Introduction to FORM

Jos Vermaseren

Part 1: The basics

How to use FORM at the NIKHEF Linux computers.
Add to your .cshrc file the line

alias form ∼form/linux/bin/form

Other executables and manuals can be found at

http://www.nikhef.nl/∼form

The Linux version is always most up to date.

FORM is a program that reads a text file containing a FORM program, executes the code it encounters and then writes the resulting output either to the screen, to a file or to both. The text file must have the extension .frm in order to be recognized as a 'FORM-file'. Hence the calling is:

```
form prog1
```

if the file prog1.frm contains the program. If the program is called with

```
form -l prog1
```

the outut will also be written to the file prog1.log. As we will see later, it is possible to select some pieces to only go to the .log file and not to the screen.

Assume that we have the file prog1.frm with the contents

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

This is a simple FORM program. We run it with the command we mentioned before and the output is

```
FORM by J.Vermaseren,version 3.1(Dec  7 2005) Run at: Tue Dec 27 14:20:25 2005
    Symbols a,b;
    Local [(a+b)^2] = (a+b)^2;
    Print;
    .end

Time =       0.00 sec    Generated terms =          3
        [(a+b)^2]        Terms in output =          3
                         Bytes used      =         54

   [(a+b)^2] =
      b^2 + 2*a*b + a^2;
```

We see here several things:

- There is a header, indicating the version and the time of the run.
- The input is listed.
- Some run time statistics are printed.
- The output is printed. FORM has expanded the formula.

Let us now look again at the little program:

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

The first statement declares the objects a and b to be symbols. The general structure of a FORM statement is a keyword, followed by some contents. The statements are terminated with a semicolon. Statements can run over many lines and their size is only limited by system dependent boundaries. More about those later.

The second statement defines `[(a+b)^2]` to be a local expression with the starting value of `(a+b)^2`. We see here that names in FORM can be either alphanumeric (starting with an alphabetic character) or any string enclosed in braces. Here the name of the expression is of the second variety.

The third statement tells FORM to print the result.

The last line is formally not a statement but it is called an instruction. The instructions that start with a period tell FORM to work out what it has and possibly print the results. The .end instruction also terminates the program.

It is now easy to make a bigger program:

```
FORM by J.Vermaseren,version 3.1(Dec  7 2005) Run at: Tue Dec 27 14:48:37 2005
    Symbols a1,...,a10;
    Local F = (a1+...+a10)^14;
    .end

Time =       0.13 sec    Generated terms =      50000
                F       1 Terms left       =      50000
                          Bytes used       =    1443406

Time =       0.30 sec    Generated terms =     100000
                F       1 Terms left       =     100000
                          Bytes used       =    2947906

Time =       0.48 sec    Generated terms =     150000
                F       1 Terms left       =     150000
                          Bytes used       =    4496926
```

```
Time =       0.65 sec    Generated terms =     200000
                F       1 Terms left      =     200000
                          Bytes used      =    6009520

Time =       0.83 sec    Generated terms =     250000
                F       1 Terms left      =     250000
                          Bytes used      =    7567886

Time =       1.01 sec    Generated terms =     300000
                F       1 Terms left      =     300000
                          Bytes used      =    9045872

Time =       1.18 sec    Generated terms =     350000
                F       1 Terms left      =     350000
                          Bytes used      =   10618212

Time =       1.37 sec    Generated terms =     400000
                F       1 Terms left      =     400000
                          Bytes used      =   12215310
```

```
Time =         1.55 sec    Generated terms =     450000
            F         1 Terms left       =     450000
                        Bytes used       =   13732872

Time =         1.73 sec    Generated terms =     500000
            F         1 Terms left       =     500000
                        Bytes used       =   15175928

Time =         1.88 sec
            F           Terms active     =     500000
                        Bytes used       =   15175866

Time =         2.04 sec    Generated terms =     550000
            F         1 Terms left       =     550000
                        Bytes used       =   16636798

Time =         2.21 sec    Generated terms =     600000
            F         1 Terms left       =     600000
                        Bytes used       =   18133144
```

```
Time =       2.39 sec    Generated terms =     650000
                F       1 Terms left      =     650000
                          Bytes used      =   19611220

Time =       2.57 sec    Generated terms =     700000
                F       1 Terms left      =     700000
                          Bytes used      =   21073000

Time =       2.74 sec    Generated terms =     750000
                F       1 Terms left      =     750000
                          Bytes used      =   22484518

Time =       2.91 sec    Generated terms =     800000
                F       1 Terms left      =     800000
                          Bytes used      =   23869160

Time =       2.97 sec    Generated terms =     817190
                F       1 Terms left      =     817190
                          Bytes used      =   24302550
```

```
Time =       3.07 sec
                F       Terms active     =     817190
                        Bytes used       =   24735854

Time =       3.28 sec   Generated terms =     817190
                F       Terms in output =     817190
                        Bytes used       =   24302462
```

