas.

# Contents

# 1 Estrutura de um Programa FORM

## 1.1 Como utilizar o FORM

O Form é um programa que trabalha com arquivos em lote. Isto significa que o modo usual de operação é editar um arquivo com instruções (o programa FORM), rodar o FORM sobre este arquivo e analisar a resposta. Este círculo de editar-compilar foi desenvolvido para ser mais conveniente e flexível do que interfaces gráficas para a manipulação de grandes fórmulas.

Portanto, não precisa-se aprender um editor de texto; mas podemos utilizar o nosso preferido (emacs, DOS, MS Word,...). A extensão padrão para um programa FORM é `.frm`.

O comando para rodar o FORM depende do sistema que se está operando; tipicamente isto se parece como

```
> form -l formprog.frm
```

O sufixo `-l` diz ao FORM para armazenar uma cópia do programa e da resposta no arquivo `formprog.log`.

# 2 Um programa de Exemplo

Temos abaixo um pequeno programa que mostra as características típicas de um programa FORM, mesmo se este não possuir muita utilidade prática.

```
* exemplo programa um.
Symbols a,b;
Local f=(a+b)^20;
print;
.sort
Symbol [a+b];
id a = [a+b] - b;
print;
.end
```

A primeira linha é um comentário - o mais importante tipo de declaração em qualquer programa. Logo após temos uma declaração de símbolos `a` e `b`. Outros tipos de dados serão discutidos posteriormente. A terceira linha declara uma expressão (local) para o FORM operá-lo, neste caso $f = (a+b)^{20}$. Portanto o programa FORM

é instruído para imprimir o resultado; como o FORM foi designado para manipular grandes expressões o padrão é não imprimir coisa alguma. O módulo é terminado pela instrução .sort, que permite que as declarações anteriores sejam compiladas e executadas.

Há um segundo módulo, no qual um novo símbolo [a+b] é definido. O uso de colchetes é uma forma especial de um nome de variável, que é muito útil para significar um símbolo para um leitor humano, sem confundir o FORM, que pega-o e o entende equivalentemente como um c. O FORM, posteriormente, é instruído para substituir cada ocorrência do símbolo a pelo lado direito, e imprimir o resultado. Isto encerra o programa.

Por curiosidade, aqui está um programa simples juntamente com sua resposta.

```
  FORM version 2.1 Aug 21 1992

    * exemplo programa um.
    Symbols a,b;
    Local f=(a+b)^20;
    print;
    .sort

Time =       0.05 sec    Generated terms =         21
               f         Terms in output =         21
                         Bytes used      =        368

   f =
      20*a*b^19 + 190*a^2*b^18 + 1140*a^3*b^17 + 4845*a^4*b^16 + 15504*a^5*
      b^15 + 38760*a^6*b^14 + 77520*a^7*b^13 + 125970*a^8*b^12 + 167960*a^9*
      b^11 + 184756*a^10*b^10 + 167960*a^11*b^9 + 125970*a^12*b^8 + 77520*a^13
      *b^7 + 38760*a^14*b^6 + 15504*a^15*b^5 + 4845*a^16*b^4 + 1140*a^17*b^3
       + 190*a^18*b^2 + 20*a^19*b + a^20 + b^20;


    Symbol [a+b];
    id a = [a+b] - b;
    print;
    .end

Time =       0.33 sec    Generated terms =        231
               f         Terms in output =          1
                         Bytes used      =         18

   f =
```

```
[a+b]^20;
```

O FORM multiplicou os colchetes e colocou a resposta em sua forma padrão: um polinômio. Isto também economiza tempo de CPU e uso do arquivo; esta informação pode ser importante quando trabalhamos com grandes problemas. Em um segundo passo ele reduz todos os termos, e, para o leitor, quase reproduz a expressão original.

# 3 Estrutura Geral de um Programa FORM

A estrutura geral de um programa FORM é uma série de `módulos` terminados por `.sort`, o último é terminado com `.end`. No primeiro módulo é declarado, primeiramente, todas as (novas) variáveis e, logo após, as expressões sobre as quais o FORM irá operar. Estas operações vêm em seguida; e o módulo é normalmente terminado por `print` e instruções de formato.

Cada declaração tem que começar em uma linha nova e terminar com um ponto-e-vírgula; qualquer coisa depois do ponto-e-vírgula é entendido como um comentário. Uma declaração pode se extender por mais linhas e podem ser indentadas para facilidade de leitura[1]. Muitas declarações consistem de um palavra de comando, seguida por um espaço em branco ou uma vírgula, opções para o comando e o resto do comando:

```
id,select,set x^n?*dx = x^(n+1)/(n+1);
```

Nomes internos do FORM, como palavras de comando, são caso insensitivo (não importa se são escritas em letras maiúsculas ou minúsculas), mas variáveis definidas pelo usuário são caso sensitivo.

Outro detalhe da linguagem FORM é que, internamente, espaços em branco com nenhum operador no lado esquerdo ou direito é substituído por vírgulas. Isto, às vezes, fornece a infame mensagem de erro

```
---> Illegal comma at bracket level zero
```

quando você esquecer um ponto-e-vírgula no final da declaração, e mesmo quando nenhuma vírgula explícita foi usada em lugar algum.

