An Introduction to FORM

J.A.M. Vermaseren
Nikhef

1 Introduction

This course is intended for people who know already about computers and have experience working with a program editor. People can use their favourite editor like emacs, vim etc. There are example files which the students can run and modify, if they desire so.

Form is a program that is run in batch mode. This means that one prepares a program in the language of FORM and then lets FORM execute it. During the execution or after one can study the results. Assume for instance that we have a program in the file ex1.frm (the extension .frm is mandatory), we can run it with

```
form ex1
```

The system then locates the executable of FORM (prefarably /usr/local/bin/form) and starts it up. Next FORM looks for the file ex1.frm in the current directory and executes it. The output would be on the screen. Slightly more sophisticated is

```
form -l ex1
```

which would create the output file ex1.log. Output will be to the screen and to the file simultaneously. Certain output (designated as such in the program file) would be only to the log file. This has the advantage that one can monitor the progress of the program on the screen while at the same time catch (lengthy) outputs in a file.

2 The first examples

The first example is the file ex1.frm. It contains:

```
Symbols a,b,c;
Local F = (a+b+c)^10;
Print;
.end
```

Running this file with the command `form -l ex1` will give us its output on the screen and in the file ex1.log. The file should contain

```
FORM by J.Vermaseren,version 3.3(Mar 23 2009) Run at: Thu Mar 26 15:29:31 2009
    Symbols a,b,c;
    Local F = (a+b+c)^10;
    Print;
```

```
    .end

Time =       0.00 sec    Generated terms =         66
               F         Terms in output =         66
                         Bytes used      =       1160

   F =
      c^10 + 10*b*c^9 + 45*b^2*c^8 + 120*b^3*c^7 + 210*b^4*c^6 + 252*b^5*c^5
       + 210*b^6*c^4 + 120*b^7*c^3 + 45*b^8*c^2 + 10*b^9*c + b^10 + 10*a*c^9
       + 90*a*b*c^8 + 360*a*b^2*c^7 + 840*a*b^3*c^6 + 1260*a*b^4*c^5 + 1260*a*
      b^5*c^4 + 840*a*b^6*c^3 + 360*a*b^7*c^2 + 90*a*b^8*c + 10*a*b^9 + 45*a^2
      *c^8 + 360*a^2*b*c^7 + 1260*a^2*b^2*c^6 + 2520*a^2*b^3*c^5 + 3150*a^2*
      b^4*c^4 + 2520*a^2*b^5*c^3 + 1260*a^2*b^6*c^2 + 360*a^2*b^7*c + 45*a^2*
      b^8 + 120*a^3*c^7 + 840*a^3*b*c^6 + 2520*a^3*b^2*c^5 + 4200*a^3*b^3*c^4
       + 4200*a^3*b^4*c^3 + 2520*a^3*b^5*c^2 + 840*a^3*b^6*c + 120*a^3*b^7 +
      210*a^4*c^6 + 1260*a^4*b*c^5 + 3150*a^4*b^2*c^4 + 4200*a^4*b^3*c^3 +
      3150*a^4*b^4*c^2 + 1260*a^4*b^5*c + 210*a^4*b^6 + 252*a^5*c^5 + 1260*a^5
      *b*c^4 + 2520*a^5*b^2*c^3 + 2520*a^5*b^3*c^2 + 1260*a^5*b^4*c + 252*a^5*
      b^5 + 210*a^6*c^4 + 840*a^6*b*c^3 + 1260*a^6*b^2*c^2 + 840*a^6*b^3*c +
      210*a^6*b^4 + 120*a^7*c^3 + 360*a^7*b*c^2 + 360*a^7*b^2*c + 120*a^7*b^3
       + 45*a^8*c^2 + 90*a^8*b*c + 45*a^8*b^2 + 10*a^9*c + 10*a^9*b + a^10;

  0.00 sec out of 0.00 sec
```

We see first a line that tells when the program ran and what version of FORM was used. Then the program is listed and after that the reaction of FORM to this program. It gives first some running statistics, indicating amoung others the CPU time since the startup of FORM, the number of terms in the output and the number of bytes that the output occupies. It should be noted that both the running time and the number of bytes used are very low in FORM as compared to other programs. It can be more than an order of magnitude better. Finally FORM shows the total amount of CPU time used and the real time elapsed since the startup of the program.

Let us now have a look at how FORM obtains its results. For this we run the file ex1a.frm.