We notice here lots of statistics. They give information about the run. One can also see that not all are of the same type. To understand what the statistics mean we need to understand a little bit how FORM works.

- Terms are generated one by one and they are temporarily stored in a buffer that is called the small buffer.
- When the small buffer is full, it is sorted and like terms have their coefficients added.
- After this sorting some statistics are printed.
- The sorted contents are written to the large buffer as one sorted 'patch'.

- When the large buffer is full, its patches are merged and written to disk as one sorted 'patch' in the sort-file. Statistics of the red variety are written.
- When generation of terms is finished the remaining pieces of the small and large buffers are sorted and either end up in the output file or in the sort-file.
- If there is a sort-file, its contents are merged and written to the output.
- The final statistics (blue) are printed.

In the case of our 'little' program we see that it doesn't take much time to generate an expression of 24 Mbytes.
We see also something new:

```
Symbols a1,...,a10;
Local F = (a1+...+a10)^14;
```

The ... operator takes its intuitive interpretation. Note however that it should be three dots.

In the next example we see a new type of variables:

```
Symbol a;
Functions B,C;
Local F1 = (a+B+C)^2;
Local F2 = (a+(B+C))^2;
Print;
.end
```

Functions are objects that can have arguments. Regular functions however are noncommuting objects. This means that the expressions F1 and F2 need a bit more care in their evaluation. The rule is that if there is more than one noncommuting object the evaluation of a power cannot use binomial coefficients. We will see the difference between F1 and F2 in the statistics.

Another point needs mentioning. Some keywords have abbreviations. Hence Symbols, Symbol, Sym or even S all mean the same. Actually also the keywords are case-insensitive. Hence sym or s would also do the job. Unless otherwise mentioned all system defined words or names are case-insensitive and all user defined objects are case-sensitive.

The result of the above program is (version and time line suppressed):

```
Symbol a;
Functions B,C;
Local F1 = (a+B+C)^2;
Local F2 = (a+(B+C))^2;
Print;
.end

Time =       0.00 sec    Generated terms =          9
               F1        Terms in output =          7
                         Bytes used      =        126

Time =       0.00 sec    Generated terms =          7
               F2        Terms in output =          7
                         Bytes used      =        126

   F1 =
      a^2 + 2*B*a + B*B + B*C + 2*C*a + C*B + C*C;

   F2 =
      a^2 + 2*B*a + B*B + B*C + 2*C*a + C*B + C*C;
```

We can see in the statitics that there is a difference between the evaluations of F1 and F2. This is because the evaluation is from out to in. Hence `(B+C)` is first seen as a single non-commuting object and the initial square can be worked out with binomial coefficients. Then the `(B+C)` is expanded without binomial coefficients.

One could argue that FORM should be smart enough to place these brackets by itself. The general rule however is that FORM works out the formulas in the way they are presented changing neither order nor grouping. If the user wants it differently, the user should present it differently.

This also means that `(a+b+c-a-b)^10` will be worked out 'the hard way'. There are of course ways to circumvent this, but they are a bit more advanced and hence we will keep them for later.

Let us have a look now at a bit more of the structure of a program. The next program shows us that commentary starts with the character `*` in column 1. After that one can put any characters.