# 4 Trabalhos Internos do FORM

É útil algumas vezes ter alguma idéia do que o FORM faz internamente quando escrito programas FORM. Como você já pode ter adivinhado desde do início, para

[1] Instruções do Pré-processador (como `.sort`, veja também seção 2.5) não possuem ponto-e-vírgula e não podem ser continuadas na linha seguinte

o FORM nem todos os operadores (+,-,*,/,$f$) são iguais. Em particular isto define uma expressão como uma soma de termos; cada termo consistindo de um coeficiente numérico (como um número racional) vezes um produto de fatores, possivelmente para alguma potência. Os fatores podem ser símbolos ou estruturas mais complicadas, como funções. Esta é a estrutura sobre o qual o FORM opera mais eficientemente; felizmente a maioria das fórmulas possuem esta forma. É possível operar sobre estruturas arbitrárias como adição(potência($a$, 2)), multiplicação(2, $a$, $b$), potenciação($b$, 2) ao invés de $a^2 + 2ab + b^2$, mas esta não é, definitivamente, o ponto forte do FORM.

O método de operação do FORM é como se segue. Ele primeiro lê um módulo e compila-o; depois aplica a operação especificada no módulo para o primeiro termo da primeira expressão (possivelmente gerando novos termos), e traz os termos resultantes na forma padrão. Ele faz o mesmo para o termo seguinte, e assim por diante até que todos os termos tenham sido processados. O FORM então ordena o resultado e chama-o na expressão do módulo seguinte. Todo este processo sequencial é no princípio feito de arquivo para arquivo, tanto que o computador é usado o mais eficientemente quanto possível. É claro que existem buffers e vários estágios de ordenamento para a eficiência, mas o modo básico é ordenar um termo em desordem de uma expressão através dos operadores que você especificou. Este modo de operação significa que o FORM não contém nenhuma declaração que se refira a mais do que um termo de cada vez (exceto ordenando); não existe nenhuma maneira de fazer este tipo de sentença

```
id a+b=c; <= não existe!
```

Isto também significa que não existe fatorização construída dentro da corrente versão do FORM, a não ser truques que serão mostrados em programas exemplo para simplificar expressões. Todas as operações são locais para um termo.

(Chapter head:)Alguns Comandos

Esta seção é voltada para uma visão dos comandos mais utilizados do FORM, ilustrados por exemplos. Esta seção não pretende ser completa. Todos os detalhes podem ser encontrados nas documentações do FORM. Muitos comandos em FORM podem ser abreviados, portanto, nos exemplos seguintes a notação `C[ommand]` significará o nome `Commando`.

# 5 Declarações

No FORM, você terá de declarar explicitamente todas as variáveis que você deseja utilizar antes de usá-las. As regras para nomes de variáves normais são simples: eles devem começar com uma letra e ser seguido por letras ou números. Eles são caso sensitivo; `A` é diferente de `a`. Um tipo especial de nome é formado por um par de colchetes com (quase) nada entre eles, por exemplo `[log(a+b)]` é o nome de uma variável, apesar de para um leitor humano trazer informações adicionais. Não há nenhuma restrição prática para o tamanho de nomes, exceto que nomes de expressões

podem ter no máximo 16 caracteres. Nomes terminando com um underscore( _ ) são reservados para um uso interno, como `i_`=$\sqrt{-1}$.

O FORM conhece os seguintes tipos de variáveis:

## 5.1 S[ymbols]

Símbolos são os objetos mais comuns escritos em fórmulas. Formalmente, são objetos que não possuem nenhum argumento, pode possuir uma potência, e comutar com qualquer coisa. Uma propriedade útil quando estamos fazendo séries de Taylor é a possibilidade de especificar a maior (ou menor) potência

```
Symbol z(:5)
```

Isto irá substituir todos os termos com $z^6$ ou maiores por zero.

```
 FORM version 2.1 Aug 21 1992

    S z(:5);
    L f = (1+z)^10;
    print;
    .end

Time =       0.00 sec    Generated terms =          6
                f        Terms in output =          6
                         Bytes used      =         74

   f =
      1 + 10*z + 45*z^2 + 120*z^3 + 210*z^4 + 252*z^5;
```

Como você vê os termos com potências mais altas de $z$ foram substituídos por zero antes do estágio de ordenação, e não foram incluídos na contagem dos termos.

## 5.2 F[unctions], CF[unctions]

Funções possuem três propriedades que distingue-as dos símbolos: elas podem possuir um argumento, não podem possuir potências e não podem comutar com outras funções. Se a última propriedade não for desejada, pode-se utilizar CF[unctions].

Objetos que não comutam são, às vezes, uma maneira muito útil de escrevermos matrizes ou operadores:

```
  FORM version 2.1 Aug 21 1992

    Symbols h,m;
```

```
Functions H,x,p;
Local comHx = H*x - x*H;
id H=p^2/(2*m);
Nw Statistics;
print;
.sort

comHx =
    - 1/2*x*p*p*m^-1 + 1/2*p*p*x*m^-1;


repeat;
id x*p = p*x + i_*h;
endrepeat;
print;
.end

comHx =
    - p*i_*h*m^-1;
```

A frase `nwrite statistics` suprime a informação de ordenação, e a declaração `repeat` será discutida na seção 3.4.1.

