Introduction to FORM

Jos Vermaseren

Part 4: The preprocessor and $-variables.

Thus far we have seen the manipulation of terms in expressions. Actually this doesn't give full flexibility yet. We have to be able to manipulate the program itself. We must have subroutines, loops and other control mechanisms. And above all, we must have string variables. This is all controled by the preprocessor.

Preprocessor commands are called instructions. They start with the character #, followed by a keyword. The preprocessor manipulates the input, usually at the string level. But this manipulation can be based on results from previous parts of the program.

The preprocessor has also variables. In principle these variables contain character strings. When they are used, we distinguish them from the symbolic variables by enclosing them in a matching backquote (`), quote (') pair. These act as a kind of parentheses and can be nested.

```
S   x;
L   F = x;
#do i = 1,5
  id  x = x+1;
#enddo
Print;
.end
```

```
F =
   5 + x;
```

Actually this is clearer in the next example:

```
CF  f;
L   F = f;
#do i = 1,5
  id  f(?a) = f('i',?a);
#enddo
Print;
.end

 F =
    f(5,4,3,2,1);
```

The use of the do-loop is as in fortran. If we specify a third variable in the RHS of the = sign, it will be the increment. If there is no increment it is supposed to be one.

There is a second way to use the do-loop. This is called a 'listed' loop. We list the string values that the loop parameter should take. This list is enclosed in a pair of curly brackets and the elements are separated by comma's (in an old notation by the | character).

```
S   a,b,c;
CF  f;
L   F = f;
#do i = {a,b,c}
  id  f(?a) = f('i',?a)+1;
#enddo
Print;
.end

  F =
     3 + f(c,b,a);
```

If we like to include a comma in the string we should 'escape' it with the character \ in front:

```
S    a,b,c;
CF   f;
L    F = f;
#do i = {a\,b,c}
  id  f(?a) = f('i',?a)+1;
#enddo
Print;
.end

   F =
      2 + f(c,a,b);
```

It is of course up to the user to make sure that the resulting strings that are substituted can be interpreted properly by the symbolic part of the program.

One can also define preprocessor variables directly. This is done with the #define and/or #redefine instructions as in:

```
#define a1 "x1"
#define a2 "x2"
#define a3 "x3"
Symbols x1,x2,x3;
CFunction f;
Local F =
#do i = 1,3
  + f(`a`i'')
#enddo
    ;
Print;
.end

F =
   f(x1) + f(x2) + f(x3);
```

Note that unlike in the language C, `F = +1;` is allowed in FORM. C does not accept the leading + sign.

Preprocessor variables are stored in a stack. Hence they can be defined several times and FORM will use the last definition. Under some circumstances the stack will be popped and the older definition will become active again.

```
#define i "-5*a"
Symbol a;
CF f;
#do i = 1,2
L   F`i' = f(`i');
#enddo
L   F = f(`i');
Print;
.end
```

```
F1 = f(1);

F2 = f(2);

F = f( - 5*a);
```

If one would like to overwrite an existing definition one should use the #redefine instruction. We will see examples of this later when we treat the procedures.

If necessary one can also remove preprocessor variables from the list by the use of the #undefine instruction.

If one would like to inspect which preprocessor variables have been defined and what their values are one can use the #show instruction:

```
    #define i "-5*a"
    Symbol a;
    CF f;
    #do i = 1,1
    #show
#The preprocessor variables:
0: VERSION_ = "3"
1: SUBVERSION_ = "1"
2: NAMEVERSION_ = ""
3: DATE_ = "Tue Jan 10 13:24:52 2006"
4: PARALLELTASK_ = "0"
5: NPARALLELTASKS_ = "1"
6: NAME_ = "progs4/ex1.frm"
7: NTHREADS_ = "1"
```

```
8: CMODULE_ = "1"
9: i = "-5*a"
10: i = "1"
    L F‘i’ = f(‘i’);
    #enddo
    L F = f(‘i’);
    Print;
    .end

   F1 = f(1);

   F = f( - 5*a);
```

One gets to see all variables, including the built in ones. We can see the double definition of the variable i.

It should be clear that this facility can be very useful for debugging.