There are various types of statements. They need to come in a specified order inside each module.

Modules are blocks of programs, terminated with an instruction that starts with a period. FORM will first compile the contents of a single module. After that it will execute it. Then it will forget the contents of this module and continue with the next one.

```
*
* Declarations
*
Functions f,g;
CFunctions F,G;
Symbol x;
*
* Specifications, e.g. no runtime statistics;
*
Off statistics;
```

```
 *
 * Definitions
 *
 * Local expression with only noncommuting functions
 *
 Local F1 = f(x)*g(x) + g(x)*f(x);
 *
 * Output
 *
 Print;
 *
 * End of module
 *
 .sort

F1 =
   f(x)*g(x) + g(x)*f(x);
```

```
*
* Local expression with only commuting functions
*
Local F2 = F(x)*G(x) + G(x)*F(x);
Print F2;
*
* Terminate the program
*
.end
```

```
F2 =
   2*F(x)*G(x);
```

In this program we see a new type of functions: CFunctions are commuting functions. They commute with all other objects. Non-commuting functions don't commute with other non-commuting functions, but they commute with all other objects.

The `.sort` instruction just causes execution of the current module.

If an expression is mentioned in a print statement, only that expression will be printed.

The example here shows a little bit of the flexibility of functions. A function can have any number of arguments (within system defined boundaries). An empty argument is interpreted as zero. Similarly an expression that is zero counts as having no terms.

```
Symbols x,y;
Commuting f;
Local F = f(x)+f(x,y)+f(x,,y);
Print;
.end
```

```
F =
   f(x) + f(x,y) + f(x,0,y);
```

FORM has a large number of built in functions. All built in variables have a name that ends in an underscore character. The user cannot define names with an underscore in it (unless between braces). Hence there will never be a conflict with system defined objects. Note also that, as mentioned before, the system defined objects are case-insensitive.

```
Symbol x;
*                        fac_(x) = x!   invfac_(x) = 1/fac_(x)
Local F1 = invfac_(3)+x*fac_(3);
*                        cos_ and sin_ are for now just reserved names
Local F2 = cos_(0) + cos_(x)^2 + sin_(x)^2;
*                 sign_(x) = (-1)^x    abs_(x) = |x|    sig_(x) = sign_of_x
Local F3 = x^3*sign_(3) + x*abs_(-1/2) + sig_(-3) + sig_(x);
*             binom_(x,y) = (x+y)!/x!/y!   root_(n,x) is the n-th root of x
Local F4 = binom_(5,2) + sqrt_(4) + x*root_(2,4);
Local F5 = bernoulli_(0) + bernoulli_(1)*x + bernoulli_(2)*x^2;
Local F6 = max_(1/2,2) + min_(1,x);
*                        mod_(x,n) = x modulus n
Local F7 = mod_(7,2);
```

```
 Print;
 .end

F1 =
   1/6 + 6*x;
F2 =
   sin_(x)^2 + cos_(x)^2 + cos_(0);
F3 =
    - 1 + 1/2*x - x^3 + sig_(x);
F4 =
   10 + 2*x + sqrt_(4);
F5 =
   1 + 1/2*x + 1/12*x^2;
F6 =
   2 + min_(1,x);
F7 =
   1;
```

When we take a good look at the output we see that when FORM doesn't know what to do, it doesn't do anything. Hence `min_(1,x)` is left as it is, because FORM has no way of knowing what x is.

We see also that functions like `sqrt_`, `sin_` and `cos_` are mostly reserved names. No decision has been taken yet what to do with them.

FORM knows only the rational numbers. Hence it cannot be used for arbitrary precision floating point calculations. It is not yet clear whether this will be changed in the future.

FORM knows the special symbol `i_`. Its property that `i_^2 = -1` is known to FORM. `1/i_` is replaced by `-i_`.