## 5.3 I[ndices], V[ectors], Delta de Kronecker (d_)

Uma característica que distingue o FORM é sua construção de conhecimento para vetores e índices. A notação usada para vetores é $\vec{p} = p_i =$ `p(i)`,onde $i$ é o índice. A convenção do somatório de Einstein é sempre assumida: `p(i)*q(i)` $=\sum_{i=1}^{N} p_i q^i =$ p.q, com $N$ a dimensão do vetor espacial[2]. O padrão para isto é 4, mas pode ser mudado pela declaração `Dimension`, ou na declaração do índice. O delta de Kronecker é denotado por `d_`; o FORM conhece suas propriedades.

Quando uma função possui um índice, e este índice é contraído com um vetor em algum lugar, O FORM segue a convenção de SchoonSchip e substitui o argumento pelo vetor. Isto torna fórmulas complicadas, às vezes, muito mais legíveis:

$$\epsilon^{\mu\nu\rho\sigma} p_\mu q_\nu r_\rho s_\sigma = \epsilon^{pqrs}$$

No FORM, o tensor de Levi_Civita $\epsilon$ é totalmente anti-simétrico e representado por `e_`.

[2]Isto pode ser suprimido colocando-se a dimensão do índice para zero

```
FORM version 2.1 Aug 21 1992

 Dimension 3;
 Vectors p,q,r,s;
 Indices i,j, mu=4, nu=4, ro=4, si=4;
 Local f1 = p(i)*d_(i,j) + p(i)*q(i)*q(j);
 Local dim3 = d_(i,i);
 Local dim4 = d_(mu,nu)*d_(nu,mu);
 Local det = e_(mu,nu,ro,si)*p(mu)*q(nu)*r(ro)*s(si);
 nw statistics;
 print;
 .end

f1 =
   p(j) + q(j)*p.q;

dim3 =
   3;

dim4 =
   4;

det =
   e_(p,q,r,s);
```

O FORM 2.0 possui também um tipo especial de função chamado `Tensor.`; ele é linear em seus argumentos e mais rápido do que funções normais.

## 5.4 L[ocal],G[lobal],expressões

Até agora vimos somente como trabalhar com expressões locais; expressões globais são, de fato, somente usadas quando expressões tem de ser salvas em um arquivo. Veja seção 2.7

Expressões podem ser utilizadas no lado direito de declarações. Quando isto ocorre no mesmo módulo como a declaração, o lado direito da definição é somente copiado, senão a última forma sorteada é usada.

# 6 Substituições (`id`)

Muitas das operações em um programa FORM são normalmente na forma de substituições: substituir um termo por outro. A declaração `id` realiza esta tarefa em uma

variedade de maneiras. Em sua forma mais simples já a encontramos algumas vezes. O lado esquerdo pode ser o produto de alguns fatores com expoentes, mas não pode conter um fator numérico ou ser a soma de termos.

```
id 4*a*b = c; ERRADO
id a+b=c; ERRADO
```

Um ponto curioso, e fonte de muitos erros, é que uma potência negativa não é a mesma coisa que uma potência positiva:

```
FORM version 2.1 Aug 21 1992

   Symbols a,b,c, [b+c];
   Local curioso = a+ 1/a;
   id a= b+c;
   id 1/a = 1/[b+c];
   nw statistics;
   print;
   .end

curioso =
   b + c + [b+c]^-1;
```

## 6.1 Wildcards

O grande poder da declaração `id` somente torna-se aparente quando wildcards são utilizadas. Estes são objetos que comparam qualquer objeto similar, e indicado no lado esquerdo por um ponto de interrogação atrás da variável (que tem de ser declarada). Por exemplo:

```
Symbols x,n;
id x^n = x^(n+1)/(n+1);
```

integrará um polinômio em $x$ (fornecido - não existe termos com $1/x$). As regras para comparação de termos são simples. Sempre que existir uma variável no lado esquerdo seguida por um ponto de interrogação, esta variável pode "ser comparada" com qualquer variável de tipo similar. No lado direito as ocorrências desta variável são então substituídas pela varíavel que está sendo comparada. Na integração da declaração acima, se em um termo $x^6$ colocarmos $n$ para 6, o lado direito será, portanto, $x^{(6+1)}/(6+1)$. Um termo $y^6$ não será comparado , como o $x$ não possui um ponto de interrogação. Um termo $7y^6x^6$ fornecerá $y^6x^7$.

As regras cujo tipo de objetos podem ser comparados são as que seguem.

- Um símbolo representa um outro símbolo, ou um argumento de função, ou uma expressão completa.

- Um vetor com índices representa exatamente aquela combinação, um vetor sem índices representa qualquer vetor (também em produtos internos e tensores de Levi-Civita), ou como um argumento de função, um vetor com argumento. No lado direito, o símbolo ? extende o número de índices se necessário:

```
id p = e_(q1,q2,q3,?);
```

Um índice representa outro índice ou um vetor que poderia ter um índice se a convenção de SchoonShip não tiver sido aplicada.