If we want to know the contents of a single variable, or we like to have a message printed when the program reaches a given point, we have the #message instruction. It prints everything that follows, till the end of the line, as interpreted by the preprocessor. Hence preprocessor variables are substituted.

```
    #define i "-5*a"
    Symbol a;
    CF f;
    #message The value of i is ‘i’
~~~The value of i is -5*a
    #do i = 1,3
    L F‘i’ = f(‘i’);
    #message The value of i is ‘i’
~~~The value of i is 1
    #enddo
~~~The value of i is 2
~~~The value of i is 3
    L F = f(‘i’);
    #message The value of i is ‘i’
~~~The value of i is -5*a
    .end
```

The lines starting with ~~~ are the results of the printing of the message. Note that the contents of the loop are listed only once, but the message is printed each time we go through the loop.

The next useful feature, which is present in nearly any programming language, is the #include instruction. The syntax is rather simple. Let us assume we have a file decl.h which contains

```
Symbols x1,...,x10;
Symbols y1,...,y10;
CFunctions f1,...,f10;
```

while the program contains:

```
#include decl.h
L  F = (x1+x2+f3(x3))^2;
Print;
.end
```

Running this program gives:

```
#include decl.h
Symbols x1,...,x10;
Symbols y1,...,y10;
CFunctions f1,...,f10;
L  F = (x1+x2+f3(x3))^2;
Print;
.end

 F =
    x2^2 + 2*x1*x2 + x1^2 + 2*f3(x3)*x2 + 2*f3(x3)*x1 + f3(x3)^2;
```

Notice that the contents of the include file are listed in the output. If we don't want that we should specify a minus sign, appended to the #include as in:

```
#include- decl.h
L  F = (x1+x2+f3(x3))^2;
Print;
.end
```

```
   F =
      x2^2 + 2*x1*x2 + x1^2 + 2*f3(x3)*x2 + 2*f3(x3)*x1 + f3(x3)^2;
```

It should also be noted that the convention for include files is to have the extension .h as it is in many other languages. If one is mixing various languages (like C, Fortran and FORM) and one would like to keep the naming of the FORM header files separately, one could use .hh as an emergency measure.

Of course one can put include instructions inside include files.

The #include instruction has some options which can be looked up in the reference manual. They concern including only parts of a file. As there is nowadays a #switch instruction (still to be shown) these options have become obsolete. At times they are however still encountered in existing code.

A more important feature concerns procedures. Procedures are technically macro's which reside in separate files. They act a bit like subroutines. Let us assume that we have the file dalemb.prc with the contents

```
#procedure dalemb(Q,t1,t2,m)
*
*   Procedure takes m powers of d'Alembertians in Q.
*   It uses t1 and t2 as temporary tensors. n and Dalemb should be symbols.
*
Multiply Dalemb^`m';
    Totensor `Q',`t1';
    id  Dalemb^n?*`t1'(?a) = distrib_(1,2*n,`t1',`t2',?a);
    ToVector,`Q',`t2';
    id  `t1'(?a) = dd_(?a);
#endprocedure
```

The file should begin with #procedure as its first characters. This should be followed by a name that corresponds exactly to the name of the file, except for that the file should have the extension .prc. After that there can be some parameters. These parameters will be preprocessor variables with string values. They will have to be referred to in the same way as regular preprocessor variables. As with the #define instruction, they are placed on a stack and will be popped again when the procedure has been completed.

Notice that commentary helps to explain what the procedure does and how it should

The way we use the procedure is as in:

```
Vectors P,p1,p2,p3;
Tensors f1,f2;
Symbol Dalemb,n;
Local F = P.p1^3*P.p2^5*P.p3^2;
#call dalemb(P,f1,f2,3)
.end
```

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

We see here the same result as we had previously with one of the earlier d'Alembertian programs. At the moment it is a bit friendlier though, because we don't have to think each time we need d'Alembertians. It is only necessary to prepare the procedure once and for all.

As the parameters are string variables, in special cases we might want to include comma's in them. This is done by placing the backslash character \ in front of the comma. The same holds for parentheses.

And of course procedures can be nested.

How does FORM locate procedures? There is a sequence of possible actions.

- First FORM looks whether the procedure is inside memory. This can be done for instance by having the text of the complete procedure inside the .frm file, before the place where it is used.
- If not inside the memory, FORM looks looks inside the current diretory for the file procname.prc (we assume that procname is the name of the procedure).
- Next FORM checks whether a directory was specified in the command that started FORM (see manual). If so, it will look in that directory.
- Finally it checks whether the environment variable FORMPATH was defined, using the same syntax as the other path variables under UNIX. If so it will look in the directories specified (in the order specified).
- If still not encountered, an errormessage will be printed and execution will be halted.

One can also put one or more procedures inside a header file and load them into the memory via the #include instruction.

Here we see an example of a procedure inside a header file. Assume the file vectors.h with the contents:

```
*
Tensors dalembf1,dalembf2;
Symbol Dalemb,dalembn;
*
#procedure dalemb(Q,m)
*
*   Procedure takes m powers of d'Alembertians in Q.
*   n and Dalemb should have been declared as symbols.
*
Multiply Dalemb^`m';
    Totensor `Q',dalembf1;
    id  Dalemb^dalembn?*dalembf1(?a) =
                     distrib_(1,2*dalembn,dalembf1,dalembf2,?a);
    ToVector,`Q',dalembf2;
    id  dalembf1(?a) = dd_(?a);
#endprocedure
```

This way we put both the necessary declarations and the procedure in one file. That can make life much easier.

The use of the above is even shorter now:

```
Vectors P,p1,p2,p3;
#include- vectors.h
*
Local F = P.p1^3*P.p2^5*P.p3^2;
*
#call dalemb(P,3)
.end
```

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

Notice that we don't need to know about the variables that are used locally. The files vectors.h has names for them that should be relatively safe against interfering with other names in the program.

The careful design of proceures and include files can make for very powerful and userfriendly libraries.

Now we can see easily that it is very efficient to take the three d'Alembertians all at the same time:

```
Vectors P,p1,p2,p3;
#include- vectors.h
*
Local F = P.p1^3*P.p2^5*P.p3^2;
*
#do i = 1,3
  #call dalemb(P,1)
#enddo
.end
```

```
Time =       0.00 sec    Generated terms =          115
                F        Terms in output =           27
                         Bytes used      =          902
```

Here we took the three d'Alembertians separately. As can be seen, there is now quite some duplicity, even though the answer is the same (one could inspect this better by listing the output).

Just for fun, the other example with d'Alembertians.

```
Vectors P,p1,p2,p3,p4;
#include- vectors.h
*
Local F = P.p1^12*P.p2^14*P.p3^16*P.p3^18;
*
#do i = 1,20
  #call dalemb(P,1)
  .sort
#enddo
.end
```

```
Time =       5.20 sec    Generated terms =      10599
                F        Terms in output =      10599
                         Bytes used      =     828824
```

When done in one pass, the execution time was 0.09 sec! And here we helped FORM actually by placing the .sort instruction inside the loop to remove duplicity after each d'Alembertian. With this .sort the situation is much worse:

With just 10 derivatives one gets

```
Vectors P,p1,p2,p3,p4;
#include- vectors.h
*
Local F = P.p1^12*P.p2^14*P.p3^16*P.p3^18;
*
#do i = 1,10
  #call dalemb(P,1)
#enddo
.end
```

```
Time =     2096.42 sec    Generated terms =   59891496
                 F        Terms in output =       2745
                          Bytes used      =     167262
```

With 20 derivatives the situation would be much worse!

Of course this doesn't mean that one should put .sort instructions after each operation. The proper placement of the .sort instructions is actually something for which one have to develop a feeling. The general rule is that if it resolves a sufficient amount of duplicity, it can be worth the cost of the sorting.

It also shows that a proper design of algorithms is very important.

Of course also the preprocessor needs some conditional flow control. We have the #if family and the #switch family.

```
#define MAX "5"
Symbols a1,...,a`MAX';
#if ( `MAX' < 5 )
  Local F = (a1+...+a`MAX')^2;
#else
  Local F = (a1+...+a`MAX')^1;
#endif
Print;
.end
```

```
F =
   a5 + a4 + a3 + a2 + a1;
```

We see here that if the string 'MAX' has a numerical interpretation we can use it as such in the #if instruction.

If the object to be compared with is not a number but a string one has to be careful that this string doesn't have an interpretation in the syntax. Hence we have to enclose the string in double quotation marks:

```
#define MAX "5ab"
Symbols a1,...,a5;
#if ( `MAX' != "3ba" )
  Local F = (a1+...+a5)^2;
#else
  Local F = (a1+...+a5)^1;
#endif
Print;
.end
```

```
F =
   a5^2 + 2*a4*a5 + a4^2 + 2*a3*a5 + 2*a3*a4 + a3^2 + 2*a2*a5 + 2*a2*a4 + 2
   *a2*a3 + a2^2 + 2*a1*a5 + 2*a1*a4 + 2*a1*a3 + 2*a1*a2 + a1^2;
```

This is particularly important when the string contains comma's or parentheses.

Of course there is also an #elseif instruction:

```
#define MAX "5ab"
Symbols a1,...,a5;
#if ( `MAX' == "3ab" )
  Local F = (a1+...+a5)^2;
#elseif ( `MAX' == "4ab" )
  Local F = (a1+...+a5)^1;
#elseif ( `MAX' == "5ab" )
  Local F = (a1+...+a5)^0;
#endif
Print;
.end
```

```
F =
   1;
```

Finally there is are also the #ifdef and the #ifndef instructions:

If the following program is called normally as in ”form prog” we obtain:

```
#ifndef `MAX'
  #define MAX "6"
#endif
Symbols a1,...,a`MAX';
Local  F = a1+...+a`MAX';
Print;
.end

 F =
    a6 + a5 + a4 + a3 + a2 + a1;
```

If on the other hand it is called as in "form -d MAX=4 prog" we obtain:

```
#ifndef `MAX'
  #define MAX "6"
#endif
Symbols a1,...,a`MAX';
Local  F = a1+...+a`MAX';
Print;
.end

 F =
    a4 + a3 + a2 + a1;
```

It should be obvious that one can make rather general programs this way.

The other control structure concerns the #switch family. Let us construct the file tryout.prc with the contents:

```
#procedure tryout(j)
#switch ‘j’
#case 0
 Multiply 10;
#break
#case 1
 Multiply 1/10;
#break
#case 2
 Multiply 105;
#break
#endswitch
#endprocedure
```

This can be used as in:

```
Symbol a;
Local F = 1+a+a^2;
if ( count(a,1) == 0 );
  #call tryout(2)
elseif ( count(a,1) == 1 );
  #call tryout(0)
else;
  #call tryout(1)
endif;
Print;
.end

 F =
    105 + 10*a + 1/10*a^2;
```

One can see that procedures can be called from inside if and other constructions, provided that the procedure doesn't contain .sort instructions. If it does, one is more limited in its use.

Finally the preprocessor has another important feature: the preprocessor calculator. If an expression is enclosed in curly brackets and it can be interpreted as a numerical expression, it will be evaluated over the 'short' integers. 'short' means that it should fit inside a 32 bits word on a 32 bits processor and inside a 64 bits word on a 64 bits processor. It works basically over the integers, hence $3/2 \longrightarrow 1$. If the expression cannot be evaluated as a numerical expression it will be left untouched and be interpreted as a set.

```
Symbols a1,...,a5;
L   F =
#do i = 1,3
    +a`i'*a{`i'+1}*a{`i'+2}
#enddo
    ;
Print;
.end
F =
    a3*a4*a5 + a2*a3*a4 + a1*a2*a3;
```

What are the possible operations that are allowed is explained in the manual. There exist some postfix operators for square roots and binary logarithms. Parentheses are allowed.

The object in an #if instruction can be a number, a string or a special function. These special functions are different from the ones encountered in the if statement, as at the level of the preprocessor we don't have individual terms. They are:

- termsin(nameofexpression) to give the number of terms in the indicated expression.
- maxpowerof(nameofsymbol) to give the maximum power of a symbol as potentially given in its declaration.
- minpowerof(nameofsymbol) to give the minimum power of a symbol as potentially given in its declaration.

The maximum and minimum powers need some extra explanation. One can declare a symbol in one of the following ways:

```
Symbol x;
Symbol a(:5);
Symbol b(-2:);
Symbol c(-2:5);
```

In this case terms that have a power of a or c that are is than 5 will be discarded. The same will happen to terms with a power of b or c that is lower than -2. The maxpowerof and minpowerof refer to these restrictions.

Considering that we have the symbolic part that operates on the level of terms and the preprocessor that manipulates the input, how do we get information about the terms and the contents of expressions to the preprocessor?

This is done with a special type of variables, named the dollar variables or in short the \$-variables. These are variables that contain symbolic objects that can also be interpreted as strings when they are used as preprocessor variables. They can contain a number, a symbol, an index, a whole term or even short expressions.

With short expression we mean expressions for which memory allocations are made. They don't reside on disk. Hence one should not make them too big or the system will eventually run out of memory and start swapping, seriously deteriorating performance.

One can set the value of a \$-variable in the preprocessor with

```
#$name = value;
```

while one sets the \$-variable in the symbolic part on a term by term basis with

```
$name = value;
```

There are more ways to set them at the symbolic level as we will see.

```
    Symbols a1,a2,a3;
    L   F = (a1+a2)^4;
    repeat id  a2^2 = a2*a3;
    #$maxi3 = 0;
    if ( count(a3,1) > $maxi3 ) $maxi3 = count_(a3,1);
    .sort

    #message The maximum power of a3 is `$maxi3'
~~~The maximum power of a3 is 3
    Print;
    .end

   F =
      a2*a3^3 + 4*a1*a2*a3^2 + 6*a1^2*a2*a3 + 4*a1^3*a2 + a1^4;
```

The count_ function returns the same value as the count object in the if statement.

The .sort is essential as we have only the maximum once all terms have been processed. If we omit it we would get 0 in the message (try to figure out why).

After the .sort we can use the $-variable either as a preprocessor variable or as a mini-expression in the RHS of a substitution. At times it can also be used as a parameter in a statement.

There is yet another mechanism to feed information back to the preprocessor. One can redefine preprocessor variables at the symbolic level. Note however that a new value will only be seen after the next .sort instruction. First we look at a simple program:

```
CF den;
S  x,n1,n2;
Local F = den(x+1)^1*den(x+2)^2*den(x+3)^3*den(x+4)^4*den(x+5)^5*den(x+6)^6;
SplitArg,((x)),den;
repeat;
  id den(n1?!{n2?},x)*den(n2?!{n1?},x) = (den(n1,x)-den(n2,x))*den(n2-n1);
  id den(x?number_) = 1/x;
endrepeat;
id den(n1?,x?) = den(n1+x);
.end
```

```
Time =       1.86 sec    Generated terms =      65973
                F        Terms in output =         21
                         Bytes used      =        920
```

We see an enormous duplicity. Of course we can make a #do loop instead of the repeat and include a .sort, but how many times should we go through the loop?

```
CF den;
S  x,n1,n2;
Local F = den(x+1)^1*den(x+2)^2*den(x+3)^3*den(x+4)^4*den(x+5)^5*den(x+6)^6;
SplitArg,((x)),den;
#do i = 1,1
  id,den(n1?!{n2?},x)*den(n2?!{n1?},x) = (den(n1,x)-den(n2,x))*den(n2-n1);
  id den(x?number_) = 1/x;
  if ( match(den(n1?!{n2?},x)*den(n2?!{n1?},x)) ) redefine i "0";
  .sort
#enddo
id den(n1?,x?) = den(n1+x);
.end

Time =       0.12 sec    Generated terms =         21
                F        Terms in output =         21
                         Bytes used      =        920
```

We have suppressed most of the statistics here, but one can see in them that there is far less duplicity. Also the running time indicates this.

We can improve this program:

```
CF den;
S  x,n1,n2;
Local F = den(x+1)^1*den(x+2)^2*den(x+3)^3*den(x+4)^4*den(x+5)^5*den(x+6)^6;
SplitArg,((x)),den;
#do i = 1,1
  id,once,den(n1?!{n2?},x)*den(n2?!{n1?},x) = (den(n1,x)-den(n2,x))*den(n2-n1)
  id den(x?number_) = 1/x;
  if ( match(den(n1?!{n2?},x)*den(n2?!{n1?},x)) ) redefine i "0";
  .sort
#enddo
id den(n1?,x?) = den(n1+x);
.end
```

```
Time =       0.01 sec    Generated terms =         21
                F        Terms in output =         21
                         Bytes used      =        920
```

Now it does only a single substitution and then immediately resolves the duplicity. It does however need far more sort operations, but they have only very few terms.

It is possible to improve it even further but that takes us beyond the level for beginners.