Introduction to FORM

Jos Vermaseren

Part 3: Some mixed topics and simple examples

Many keywords in FORM can be abbreviated. This makes for easier typing, although it may not make the program more readable. Actually in those commands one can drop any number of characters from the end till one is left with the 'root' of the command. Hence

I[ndex] means that the full command is Index, and its root is I.

Id[entify] means that the full command is Identify, and its root is Id.

Other popular ones are S[ymbols], V[ectors], CF[unctions], F[unctions], T[ensors], B[rackets], L[ocal], G[lobal], M[ultiply] and P[rint]. In the reference manual of each keyword is mentioned whether it has abbreviations and what its root is.

Note also that of course the keywords are case insensitive.

Let us try this out on a program with a new command:

```
V   Q,p1,p2,p3;
T   Qtensor;
L   F = Q.p1*Q.p2^2*Q.p3^3*Q.Q;
ToTensor Q,Qtensor;
Print;
.end
```

```
  F =
     Qtensor(p1,p2,p2,p3,p3,p3,N1_?,N1_?);
```

Effectively we have

$$\mathrm{Qtensor}^{\mu_1\cdots\mu_n} = Q^{\mu_1\cdots\mu_n}$$

in which Qtensor is just some tensor (the name could have been any name but we selected something that refers to its use. The two dummy indices represent the $Q\cdot Q$.

Basically the ToTensor statement is just a change of notation.

The above command is very useful when we are confronted with d'Alembertians. The d'Alembertian w.r.t. the vector Q is defined by:

$$\Box = \frac{\partial}{\partial Q_\mu}\frac{\partial}{\partial Q^\mu}$$

Hence, if we have in the following program 4 powers of the d'Alembertian, we should eliminate 8 powers of Q. This would give the program:

```
V   Q,p1,p2;
L   F = Q.p1^3*Q.p2^5;
Totensor,Q,dd_;
Print +f;
.end
```

```
F =
     + 45*p1.p1*p1.p2*p2.p2^2 + 60*p1.p2^3*p2.p2;
```

Try to verify this!

In the previous example we saw the simultaneous use of the ToTensor statement and the `dd_` function. This can be very efficient as we see below:

```
V   Q,p1,p2,p3,p4;
L   F = Q.p1^7*Q.p2^9*Q.p3^11*Q.p4^13;
ToTensor Q,dd_;
.end
```

```
Time =        0.03 sec    Generated terms =       4940
                 F         Terms in output =       4940
                           Bytes used      =     265070
```

Taking a total of 40 derivatives will usually result in much more work and almost always in a great duplication of terms.

In the next example we will have the number of d'Alembertians such that we don't eliminate all powers of Q.

```
 Vectors Q,p1,p2,p3;
 Tensors t1,t2;
 Symbol Dalemb,n;
 Off Statistics;
 *
 *  Ten powers of Q, three d'Alembertians
 *
 Local F = Dalemb^3*Q.p1^3*Q.p2^5*Q.p3^2;
 Totensor Q,t1;
 Print;
 .sort

F =
   t1(p1,p1,p1,p2,p2,p2,p2,p2,p3,p3)*Dalemb^3;
```

Next we use the distrib_ function to pull out the vectors that combine with the Q that will be taken out by the d'Alembertians. Notice how the wildcards are being used.

```
*
*    Pull 2*n powers out in all possible ways
*    Those go into t1, the remainder into t2.
*
id  Dalemb^n?*t1(?a) = distrib_(1,2*n,t1,t2,?a);
Print +s;
.sort
```

The result is shown below. Notice that the distrib_ function can handle the combinatorics when adjacent arguments are identical.

```
F =
      + 10*t1(p1,p1,p1,p2,p2,p2)*t2(p2,p2,p3,p3)
      + 20*t1(p1,p1,p1,p2,p2,p3)*t2(p2,p2,p2,p3)
      + 5*t1(p1,p1,p1,p2,p3,p3)*t2(p2,p2,p2,p2)
      + 15*t1(p1,p1,p2,p2,p2,p2)*t2(p1,p2,p3,p3)
      + 60*t1(p1,p1,p2,p2,p2,p3)*t2(p1,p2,p2,p3)
      + 30*t1(p1,p1,p2,p2,p3,p3)*t2(p1,p2,p2,p2)
      + 3*t1(p1,p2,p2,p2,p2,p2)*t2(p1,p1,p3,p3)
      + 30*t1(p1,p2,p2,p2,p2,p3)*t2(p1,p1,p2,p3)
      + 30*t1(p1,p2,p2,p2,p3,p3)*t2(p1,p1,p2,p2)
      + 2*t1(p2,p2,p2,p2,p2,p3)*t2(p1,p1,p1,p3)
      + 5*t1(p2,p2,p2,p2,p3,p3)*t2(p1,p1,p1,p2)
     ;
```