- Uma função somente representa outra função se o número e tipos de argumento combinarem.
- Substituições *não são* executadas dentro de argumentos de funções. No FORM 2.0, a declaração `id` pode ser substituída por `Argument` e `Endargument` para forçar o FORM a olhar dentro dos argumentos de função.
- Qualquer número de argumentos podem ser substituídos com os termos `?,??,???,...`; ,no lado direito estas variáveis correspondem ao mesmo número de pontos.

Combinações pequenas e raras de termos não podem ser implementadas em FORM 1. Aqui são alguns exemplos utilizando vetores e índices:

```
FORM version 2.1 Aug 21 1992

 V p1,p2,p3,p4;
 I m1,m2,m3,m4;
 CF delta;
 L f= delta(p1,p2,p1,p2);

 id p1 = p2+p3;
 nwrite statistics;
 print;
 .sort

f =
   delta(p1,p2,p1,p2);



 *expandindo os deltas de kronecker generalizados

 id delta(m1?,m2?,m3?,m4?) = d_(m1,m3)*d_(m2,m4) - d_(m1,m4)*d_(m2,m3);
```

```
id p1 = p2+p3;
print;
.end

f =
   p2.p2*p3.p3 - p2.p3^2;
```

## 6.2 Conjuntos

Podemos ter casos que desejamos restringir a margem de uma variável wildcard; por exemplo `N?` deveria comparar `N1,N2,N3`, mas não `x`. A solução para isto é a possibilidade de declararmos conjuntos de variáveis:

```
Set NN:N1,N2,N3;
```

Com isto podemos especificar

```
id N?NN = fun(N)
```

e somente as variáveis $N_1, N_2$ e $N_3$ serão afetadas. Outro uso de conjuntos é a possibilidade de exigirmos que *todas* as variáveis de um certo tipo sejam substituídas. Por exemplo, quando integrando, podemos ter variáveis separadas para $x$ e $\log(x)$.

Escrevendo

```
id dx*x^n?  = x^(n+1)/(n+1);     ERRADO
id dx*x^n*[log(x)] = x^(n+1)/(n+1)*([log(x)] - 1/(n+1));
```

estaria errado, como no termo $x\log(x)$ a primeira declaração executaria a substituição antes do segundo termo. A solução é utilizar um conjunto:

```
Set xx:dx,x,[log(x)];
id,select,xx dx*x^n = x^(n+1)/(n+1);
id,select,xx dx*x^n*[log(x)] = x^(n+1)/(n+1)*([log(x)]-1/(n+1));
```

A opção `select` permite que a substituição somente seja executada quando não existir nenhum termo do conjunto deixado para trás depois da substituição, portanto no termo $x\log(x)$ somente a segunda declaração é executada. Poderíamos ter o mesmo efeito por ordenação cuidadosa, mas que é muito mais propícia a erros.

## 6.3 Opções

Exceto pela opção `select` discutida acima, existem muitas possibilidades de influenciar o comportamento da declaração `id`. Uma opção útil é `once`, que causa ao termo a ser usado somente uma substituição dentre as muitas possíveis. Desta maneira podemos, cuidadosamente, trabalhar com somente um nível de cada vez:

```
FORM version 2.1 Aug 21 1992

CF f1,f2;
S a,b;
L f=f1(a)*f1(b);
id,once f1?(??) = f2(..);
nw statistics;
print;
.end

f =
   f1(b)*f2(a);
```

# 7 Outras Operações

Existem outras operações comuns, todas, é claro, agindo sobre um único termo de cada vez:

`Multiply,`*expressão* multiplica o termo com a expressão; as opções `left` e `right` pode ser usada para objetos não-comutativos.

`Discard` é o mesmo que `Multiply,0`. É útil principalmente em combinações com a declaração `if`.

`Contract` contrai todos os pares de tensores Levi-Civita.

`Trace4,` $i$ tira o traço de uma linha das matrizes gamma de Dirac (com índices $i$) em 4 dimensões, veja exercício 5.3 para descrição mais detalhada e um exemplo.

`Tracen,` $i$ faz o mesmo em $n$ dimensões.

# 8 Controle de Módulos

As capacidades de substituição são ampliadas pelas possibilidades de controle de módulos: a habilidade de aplicar operações em certos termos ou somente em expressões. Nós discutimos os mais usados: **`repeat`**, **`if`**, **`skip`** e **`drop`**. Existem mais: **`while`**, e (como em qualquer linguagem de programação REAL) **`goto`**, que são discutidas no manual. Todas as construções discutidas aqui devem estar contidas em um módulo; um módulo que não pode possuir uma declaração **`.sort`** dentro de uma declaração **`if`** ou **`repeat`**.