Next we see two more important types of variables: vectors and indices:

```
Vectors u,v;
Indices i,j;
Function f;
Local w1 = u(1) + v(i);
Local w2 = u(i) * v(j);
Local w3 = u(i) * u(i);
Local w4 = v(i) * u(i);
Local w5 = f(i,j) * u(i) * v(j);
Print;
.end
```

```
w1 = u(1) + v(i);
w2 = u(i)*v(j);
w3 = u.u;
w4 = u.v;
w5 = f(u,v);
```

If we take a good look at the output of the previous program we see that the arguments of vectors can be either (short nonnegative integer) numbers or indices.

Repeated indices are summed over. In the next example we will see how to avoid that by declaring an index zero-dimensional.

When two vectors have a common index that is summed over the result is written as a dotproduct.

When the index of a vector is contracted with the index of a function, the vector is written as the argument that used to be the index. This is called SCHOONSCHIP notation.

FORM doesn't know upper and lower indices. It assumes that when indices are contracted one must have been an upper and the other must have been a lower index.

The definition of a metric is usually only relevant when the dotproducts are given a value (like $+m^2$ or $-m^2$) or when there are uncontracted indices left in the answer. Usually the blunt approach gives the correct answers. Only in exceptional cases one has to pay close attention.

```
 Vector u;
 Index i = 0;       *  No contraction over index i
 Local P = u(i) * u(i);
 Print;
 .sort
```

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

```
 sum i;             *  Forces summing over i
 Print;
 .sort
```

```
P = u.u;
```

```
 Function f;
 Local F = f(i);
 sum i,1,3,5;       * Gives i the specified values
 Print F;
 .end
```

```
F = f(1) + f(3) + f(5);
```

In the above example we see a new feature: we can create inline commentary if we put a star as the first non-whitespace (blank or tab) character after the semicolon.

Actually FORM supports multiple statements on a line, provided they are properly separated by semicolons. Note however that we have to be careful with the end-of-module instructions and that multiple statements on one line doesn't always improve the readability of a program.

If the name of an index in its declaration is followed by an equal sign and then either a nonnegative (short) integer or a symbol, this object is to be the dimension of the index. Note that dimensions are assigned to indices, not to vectors. The index we use as an argument for a vector implicitly sets the dimension of the vector.

Actually the dimension of indices or vectors is only relevant in two cases: when objects with vectors are written out in vector components (happens very rarely in well written programs) or when one has to take the trace of a metric tensor cq. Kronecker delta. The first case the user controls and the second case FORM controls. But one can see that this second case only involves indices.

```
Tensors S,T;
Indices i,j,k,l;
Local F = S(i,k)*T(k,j) + S(i,l)*T(l,j);
Print;
.sort
```

```
F = S(i,k)*T(k,j) + S(i,l)*T(l,j);
```

```
Sum k,l;
Print;
.end
```

```
F = 2*S(i,N1_?)*T(N1_?,j);
```

In this example we first see the definition of tensors. Tensors are special functions that can have only indices or vectors for their arguments. If there is a vector it is assumed that it is the result of the contraction of an index with the index of the vector.

When we sum explicitly over contracted indices that cannot be removed (as with the contraction of the indices of vectors) FORM introduces an internal type of indices. These indices have the default dimension (we will come back to that later) and can be renamed internally. This can cause terms to become identical as in the example.

```
Symbols x,i;
Local expr = sum_(i,0,5,x^i/fac_(i));
Print;
.end
```

```
expr =
   1 + x + 1/2*x^2 + 1/6*x^3 + 1/24*x^4 + 1/120*x^5;
```

Here we see the **sum_** function. It has either four or five arguments. The last argument is the summand. The first argument must be a symbol and is the object that is summed over. The next two arguments are the start value and the upper limit. If there are five arguments the fourth argument is the increment. If the increment is absent it is supposed to be $+1$.

Of course it can happen that the range results in no values as in

```
Local expr = sum_(i,4,2,x^i/fac_(i));
```

in which case the value will be zero.

Let us take a little timeout here. You may have noticed that sometimes objects are separated by comma's and sometimes by blank spaces (or tabulator characters). Actually there is some systematics in this.

- Multiple whitespace characters are interpreted as a single blank character.
- A blank character next to an operator or natural separator like `*/+-^, =!%&|<> (){}` will be absorbed by it.
- Remaining blanks will be seen as separators and be replaced by a comma.