To write the remaining powers back as contractions with Q, we use the ToVector statement, which is exactly the opposite of the ToTensor statement. Note that the order of the two arguments is not relevant. One has to be a vector, while the other must be a tensor.

```
*
*   Write the remaining powers back. We do that first before we create
*   more terms with dd_. This way the program is more efficient.
*
ToVector,Q,t2;
Print +s;
.sort
```

One can see the result below.

```
F =
     + 10*t1(p1,p1,p1,p2,p2,p2)*Q.p2^2*Q.p3^2
     + 20*t1(p1,p1,p1,p2,p2,p3)*Q.p2^3*Q.p3
     + 5*t1(p1,p1,p1,p2,p3,p3)*Q.p2^4
     + 15*t1(p1,p1,p2,p2,p2,p2)*Q.p1*Q.p2*Q.p3^2
     + 60*t1(p1,p1,p2,p2,p2,p3)*Q.p1*Q.p2^2*Q.p3
     + 30*t1(p1,p1,p2,p2,p3,p3)*Q.p1*Q.p2^3
     + 3*t1(p1,p2,p2,p2,p2,p2)*Q.p1^2*Q.p3^2
     + 30*t1(p1,p2,p2,p2,p2,p3)*Q.p1^2*Q.p2*Q.p3
     + 30*t1(p1,p2,p2,p2,p3,p3)*Q.p1^2*Q.p2^2
     + 2*t1(p2,p2,p2,p2,p2,p3)*Q.p1^3*Q.p3
     + 5*t1(p2,p2,p2,p2,p3,p3)*Q.p1^3*Q.p2
    ;
```

```
*
*   And finally the dd_:
*
id  t1(?a) = dd_(?a);
Print +s;
.end
```

```
F =
    + 45*Q.p1*Q.p2*Q.p3^2*p1.p1*p2.p2^2
    + 180*Q.p1*Q.p2*Q.p3^2*p1.p2^2*p2.p2
    + 180*Q.p1*Q.p2^2*Q.p3*p1.p1*p2.p2*p2.p3
    + 360*Q.p1*Q.p2^2*Q.p3*p1.p2*p1.p3*p2.p2
    + 360*Q.p1*Q.p2^2*Q.p3*p1.p2^2*p2.p3
    + 30*Q.p1*Q.p2^3*p1.p1*p2.p2*p3.p3
    + 60*Q.p1*Q.p2^3*p1.p1*p2.p3^2
    + 240*Q.p1*Q.p2^3*p1.p2*p1.p3*p2.p3
```

```
+ 60*Q.p1*Q.p2^3*p1.p2^2*p3.p3
+ 60*Q.p1*Q.p2^3*p1.p3^2*p2.p2
+ 360*Q.p1^2*Q.p2*Q.p3*p1.p2*p2.p2*p2.p3
+ 90*Q.p1^2*Q.p2*Q.p3*p1.p3*p2.p2^2
+ 90*Q.p1^2*Q.p2^2*p1.p2*p2.p2*p3.p3
+ 180*Q.p1^2*Q.p2^2*p1.p2*p2.p3^2
+ 180*Q.p1^2*Q.p2^2*p1.p3*p2.p2*p2.p3
+ 45*Q.p1^2*Q.p3^2*p1.p2*p2.p2^2
+ 60*Q.p1^3*Q.p2*p2.p2*p2.p3^2
+ 15*Q.p1^3*Q.p2*p2.p2^2*p3.p3
+ 30*Q.p1^3*Q.p3*p2.p2^2*p2.p3
+ 90*Q.p2^2*Q.p3^2*p1.p1*p1.p2*p2.p2
+ 60*Q.p2^2*Q.p3^2*p1.p2^3
+ 120*Q.p2^3*Q.p3*p1.p1*p1.p2*p2.p3
+ 60*Q.p2^3*Q.p3*p1.p1*p1.p3*p2.p2
+ 120*Q.p2^3*Q.p3*p1.p2^2*p1.p3
```

```
 + 15*Q.p2^4*p1.p1*p1.p2*p3.p3
 + 30*Q.p2^4*p1.p1*p1.p3*p2.p3
 + 30*Q.p2^4*p1.p2*p1.p3^2
;
```

To obtain such a result by different means is usually much more elaborate and involves much duplicity.