## 8.1 Repeat

Algumas vezes queremos aplicar um certo conjunto de substituições um certo número de vezes, até que elas não tenham mais efeito. Isto pode ser feito utilizando-se os comandos repeat e endrepeat:

```
FORM version 2.1 Aug 21 1992

    *Exemplo de reducao de uma cadeia de matrizes gamma de Dirac
matrizes. Matrizes gamma sao representadas pela funcao g, a funcao
pre-construida g_ somente deve ser usada para tracos em FORM 1.0

    Function g;
    Vectors p1,p2;
    Indices mu,nu;
    Symbols m1,m2;

    Local string = g(mu)*g(p2)*g(nu)*g(p1)*g(mu);

    repeat;
    * anti-comute p1 para a esquerda
    id g(mu?)*g(p1) = - g(p1)*g(mu)+2*p1(mu);
    endrepeat;

    *p1 esta agora do lado esquerdo
    id g(p1)=m1;

    repeat;
    * anti-comute p2 para a direita
    id g(p2)*g(mu) = - g(mu)*g(p2)+2*p2(mu);
    endrepeat;

    id g(p2)=m2;

    *use algumas propriedades de matrizes gamma
    id g(mu?)*g(mu?) = d_(mu,mu);
    id g(mu?)*g(nu?)*g(mu?) = (2-d_(mu,mu))*g(nu);

    print;
    .end

Time =       0.28 sec    Generated terms =          9
          string         Terms in output =          1
```

```
                    Bytes used      =         30

   string =
      4*g(nu)*m1*m2;
```

## 8.2 If

Se desejarmos aplicar as operações somente para um subconjunto de termos existe a declaração `if`. A condição pode ser construída sobre três funções.

`match(`*wildcards*`)` retorna o número de vezes que as *wildcards* representam o termo, como o lado esquerdo de uma substituição.

`count(`*objeto, tamanho*`)` conta o número de vezes que um objeto ocorre com um certo tamanho no termo, também útil para checar dimensões.

`coefficient` retorna um coeficiente numérico.

Comandos `if` podem ser combinados com operadores C: ==,!=,>=,<=,>,<,&&,||. Podemos extender a sintaxe com `else` e `elseif` como esperado.

```
  FORM version 2.1 Aug 21 1992

    V p1,p2,p3,p4;
    L err = (e_(p1,p2,p3,p4))^2 - p1.p1*p2.p2*p3.p3;
    Contract,0;
    .sort

Time =       0.06 sec    Generated terms =         25
              err        Terms in output =         18
                         Bytes used      =        488

    *checando dimensoes!
    if (count(p1,1,p2,1,p3,1,p4,1)==8);
    discard;
    endif;
    nw statistics;
    print;
    .end

   err =
       - p1.p1*p2.p2*p3.p3;
```

### 8.3 Skip,Drop

Às vezes desejamos aplicar uma certa operação somente em um subconjunto de expressões, ou pular uma expressão completamente. A declaração `skip` *expressões* em qualquer lugar de um módulo significa que as operações daquele módulo não são aplicadas àquela expressão; a declaração **drop** significa que não somente a expressão estará inativa no módulo corrente, mas também será apagada ao final do módulo. Note que expressões inativas estão também disponíveis para uso no lado direito de declarações. Pular uma expressão que é novamente definida no mesmo módulo fornece resultados estranhos.

```
Local wrong = (a+b)^20;
Skip wrong;                    <=== os resultados estão indefinidos
.sort
```

Infelizmente, a única maneira de aplicar operações em somente uma expressão é pular todas as outras (ou usar truques como declarar a expressão com o fator global). O uso freqüente destas características está em algumas checagens rápidas entre os módulos durante o cálculo, como em:

```
L bigexpression = ...;
...muitas operações...
.sort
*checando dimensões
skip bigexpression;
L zero = bigexpression;
if ( count(p1,1,p1,1,m,1,s,2)==8)
discard;
endif;
print;
.sort
drop zero;
...and go on...
```

# 9 Pré-processador

O Form é equipado com um pré-processador como a linguagem C. Este pré-processador muda o input para um nível textual puro; antes do atual compilador interpretar o código. Ele possui variáveis, loops-do, declarações if e procedimentos, e oferece a possibilidade de incluir outros arquivos. Procedimentos não são utilizados aqui, precisamos de um mínimo de experiência com programação em Form para sermos capazes de escrever procedimentos com bons resultados .

## 9.1 # include

Este inclui o arquivo na posição atual no programa. Um uso comum é possuir declarações às vezes usadas em um arquivo cabeçalho, apesar de podermos incluir qualquer coisa que desejarmos.

## 9.2 #define

Podemos definir variáveis textuais, e depois utilizá-las:

```
#define N ''100''
#define CHECK ''1''
...
Symbol z(:'N'|);
#if 'CHECK' ...
```

Para tornar o uso destas variáveis tão geral quanto possível a sintaxe é um tanto quanto peculiar, e uma fonte frequente de erros de digitação.

## 9.3 #do

Loops-do são um tipo especial de variáveis. Pequenos cálculos (inteiros) podem ser realizados para colocarmos limites na repetição **do**. A sintaxe e uso devem ficar mais claras com o próximo exemplo. A variável de repetição novamente é uma variável puramente textual; o seguinte é uma maneira rápida para declarar muitas variáveis.

```
Symbols
#do i=1,42
      A 'i'
#enddo
;
```