Hence

```
Local expr = sum_( i, 4, 2, x^i/fac_(i) ) ;
```

is exactly the same as

```
Local,expr=sum_(i,4,2,x^i/fac_(i));
```

Most of the cases that might give problems with this (like `Multiply -1`) have been programmed as exceptions. Unfortunately it wasn't possible to deal with all exceptional cases. At times one needs to place the comma explicitly.

Hence one should remember that the correct notation is that each separator is a comma, but that almost all the time blank spaces are tolerated as well. This often makes the input look prettier and more readable.

The vectors and indices are not complete without Kronecker delta's (or a metric tensor), Levi-Civita tensors and the definition of a default dimension. They are all shown below. `e_` is the Levi-Civita tensor and `d_` is the Kronecker delta. The dimension statement takes for its argument a valid dimension, ie a nonnegative (short) integer or the name of a symbol.

```
Dimension 3;
Indices i,j,k,p,q,r;
Local f0 = e_(i,j,k) * e_(p,q,r);
Local f1 = e_(i,j,k) * e_(p,q,k);
Local f2 = e_(i,j,k) * e_(p,j,k);
Local f3 = e_(i,j,k) * e_(i,j,k);
Contract;
Print +s;
.end
```

```
f0 =
     + d_(i,p)*d_(j,q)*d_(k,r)
     - d_(i,p)*d_(j,r)*d_(k,q)
     - d_(i,q)*d_(j,p)*d_(k,r)
     + d_(i,q)*d_(j,r)*d_(k,p)
     + d_(i,r)*d_(j,p)*d_(k,q)
     - d_(i,r)*d_(j,q)*d_(k,p)
    ;

f1 =
     + d_(i,p)*d_(j,q)
     - d_(i,q)*d_(j,p)
    ;

f2 =
     + 2*d_(i,p)
    ;

f3 =
     + 6
    ;
```

The contract statement contracts pairs of Levi-Civita tensors, expressing them in terms of products of Kronecker delta's.

So what happens if the dimension of the indices is not the same as the number of arguments of the Levi-Civita tensors?

If any of the indices involved has a dimension that is not in agreement with the number of indices in the Levi-Civita tensors, no shortcuts are taken. This means that we use the expression with n! terms if the Levi-Civita tensors have n indices.

The resulting formula may then have Kronecker delta's like `d_(i,i)` which then are replaced by the dimension of the index i. We see this at work in the following example. The +s option in the print statement forces the output to be printed one term per line.

```
Symbol n;
Dimension n;
Indices i,j,k,p,q,r;
Local f0 = e_(i,j,k) * e_(p,q,r);
Local f1 = e_(i,j,k) * e_(p,q,k);
Local f2 = e_(i,j,k) * e_(p,j,k);
Local f3 = e_(i,j,k) * e_(i,j,k);
Contract;
Print +s;
.end
```

```
f0 =
    + d_(i,p)*d_(j,q)*d_(k,r)
    - d_(i,p)*d_(j,r)*d_(k,q)
    - d_(i,q)*d_(j,p)*d_(k,r)
    + d_(i,q)*d_(j,r)*d_(k,p)
    + d_(i,r)*d_(j,p)*d_(k,q)
    - d_(i,r)*d_(j,q)*d_(k,p);
f1 =
    - 2*d_(i,p)*d_(j,q)
    + d_(i,p)*d_(j,q)*n
    + 2*d_(i,q)*d_(j,p)
    - d_(i,q)*d_(j,p)*n;
f2 =
    + 2*d_(i,p)
    - 3*d_(i,p)*n
    + d_(i,p)*n^2;
f3 =
    + 2*n
    - 3*n^2
    + n^3;
```

One should be very careful about mixing indices with different dimensions. The outcome can become unpredictable. Take for instance

```
Indices i3 = 3, i4 = 4;
Local F = d_(i3,i4)*d_(i3,i4);
```

The outcome depends on which index is contracted first. If i3 is contracted first, FORM will encounter `d_(i4,i4)` which will become 4. If on the other hand i4 is contracted first, FORM will encounter `d_(i3,i3)` and the answer will become 3. And this all depends on how FORM proceeds internally, which again depends (amoung other things) on the order in which i3 and i4 were declared.