Just for reference we will show the complete program now without the intermediate .sort instructions and without printing the output, but with the statistics:

```
Vectors Q,p1,p2,p3;
Tensors t1,t2;
Symbol Dalemb,n;
Local F = Dalemb^3*Q.p1^3*Q.p2^5*Q.p3^2;
Totensor Q,t1;
id  Dalemb^n?*t1(?a) = distrib_(1,2*n,t1,t2,?a);
ToVector,Q,t2;
id  t1(?a) = dd_(?a);
.end
```

```
Time =       0.00 sec    Generated terms =         27
                F        Terms in output =         27
                         Bytes used      =        902
```

As one can see, there is absolutely no duplicity here. This goes similar for much wilder expressions:

```
Vectors Q,p1,p2,p3,p4;
Tensors t1,t2;
Symbol Dalemb,n;
Local F = Dalemb^20*Q.p1^12*Q.p2^14*Q.p3^16*Q.p3^18;
Totensor Q,t1;
id  Dalemb^n?*t1(?a) = distrib_(1,2*n,t1,t2,?a);
ToVector,Q,t2;
id  t1(?a) = dd_(?a);
.end
```

```
Time =        0.09 sec    Generated terms =      10599
                 F        Terms in output =      10599
                          Bytes used      =     667780
```

As one can see, we had to go to rather large values to get a running time that is measurable. If you are very familiar with other systems, it would be interesting to compare with their performance for this kind of operations.

Next we turn to a different topic. We will see a form of the print statement that can be very useful in debugging a FORM program, and simultaneously we will see how FORM works through a module.

A print statement with a text string prints that string, each time FORM passes this statement during execution. This means for each term that is being treated by the statements. If the object `%t` is part of the string, the current term will be printed in that position in the string:

```
    Symbols a,b;
    Local F = (a+b)^2;
    Print "text: %t";
    Print;
    .end
text:  + a^2
text:  + 2*a*b
text:  + b^2
   F =
      b^2 + 2*a*b + a^2;
```

We see that when `(a+b^2` is worked out, first the `a^2` is generated, then the `2*a*b` and finally the `b^2`. One by one these terms run into the end of the module and are then sent to the sorting routines. Next we put a slightly more complicated example:

```
Symbols a,b;
Local F = (a+b)^2;
Print "<1> %t";
Multiply (a-b)^2;
Print "  <2> %t";
Print +s;
.end
```

```
<1>  + a^2
  <2>  + a^4
  <2>  - 2*a^3*b
  <2>  + a^2*b^2
<1>  + 2*a*b
  <2>  + 2*a^3*b
  <2>  - 4*a^2*b^2
  <2>  + 2*a*b^3
<1>  + b^2
  <2>  + a^2*b^2
  <2>  - 2*a*b^3
  <2>  + b^4
```

```
Time =         0.00 sec    Generated terms =          9
                  F        Terms in output =          3
                           Bytes used      =         54

   F =
       + b^4
       - 2*a^2*b^2
       + a^4
      ;
```

A careful study of this example shows that the `a^2` term that is generated first is next multiplied by `(a-b)^2` which is then expanded. After all those terms have been generated and sent off to the sorting routines FORM falls back to generating the term `2*a*b` and will multiply that one by `(a-b)^2`. Etc.

We put the strings `<1>` and `<2>` to distinguish the print statements from each other. There is no significance in how we do this.

The next new statement that is at times very valuable is the SplitArg statement. It has a variety of options for selecting the function and the arguments on which it should operate. These can be looked up in the manual. We will use here only one special option.

```
 Symbols a,b,c;
 CFunction f;
 L   F = f(a+b+3*c+1/a);
 Print;
 .sort
F =
   f(a^-1 + 3*c + b + a);
 SplitArg,f;
 Print;
 .end
F =
   f(a^-1,3*c,b,a);
```

We see that the different terms in the argument of f are all separated and put in individual arguments. We could have specified more functions. If no functions are specified, all functions are taken, including the built in functions. Notice that FORM has first brought the argument to 'normal form'. This explains why the order of the arguments is different from the way we typed the terms originally. The exact order is determined by the order of declaration of the symbols a,b,c:

```
 Symbols c,b,a;
 CFunction f;
 L   F = f(a+b+3*c+1/a);
 Print;
 .sort
F =
   f(a^-1 + a + b + 3*c);
 SplitArg,f;
 Print;
 .end
F =
   f(a^-1,a,b,3*c);
```

The important option here is ((a)), which indicates that we are only interested in terms that contain positive powers of a. (a) on the other hand indicates that only terms which contain multiples of a are taken separately:

```
Symbols a,b,c;
CFunction f,g;
L   F = f(a+b+3*c+2*a^2*b+1/a) + g(a+b+3*c+2*a^2*b+1/a);
SplitArg,((a)),f;
SplitArg,(a),g;
Print;
.end
```

```
F =
   f(a^-1 + 3*c + b,a,2*a^2*b) + g(a^-1 + 3*c + b + 2*a^2*b,a);
```

We see that in the case of the function g, only the term in a has been taken apart.