Checamos as rotinas aritméticas do FORM computando $\exp(\log(1+x))-1=x$ utilizando as expansões em série $\log(1+x)=\sum_{i=1}^{N}(-)^{i+1}\frac{x^i}{i}+O(x^{N+1})$ e $\exp(x)=\sum_{i=1}^{N}\frac{x^i}{i!}++O(x^{N+1})=x(1+\frac{x}{2}(1+\frac{x}{3}(1+...)))$. A definição de $X$ novamente mostra a natureza textual do Pré-processador.

```
FORM version 2.1 Aug 21 1992

 *checar que exp(ln(1+x))-1=x, adaptado do manual de FORM

 #define N ''20''
 S x (: 'N'),y;

 *definindo log(1+x)
```

```
L X=
#do i=1,'N'
      + (-1)^('i'+1)*x^'i'/'i'
#enddo
;
id x=x*y;

*e expandir - aos poucos

#do i=2,'N'
    id y = 1 + y*x/'i';
    nwrite statistics;
    .sort

#enddo

*fazendo y igual a 1

id y=1;

write statistics;
print;
.end
```

```
Time =       0.99 sec    Generated terms =          3
                X        Terms in output =          1

                         Bytes used      =         18

   X =
      x;
```

Uma otimização foi usada aqui: a instrução `.sort` dentro do segundo loop do força uma combinação de termos similares; sem ele o programa roda muito mais lentamente. O leitor interessado deve desejar dar uma olhada em alguns dos resultados intermediários.

## 9.4 #if

O pré-processador `if` (não confunda com o compilador `if` da seção 2.4.2) é normalmente usada para dois própositos. O primeiro é executar o código somente quando certas marcas são definidas no início:

```
#define CHECKING ''0''
...
#if 'CHECKING'
*checando a dimensão
...
#endif
```

O segundo uso comum é excluir parte do loop:

```
L E=1/2*m*v^2
#do i=1,'N'
      #do j='i','N'
            #if 'i'!= 'j'
                  + e^2/(x 'i'-x 'j')
            #endif
     #enddo
#enddo
;
```

A declaração `if` do pré-processador *não* possui a possibilidade de combinação de "and" e "or"; devemos usar uma concatenação de estruturas `if`, e nem existe a declaração "else".

Formalmente, as declarações de fim de módulo como `.sort` e `.end` são também consideradas parte do pré-processador.

# 10 Funções Especiais

Nas seções anteriores encontramos o delta de Kronecker `d_` e o tensor Levi-Civita `e_` (seção 2.1.3). Outras funções especiais são:

`delta_(`$a$`)`$= 1$ se $a = 0; 0$ se $a \neq 0$ mas numérico.

`theta_(`$a$`)`$= 1$ se $a \succeq 0; 0$ se $a \prec 0$. Não é necessário pensarmos muito; `\theta_(x^2)` não é simplificado.

`fac_(`$n$`)` $= n!$

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

`g_` A função da matriz Gamma de Dirac é de interesse fundamental para a Física de Partículas.

Tenha cuidado que substituições utilizando-a não estão disponíveis em FORM 1.0 .

# 11 Output

## 11.1 Print,Format,Bracket

Apesar de uma impressão bonita não estar incluída, o FORM possui algum controle sobre a output (resposta). Como mencionado antes, o padrão é não imprimir coisa alguma. Quando a declaração `print` é encontrada em qualquer lugar do módulo, a expressão mencionada é impressa ou todas as expressões são impressas. O sufixo `+f` indica que se o FORM estiver rodando com o sufixo `-l`, a expressão impressa deve ir somente para o arquivo. O sufixo `+s` força cada termo a começar em uma nova. linha.

O comando `b[racket]` diz ao FOR|M para retirar a variável indicada de dentro dos parênteses quando imprimirmos a expressão (ou expressões). Isto pode ser útil quando também retirarmos algum fator global, ou para ver a estrutura mais claramente.

```
   FORM version 2.1 Aug 21 1992

    #define N ''3''
    S eps(:'N'),e,i,j,x,y;
    F f,g;

    L fun = e^2*sum_(i,0,'N',f(i)/fac_(i)*eps^i*
                sum_(j,0,'N'-i,g(j)*eps^j));


    Bracket e,eps;
    print;
    .end

Time =       0.16 sec    Generated terms =         21
              fun        Terms in output =         10
                         Bytes used      =        278

   fun =
       + eps*e^2 * ( f(0)*g(1) + f(1)*g(0) )

       + eps^2*e^2 * ( f(0)*g(2) + f(1)*g(1) + 1/2*f(2)*g(0) )


       + eps^3*e^2 * ( f(0)*g(3) + f(1)*g(2) + 1/2*f(2)*g(1) + 1/6*f(3)*g(0) )

       + e^2 * ( f(0)*g(0) );
```

Pode também pegar os coeficientes de uma expressão, como ilustrado no próximo exemplo:

```
FORM version 2.1 Aug 21 1992

  Nw Stat;
  S a,b,c,x;
  L poly = a*x^2 + b*x + c;
  B x;
  .sort

  L disc = poly[x]^2 - poly[x^2]*poly[1];

  print disc;
  .end

 disc =
     - a*c + b^2;
```

A declaração `format fortran` pode ser usada para forçar o FORM a fornecer uma resposta no formato Fortran.