Because the exact order in which FORM works things out internally is not part of the specification of the language, one should try to avoid such constructions!

Levi-Civita tensors can be used in cross products:

```
Dimension 3;
Vectors u,v,w;
Indices i,j,k,l,m,n;
Local [uxv] = e_(i,j,k) * u(i) * v(j);
Local [uxv.w] = e_(i,j,k) * u(i) * v(j) * w(k);
Local [ux(vxw)] = e_(i,j,k) * u(i) * (e_(m,n,j)*v(m)*w(n));
Contract;
Print;
.end
```

```
[uxv] =
   e_(u,v,k);

[uxv.w] =
   e_(u,v,w);

[ux(vxw)] =
   v(k)*u.w - w(k)*u.v;
```

Another place in which Levi-Civita tensors can be useful is in the evaluation of determinants. There are many ways in which this can be done. We start with a rather direct one:

```
CFunction M;
Indices i,j;
Local det = e_(1,2) * e_(i,j) * M(1,i) * M(2,j);
Contract;
Print;
.end
```

```
det =
   M(1,1)*M(2,2) - M(1,2)*M(2,1);
```

Gram determinants are a special type of determinants in which the elements of the Gram matrix are given by $M_{ij} = v_i \cdot w_j$. The v_i and w_j are each n different vectors if n x n is the size of the matrix. We generate the determinant in a very simple manner:

```
 Vectors v1,v2,v3;
 Vectors w1,w2,w3;
 Local H2 = e_(v1,v2)*e_(w1,w2);
 Local H3 = e_(v1,v2,v3)*e_(w1,w2,w3);
 Contract;
 Print +s;
 .end
H2 =
    + v1.w1*v2.w2
    - v1.w2*v2.w1;
H3 =
    + v1.w1*v2.w2*v3.w3
    - v1.w1*v2.w3*v3.w2
    - v1.w2*v2.w1*v3.w3
    + v1.w2*v2.w3*v3.w1
    + v1.w3*v2.w1*v3.w2
    - v1.w3*v2.w2*v3.w1;
```

It is also not difficult to derive the famous Schouten identity. We work here in 4 dimensions (the default). Hence a Levi-Civita tensor with 5 indices is zero. Hence the contraction of two such objects is on the one hand zero, and on the other hand gives us 120 terms with each 5 Kronecker delta's. When all indices that need to be contracted are indeed contracted we obtain the famous identity (F is zero):

```
I   n1,...,n5,m1,...,m6;
L   F = e_(m1,m2,m3,m4)*d_(m5,m6);
Multiply e_(m1,m2,m3,m4,m5)*e_(n1,n2,n3,n4,n5)/24;
Contract;
Print +s;
.end
```

```
  F =
      + e_(n1,n2,n3,n4)*d_(n5,m6)
      - e_(n1,n2,n3,n5)*d_(n4,m6)
      + e_(n1,n2,n4,n5)*d_(n3,m6)
      - e_(n1,n3,n4,n5)*d_(n2,m6)
      + e_(n2,n3,n4,n5)*d_(n1,m6)
     ;
```

Just for fun, we show that a 10 x 10 Gram determinant is no problem:

```
Vector v1,...,v10;
On Statistics;
Local G10 = e_(v1,...,v10)^2;
Contract;
.end
```

```
Time =        0.06 sec    Generated terms =      36240
               G10      1 Terms left      =      17880
                          Bytes used      =     677260

Time =        0.14 sec    Generated terms =      72562
               G10      1 Terms left      =      45929
                          Bytes used      =    1733236

                         :
                         :
                         :
```

```
                              :
                              :
                              :

Time =        9.77 sec    Generated terms =    3628800
               G10      1 Terms left      =    2622117
                          Bytes used      =   91137406

Time =        9.83 sec
               G10        Terms active    =    2615813
                          Bytes used      =   91091304

Time =       10.84 sec    Generated terms =    3628800
               G10        Terms in output =    1436714
                          Bytes used      =   50113622
```