We can use this feature for making a facility for splitting fractions. Assume that the function `den(x)` stands for `1/x`. We will also need some new features in the wildcarding as the next example shows:

```
Symbols x,a,b,x1,x2;
CFunction den;
L   F = den(x+a)*den(x+b);
L   G = den(x+a)*den(x+a);
SplitArg,((x)),den;
id  den(x1?,x)*den(x2?,x) = (den(x1,x)-den(x2,x))*den(x2-x1);
id den(x1?,x2?) = den(x1+x2);
Print;
.end
```

```
F =
    - den(b + x)*den(b - a) + den(a + x)*den(b - a);

G = 0;
```

We can see here that somehow we have to exclude the case that $x_1 = x_2$.

As we see from the previous example we have to be able to restrict matches in the wildcarding. This is done with a new type of variables: sets. A set is a collection of objects of the same type (we can however mix symbols and numbers). They are declared and used as follows:

```
S   x,y,x1,...,x5,y1,...,y5;
Set xx:x1,...,x5;
CF  f;
L   F = f(x1)+f(x2)+f(x5)-f(y1)-f(y2)-f(y5);
id f(x?xx) = f(x,x);
Print;
.end
```

```
  F =
     f(x1,x1) + f(x2,x2) + f(x5,x5) - f(y1) - f(y2) - f(y5);
```

We see that the set is attached to the questionmark. It indicates that only elements of the set will be used for a match.

The opposite is also possible. One can exclude members of a set as in

```
S   x,y,x1,...,x5,y1,...,y5;
Set xx:x1,...,x5;
CF  f;
L   F = f(x1)+f(x2)+f(x5)-f(y1)-f(y2)-f(y5);
id f(x?!xx) = f(x,x);
Print;
.end
```

```
F =
   f(x1) + f(x2) + f(x5) - f(y1,y1) - f(y2,y2) - f(y5,y5);
```

Set can also be used as some type of array. In the next example the parameter n obtains the number of the set element that matches. This number starts counting from 1.

```
S   x,y,n,x1,...,x5,y1,...,y5;
Set xx:x1,...,x5;
CF  f;
L   F = f(x1)+f(x2)+f(x5)-f(y1)-f(y2)-f(y5);
id f(x?xx[n]) = f(x,x,n);
Print;
.end
```

```
F =
   f(x1,x1,1) + f(x2,x2,2) + f(x5,x5,5) - f(y1) - f(y2) - f(y5);
```

We see here that n could be used in the RHS of the substitution. One should not use a quastionmark on the n in the LHS as it is already clear that it is a wildcard.

One can use elements of the set in the RHS as well.

```
S   x,y,n,x1,...,x5,y1,...,y5;
CF  f,f1,...,f5;
Set xx:x1,...,x5;
Set yy:y1,...,y5;
Set ff:f1,...,f5;
L   F = f(x1)+f(x2)+f(x5)-f(y1)-f(y2)-f(y5);
id f(x?xx[n]) = ff[n](yy[n]);
Print;
.end
```

```
F =
   - f(y1) - f(y2) - f(y5) + f1(y1) + f2(y2) + f5(y5);
```

One can also define sets 'on the fly' by enclosing the elements between curly brackets as in:

```
S   x,y,n,x1,...,x5,y1,...,y5;
CF  f,f1,...,f5;
L   F = f(x1)+f(x2)+f(x5)-f(y1)-f(y2)-f(y5);
id f(x?{x1,x2,x3,x4,x5}[n]) = {f1,f2,f3,f4,f5}[n]({y1,...,y5}[n]);
Print;
.end
```

```
  F =
     - f(y1) - f(y2) - f(y5) + f1(y1) + f2(y2) + f5(y5);
```

This notation is easier for sets that have relatively few elements and are used only once. We are now ready to return to the program for splitting the fractions.