## 11.2 Save,Load

Existe a possibilidade de salvar uma expressão como arquivo binário e carregá-lo em outras computações. A sequência básica é:

```
Global expression = ...
.store
save file.sav;
.end
```

Depois de `.store` a expressão torna-se inativa; ela somente pode ser usada no lado direito, mas não é modificada pelas operações. A extensão `.SAV` é uma convenção. Carregando-a a informação é acompanhada de:

```
load file.sav;
Local new = expression;
```

A expressão carregada está novamente inativa. Isso está modificado na versão atual do FORM.

# 12 Truques e Algoritmos Úteis

Nesta seção alguns algoritmos e truques foram coletados. Em geral, escrever um programa FORM é simples: devemos traduzir somente as manipulações que desejamos

de um pedaço de papel para dentro de uma série de comandos FORM. Existem, no entanto, algumas poucas diferenças. Primeiro, o FORM é fraco em simplificação de termos similares, tanto que os resultados tornam-se muito maiores do que se estivéssemos fazendo no papel. Algumas técnicas de simplificação são úteis para reduzir o tamanho da fórmula. O segundo problema é descobrir quais termos devem ser tratados e quais não, mas isto pode ser resolvido utilizando-se uma declaração. Finalmente, mostraremos um simples, mas eficiente, método para resolver um conjunto de equações, e outro utilizando projeções.

# 13 Simplificação

No final de um cálculo, normalmente não desejamos muitos papéis com a resposta da fórmula. Infelizmente, isso é o que às vezes obtemos. A simplificação de uma grande fórmula é um trabalho difícil, e normalmente requer algum conhecimento da estrutura da solução. Como um primeiro passo é sempre sensato retirar dos parênteses todas as estruturas globais, tanto que a estrutura torna-se mais clara. Por inspeção do resto, podemos encontrar alguns termos promissores que podem ser investigados utilizando substituições. Como exemplo, considere o seguinte programa: a entrada é a resposta de alguns programas anteriores. Cada declaração é o resultado de experimentações até que um resultado simplificado seja encontrado.

```
FORM version 2.1 Aug 21 1992

    S a,b,c;
    L f = - 10*a^3*b^9*c + 45*a^4*b^8*c - 120*a^5*b^7*c
          + 210*a^6*b^6*c - 252*a^7*b^5*c + 210*a^8*b^4*c
          - 120*a^9*b^3*c + 45*a^10*b^2*c - 10*a^11*b*c
          + a^12*c;
    Bracket a,c;
    nw statistics;
    print;
    .sort

   f =
       + a^3*c * (  - 10*b^9 )

       + a^4*c * ( 45*b^8 )

       + a^5*c * (  - 120*b^7 )

       + a^6*c * ( 210*b^6 )
```

```
    + a^7*c * (  - 252*b^5 )

    + a^8*c * ( 210*b^4 )

    + a^9*c * (  - 120*b^3 )

    + a^10*c * ( 45*b^2 )

    + a^11*c * (  - 10*b )

    + a^12*c * ( 1 );


 S [a-b];
 * nao podemos trabalhar com 1/a^3
 multiply 1/a^2/c;

 id a = [a-b] + b;
 print;
 .end

f =
    - b^10 + [a-b]^10;
```

Outra possibilidade é "dar um chute" de qual deveria ser o resultado, subtraí-lo do original e trazê-lo à forma padrão para vermos se a diferença é ou não zero. Isto é especialmente útil quando existem muitas variáveis dependentes, e a forma padrão é muito maior do que a forma original. Este exemplo prova uma identidade entre determinantes que é útil na avaliação da função de três pontos escalares. Os determinantes são descritos como delta de Kronecker generalizados:

$$\delta^{\mu_1 \cdots \mu_n}_{\nu_1 \cdots \nu_n} = \begin{vmatrix} \delta^{\mu_1}_{\nu_1} & \cdots & \delta^{\mu_1}_{\nu_n} \\ \cdots & \cdots & \cdots \\ \delta^{\mu_n}_{\nu_1} & \cdots & \delta^{\mu_n}_{\nu_n} \end{vmatrix}$$

Desejamos mostrar que

$$s^2_{i+1} - \delta^{s_1 s_2 s_3}_{s_1 s_2 s_3}/\Delta_2 = \left(\delta^{p_i\, p_{i+1}}_{\mu\, s_{i+1}}\right)^2/\Delta_2$$

com $\Delta_2 = \delta^{p_1 p_2}_{p_1 p_2}$ e $p^\mu_i = s^\mu_{i+1} - s^\mu_i$.

```
FORM version 2.1 Aug 21 1992

   V p1,p2,p3,s1,s2,s3;

   I m1,m2,m3,m4,mu;
   S Del2;
   CF delta;

   L LHS = s2.s2 - delta(s1,s2,s3,s1,s2,s3)/Del2;
   L RHS = (delta(p1,p2,mu,s2))^2/Del2;

   L zero = LHS - RHS;
   nwrite statistics;
   .sort

   drop LHS,RHS;

   *colocando-os sobre um denominador comum

   Multiply Del2;

   *multiplicando os determinantes

   id Del2 = delta(p1,p2,p1,p2);
   id delta(p1?,p2?,p3?,s1?,s2?,s3?) =
      p1.s1*p2.s2*p3.s3 - p1.s1*p2.s3*p3.s2
      - p1.s2*p2.s1*p3.s3 + p1.s2*p2.s3*p3.s1
      + p1.s3*p2.s1*p3.s2 - p1.s3*p2.s2*p3.s1;

   id delta(m1?,m2?,m3?,m4?) = d_(m1,m3)*d_(m2,m4) - d_(m1,m4)*d_(m2,m3);

   * colocando-os em uma base de vetores padrao

   id p1 = s2 - s1;
   id p2 = s3 - s2;
   id p3 = s1 - s3;
   write statistics;

   print;
   .end

Time =       0.28 sec    Generated terms =          57
```

```
            zero        Terms in output =          0
                        Bytes used      =          2

   zero = 0;
```