There is another important function connected to vectors and tensors. It is called `dd_` and it is defined by

$$
\begin{aligned}
dd\_(i_1,i_2) &= d\_(i_1,i_2) \\
dd\_(i_1,i_2,i_3,i_4) &= d\_(i_1,i_2)d\_(i_3,i_4) + d\_(i_1,i_3)d\_(i_2,i_4) + d\_(i_1,i_4)d\_(i_2,i_3) \\
dd\_(i_1,\cdots,i_n) &= d\_(i_1,i_2)dd\_(i_3,\cdots,i_n) + \cdots + d\_(i_1,i_n)dd\_(i_2,\cdots,i_{n-1})
\end{aligned}
$$

The importance of this tensor concerns contractions with vectors. When there are identical vectors, many terms that would be generated by the above definition would be the same. FORM uses some internal combinatorics to generate each resulting term only once with the proper coefficient.

```
    Indices i1,...,i6;
    Vectors p1,p2;
    Local F1 = dd_(i1,i2,i3,i4,i5,i6);
    .sort

Time =       0.00 sec    Generated terms =          15
               F1        Terms in output =          15
                         Bytes used      =         266
```

```
    Multiply p1(i1)*p2(i2)*p1(i3)*p2(i4)*p1(i5)*p1(i6);
    Print;
    .sort

Time =       0.00 sec    Generated terms =         15
               F1        Terms in output =          2
                         Bytes used      =         44

   F1 =
      12*p1.p1*p1.p2^2 + 3*p1.p1^2*p2.p2;

    Drop;
    Local F1 = dd_(p1,p2,p1,p2,p1,p1);
    Print;
    .end
```

```
Time =       0.00 sec    Generated terms =          2
               F1         Terms in output =          2
                          Bytes used      =         44

   F1 =
      12*p1.p1*p1.p2^2 + 3*p1.p1^2*p2.p2;
```

As we can see, the answer is exactly the same, but in the second case only two terms were generated.

The drop-statement removes all previous expressions. Hence F1 is forgotten from this point on.

The multiply-statement multiplies all terms in all currently active expressions by the expression that is specified in it.

The final and most complicated special function we will treat in this lecture is the `distrib_` function. For any action it needs at least 4 arguments. Let us assume there are 4+n arguments.

The third and fourth arguments should be the names of two functions. The function will generate terms in which the n arguments will be distributed over the two functions.

The second argument should be a non-negative integer. It indicates that one of the two functions should have exactly the specified number of arguments.

If the first argument is either 1 or -1, the second argument refers to the first function. If the first argument is 2 or -2, the second argument refers to the second function. For a negative value it is assumed that each change in the order of the combined arguments results in a minus sign.

If the first argument is zero, all possible distributions are made. If the second argument is negative, it is assumed that each change in the order of the combined arguments results in a minus sign.

Of course, an example is best:

```
Symbols x1,x2,x3,x4,x5;
CFunctions f1,f2;
Local F = distrib_(-1,2,f1,f2,x1,x2,x3,x4,x5);
Print;
.end
```

```
Time =        0.00 sec    Generated terms =          10
                 F        Terms in output =          10
                          Bytes used      =         324

   F =
      f1(x1,x2)*f2(x3,x4,x5) - f1(x1,x3)*f2(x2,x4,x5) + f1(x1,x4)*f2(x2,x3,x5)
       - f1(x1,x5)*f2(x2,x3,x4) + f1(x2,x3)*f2(x1,x4,x5) - f1(x2,x4)*f2(x1,x3,
      x5) + f1(x2,x5)*f2(x1,x3,x4) + f1(x3,x4)*f2(x1,x2,x5) - f1(x3,x5)*f2(x1,
      x2,x4) + f1(x4,x5)*f2(x1,x2,x3);
```

Also here there are some provisions for symmetries, but the order of the arguments is left untouched as much as possible.

The arguments can be any type. Of course, if either f1 or f2 is a tensor, the arguments should be valid indices or vectors only.