We use now the exclusion of sets. But these sets contain wildcards:

```
Symbols x,a,b,x1,x2;
CFunction den;
L   F = den(x+a)*den(x+b);
L   G = den(x+a)*den(x+a);
SplitArg,((x)),den;
id  den(x1?!{x2?},x)*den(x2?!{x1?},x) = (den(x1,x)-den(x2,x))*den(x2-x1);
id den(x1?,x2?) = den(x1+x2);
Print;
.end
```

```
F =
    - den(b + x)*den(b - a) + den(a + x)*den(b - a);

G =
   den(a + x)^2;
```

This time it works properly. From the mathematical viewpoint we need only to provide one exclusive set, but as we don't know which match is made first by FORM and which match last (which is the one that needs the restriction), we put the restriction on both.

There are some built in sets. The manual gives a complete list.

```
Symbols x,x1,x2;
CFunction den;
L   F = den(x+1)*den(x+2)*den(x+3)*den(x+4);
SplitArg,((x)),den;
repeat;
  id  den(x1?!{x2?},x)*den(x2?!{x1?},x) = (den(x1,x)-den(x2,x))*den(x2-x1);
endrepeat;
id den(x?number_) = 1/x;
id den(x1?,x2?) = den(x1+x2);
Print;
.end
```

```
 F =
    1/6*den(1 + x) - 1/2*den(2 + x) + 1/2*den(3 + x) - 1/6*den(4 + x);
```

The set number_ is the set of all rational numbers.

The sets in the wildcarding introduces already a conditional element in the execution of programs, but this is rather incomplete. To be complete we need something like an if-statement. And in the case of symbolic processing, the question is what conditions can be used in the if-statement.

```
Symbols a,b,c;
L   F = (a+b)^4;
if ( count(a,1) > 2 );
  id  a = c;
endif;
Print;
.end
```

```
 F =
    c^4 + 4*b*c^3 + b^4 + 4*a*b^3 + 6*a^2*b^2;
```

The count condition has pairs of arguments. It is some form of power counting. First comes an object and then its 'dimension'. This 'dimension' can be negative, but it must be integer. There can be as many pairs as one likes.

Also here there is a shorthand notation if there is only an if-statement (and no else or elseif statements) and the content is only a single statement as in:

```
Symbols a,b,c;
L   F = (a+b)^4;
if ( count(a,1) > 2 ) id a = c;
Print;
.end

 F =
    c^4 + 4*b*c^3 + b^4 + 4*a*b^3 + 6*a^2*b^2;
```

One can nest repeat and if-statements. One should however realize that the complete construction should be inside a single module. As the statements act on terms only and the .sort creates complete expressions they cannot mix. We will see constructions that can include .sort instructions later when we deal with the preprocessor.

The second conditional is match(pattern). This returns the number of times that the pattern matches.

```
Symbols a,b,c;
L   F = (a+b)^4;
if ( match(a*b) ) id a = c;
Print;
.end
```

```
  F =
     4*b*c^3 + 6*b^2*c^2 + 4*b^3*c + b^4 + a^4;
```

If we don't put a compare with a number the test is whether the object is unequal to zero. This is as in the language C.

The third conditional is coeff which is the size of the coefficient of the current term.

```
Symbols a,b,c;
L   F = (a+b)^4;
if ( coeff > 4 ) id a = c;
Print;
.end
```

```
F =
   6*b^2*c^2 + b^4 + 4*a*b^3 + 4*a^3*b + a^4;
```

Rather than specifying a number one can also use the multipleof(number) object:

```
Symbols a,b,c;
L   F = (a+b)^4;
if ( coeff == multipleof(3) ) id a = c;
Print;
.end

F =
   6*b^2*c^2 + b^4 + 4*a*b^3 + 4*a^3*b + a^4;
```

The last important conditional is the expression(name(s)). It determines in which expressions the condition is taken when there more expressions active at the same time:

```
Symbols a,b,c;
L   F = (a+b)^4;
L   G = (a+b)^4;
if ( expression(F) ) id a = c;
Print;
.end

 F =
    c^4 + 4*b*c^3 + 6*b^2*c^2 + 4*b^3*c + b^4;

 G =
    b^4 + 4*a*b^3 + 6*a^2*b^2 + 4*a^3*b + a^4;
```

Other objects are more rare and can be found in the reference manual.

Finally one can combine conditions. Be however careful and place the parentheses properly or unexpected things can happen.

```
Symbols a,b,c;
L   F = (a+b)^4;
if ( ( coeff == multipleof(2) ) && ( count(a,1) != 2 ) ) id a = c;
Print;
.end
```

```
F =
   4*b*c^3 + 4*b^3*c + b^4 + 6*a^2*b^2 + a^4;
```