# 14 Derivação, Integração

Apesar do FORM não conhecer nada sobre análise, ele pode ser usado para integrar e derivar. Devemos somente reduzir as integrais para substituições utilizando outros sistemas de computação algébrica, nosso próprio conhecimento ou conjunto de tabelas. O FORM usa estas regras para integrar as (grandes) fórmulas. (Alguns cálculos interessantes em processsos de QCD a 2 e 3 loops foram realizados deste modo). Um truque útil para se utilizar aqui é marcando: para sabermos quais termos serão tratados e quais não, a fórmula inteira é multiplicada por um fator global que será usado na substituição. Com a opção `select`, seção 2.2.2, um programa de integração seria assim:

```
S dx,x,[log(x)],...;
Set xx:dx,x,[log(x)],...; todas as variaveis que dependem de x
L expr = ...

Multiply dx;

* a tabela de integrais

id,select, xx, dx/x = [log(x)];
id,select, xx dx*x^n? = x^(n+1)/(n+1);
...

* observe se existem alguns termos nao integrados a esquerda

bracket dx;
print;
.end
```

Derivação pode ser realizada com um truque similar. Permita que $f(n,x)$ seja a enésima derivada de $f$ em relação a $x$.[3]

[3]Este exemplo foi copiado da Ref.[5]

```
Functions g1,g2,dx,g;
Symbols x,n,m;
Local F = g1(0,x)*g2(m,x);

* tirando a terceira derivada

Multiply left dx*dx*dx;
repeat;
    id dx*g?(n?,x) = g(n+1)+g(n,x)*dx;
endrepeat;
id dx = 0;
print;
.end
```

# 15 Resolvendo Conjuntos de Equações

Novamente, o FORM não possui um comando prédefinido "`solve`". Uma maneira eficiente e surpreendente para resolver um grande conjunto de equações (em qualquer sistema algébrico) é pegar o menor de todos, resolvê-lo para uma variável utilizando seu editor favorito ou o sistema de computação algébrica de sua preferência e usá-lo como substituição para reduzir o número de equações. Repita este passo até terminar.

Outro método que às vezes é utilizado seria projetar as soluções. Como um simples exemplo para físicos de partículas, considere a polarização do vácuo

$$\Pi^{\mu\nu}(k^2) = \Pi_T\,(k^2)\left(g^{\mu\nu} - \frac{k^\mu k^\nu}{k^2}\right) + \Pi_L(k^2)\frac{k^\mu k^\nu}{k^2}$$

Podemos resolver este (trivial) conjunto de equações deste modo:

```
V k;
I mu,nu;
S projL, projT, P1,P2;
S algo,algomais;

L PolVac =  algo*d_(mu,nu) + algomais*k(mu)*k(nu);
nwrite stat;
.sort
L PiT = projT*PolVac;
L PiL = projL*PolVac;
L Check = P1*(d_(mu,nu) - k(mu)*k(nu)/k.k) + P2*k(mu)*k(nu)/k.k;
L PP1 = projT*Check;
```

```
L PP2 = projL*Check;
id projT = (d_(mu,nu) - k(mu)*k(nu)/k.k)/3;
id projL = k(mu)*k(nu)/k.k;
print;
.end
```

Note que sempre é uma boa idéia checar se a projeção faz exatamente o que se espera que ela faça.

```
bracket dx;
print;
.end
```

# 16 Conclusão

O FORM é um programa de manipulação simbólica otimizado para *grandes* problemas. No entanto, seu modelo de programação é direto e seu conhecimento vetorial também o torna oportuno para problemas cotidianos. Sua divisão em módulos permite a visualização direta do usuário sobre as outputs e seus comandos simplificados tornam-se uma opção atraente para jovens e adultos que desejam iniciar-se no mundo da Computação Algébrica.

# 17 Bibliografia

1. H.Strubbe. Comp. Phys. Comm. (1974)
2. A.c.Hearn. Reduce User's Manual. the Rand Corporation, Santa Monica, CA, 1985.
3. J.A.M.Vermaseren. The simbolic manipulation program FORM foi escrito por J.A.M.Vermaseren. A versão 1 deste programa está disponível via ftp anônimo através de ftp.nikhef.nl.
4. J.A.M. Vermaseren. FORM,1989.
5. J.A.M. Vermaseren. Symbolic Manipulation with FORM, version 2 (Computer Algebra Nederland, Amsterdam,1991)

Informações adicionais sobre o FORM podem ser obtidas através da usenet,sci.math.symbo através da web na http:\www.can.nl e http:\rulgm4.LeidenUniv.nl.

Para o FORM 2, você pode mandar e-mails para form@can.nl