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

and the output is

```
FORM by J.Vermaseren,version 3.3(Mar 23 2009) Run at: Thu Mar 26 15:29:26 2009
    Symbols a,b;
    Local F = (2*a+b-a)^2;
    Print "=== %t";
    Print;
    .sort
===  + 4*a^2
===  + 4*a*b
===  - 4*a^2
```

```
=== + b^2
=== - 2*a*b
=== + a^2

Time =       0.00 sec    Generated terms =          6
               F         Terms in output =          3
                         Bytes used      =         54

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

    Print "--- %t";
    .end
---  + b^2
---  + 2*a*b
---  + a^2

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

We see that in the beginning only the input till the .sort instruction is being listed. This is because FORM programs consist of modules that are separated by instructions that start with a period. Each module is translated and immediately after it is executed. After the execution of a module is completed, FORM will forget it (and its input) and use its output as input for the next module.

The print statement with the double quotation marks is a statement that is executed each time FORM passes that statement during execution. What happens is that FORM encounters the definition of F and then starts working out the rhs. For each term that it generates it applies the remaining statements of the module and then it stores the term away. Hence we see that it generates first `4*a^2`. After writing this away, it generates `4*a*b` etc. In the end 6 terms are written away and then FORM sorts them bringing the expression to 'normal form'. This may involve adding coefficients and if necessary cancelling terms. The final expression is then written away and printed (if asked for) and the module is finished. FORM will clear the module from its buffers and then read the next module. The output of the first module will become the input for the second. We can see this in the printing as now the order of the terms is quite different from the order in the first module.

What happens with the terms that are written away one by one and how they are sorted can be seen better in the third example ex1b.frm

```
   #:SmallSize 1000
   #:LargePatches 4
   Symbols a,b,c,d;
   Local F1 = (a+b+c)^10;
   Local F2 = (a+b+c+d)^10;
   .end
```

This program gives the output

```
FORM by J.Vermaseren,version 3.3(Mar 23 2009) Run at: Thu Mar 26 15:29:26 2009
```

```
    #:SmallSize 1000
    #:LargePatches 4
    Symbols a,b,c,d;
    Local F1 = (a+b+c)^10;
    Local F2 = (a+b+c+d)^10;
    .end

Time =       0.00 sec    Generated terms =         44
              F1       1 Terms left      =         44
                         Bytes used      =        788

Time =       0.00 sec    Generated terms =         66
              F1       1 Terms left      =         66
                         Bytes used      =       1162

Time =       0.00 sec    Generated terms =         66
              F1         Terms in output =         66
                         Bytes used      =       1160

Time =       0.00 sec    Generated terms =         43
              F2       1 Terms left      =         43
                         Bytes used      =        854

Time =       0.00 sec    Generated terms =         82
              F2       1 Terms left      =         82
                         Bytes used      =       1634

Time =       0.00 sec    Generated terms =        121
              F2       1 Terms left      =        121
                         Bytes used      =       2394

Time =       0.00 sec    Generated terms =        159
              F2       1 Terms left      =        159
                         Bytes used      =       3142

Time =       0.00 sec    Generated terms =        197
              F2       1 Terms left      =        197
                         Bytes used      =       3882

Time =       0.00 sec
              F2         Terms active    =        197
                         Bytes used      =       3856

Time =       0.00 sec    Generated terms =        239
              F2       1 Terms left      =        239
                         Bytes used      =       4632

Time =       0.00 sec    Generated terms =        283
              F2       1 Terms left      =        283
                         Bytes used      =       5402

Time =       0.00 sec    Generated terms =        286
              F2       1 Terms left      =        286
                         Bytes used      =       5456
```

```
Time =       0.00 sec
              F2        Terms active    =        286
                        Bytes used      =       5480

Time =       0.00 sec   Generated terms =        286
              F2        Terms in output =        286
                        Bytes used      =       5416
  0.00 sec out of 0.00 sec
```

In this program we encounter first some global settings that hold for the whole FORM run. They start with `#:` and must be at the beginning of the program. They control how much space FORM reserves for some buffers. In this case we make these buffers artificially small so that we see what happens when such a buffer becomes full.

First we look at the first three statistic blocks. They all contain the name F1 in the second line, indicating that they refer to the treatment of the first expression. When terms are written away they are written to a buffer which is called the small buffer. With the `#: SmallSize 1000` instruction we forced this buffer to be 1000 bytes. Hence after 43 terms have been written in there it is full. The 44-th term doesn't fit. So when the 44-th term arrives the contents of the small buffer are sorted, compressed and written to another buffer which is called the large buffer. Now the 44-th term can be written into the small buffer. Each time the small buffer has been sorted statistics are printed. After 66 terms there are no more terms to be generated. Then the final sorts take place. First the contents of the small buffer are sorted and placed in the large buffer and then the two sorted 'patches' in the large buffer are merged and written to output. This last merge produces the final statistics for this expression.

With expression F2 we run into a limit for the large buffer. Of this the size can be controled, but also the maximum number of patches in it. We have set this to 4. Hence after the small buffer has been sorted for the fifth time we cannot write its sorted results to the large buffer. Therefore the large buffer is now sorted and the result is written to file. This file is called the sort file. This gives the special statistics with 'Terms active'. Now there is room again in the large buffer. In the end the remains in the small buffer are sorted. Then the remains in the large buffer are sorted and written to file and finally the patches in the file are merged, using the combined small and large buffers as a cache system, and the results are written to the output.

The above example was rather artificial as we set some buffer values to unrealistically small amounts. We could have used some brute force and test at the same moment the speed of FORM:

```
Symbols a,b,c,d,e,f,g;
Local F = (a+b+c+d+e+f+g)^32;
.end
```

This file results in ex1c.log

```
FORM by J.Vermaseren,version 3.3(Mar 23 2009) Run at: Sat Mar 28 20:11:15 2009
    Symbols a,b,c,d,e,f,g;
    L F = (a+b+c+d+e+f+g)^32;
    .end

Time =       0.71 sec    Generated terms =     100000
               F       1 Terms left      =     100000
                         Bytes used      =    3328872
```

```
Time =        1.57 sec    Generated terms =     200000
                 F       1 Terms left      =     200000
                           Bytes used      =    6783982

Time =        2.49 sec    Generated terms =     300000
                 F       1 Terms left      =     300000
                           Bytes used      =   10236598

Time =        3.44 sec    Generated terms =     400000
                 F       1 Terms left      =     400000
                           Bytes used      =   13766904

Time =        4.43 sec    Generated terms =     500000
                 F       1 Terms left      =     500000
                           Bytes used      =   17281720

Time =        5.49 sec    Generated terms =     600000
                 F       1 Terms left      =     600000
                           Bytes used      =   20768048

Time =        6.57 sec    Generated terms =     700000
                 F       1 Terms left      =     700000
                           Bytes used      =   24366208

Time =        7.60 sec    Generated terms =     800000
                 F       1 Terms left      =     800000
                           Bytes used      =   27865620

Time =        8.70 sec    Generated terms =     900000
                 F       1 Terms left      =     900000
                           Bytes used      =   31320952

Time =        9.78 sec    Generated terms =    1000000
                 F       1 Terms left      =    1000000
                           Bytes used      =   34885940

Time =       10.92 sec    Generated terms =    1100000
                 F       1 Terms left      =    1100000
                           Bytes used      =   38297250

Time =       11.96 sec    Generated terms =    1200000
                 F       1 Terms left      =    1200000
                           Bytes used      =   41837466

Time =       13.12 sec    Generated terms =    1300000
                 F       1 Terms left      =    1300000
                           Bytes used      =   45309350
```

```
Time =       14.14 sec    Generated terms =    1400000
                F       1 Terms left      =    1400000
                          Bytes used      =   48679204

Time =       15.30 sec    Generated terms =    1500000
                F       1 Terms left      =    1500000
                          Bytes used      =   52213990

Time =       15.85 sec
                F         Terms active    =    1500000
                          Bytes used      =   52213714

Time =       16.97 sec    Generated terms =    1600000
                F       1 Terms left      =    1600000
                          Bytes used      =   55556044

Time =       18.03 sec    Generated terms =    1700000
                F       1 Terms left      =    1700000
                          Bytes used      =   58949128

Time =       19.17 sec    Generated terms =    1800000
                F       1 Terms left      =    1800000
                          Bytes used      =   62423186

Time =       20.32 sec    Generated terms =    1900000
                F       1 Terms left      =    1900000
                          Bytes used      =   65755070

Time =       21.32 sec    Generated terms =    2000000
                F       1 Terms left      =    2000000
                          Bytes used      =   69002468

Time =       22.42 sec    Generated terms =    2100000
                F       1 Terms left      =    2100000
                          Bytes used      =   72453706

Time =       23.56 sec    Generated terms =    2200000
                F       1 Terms left      =    2200000
                          Bytes used      =   75808688

Time =       24.69 sec    Generated terms =    2300000
                F       1 Terms left      =    2300000
                          Bytes used      =   78996440

Time =       25.63 sec    Generated terms =    2400000
                F       1 Terms left      =    2400000
                          Bytes used      =   82128736
```

```
Time =       26.66 sec    Generated terms =    2500000
                F       1 Terms left      =    2500000
                          Bytes used      =   85365974

Time =       27.76 sec    Generated terms =    2600000
                F       1 Terms left      =    2600000
                          Bytes used      =   88549490

Time =       28.84 sec    Generated terms =    2700000
                F       1 Terms left      =    2700000
                          Bytes used      =   91616378

Time =       29.44 sec    Generated terms =    2760681
                F       1 Terms left      =    2760681
                          Bytes used      =   93353936

Time =       29.96 sec
                F         Terms active    =    2760681
                          Bytes used      =   95091252

Time =       31.16 sec    Generated terms =    2760681
                F         Terms in output =    2760681
                          Bytes used      =   93353668
  31.27 sec out of 32.80 sec
```

This example was run on a PentiumIV-2800 notebook computer.

3 Variables

FORM has a number of types of variables. Each has to be declared as such. There are symbols, vectors, indices, commuting functions, non-commuting functions,tensors and sets. These are called algebraic variables. In addition there are preprocessor variables and $-variables. Finally there are the expressions, which are collections of terms. It are the terms that we are manipulating.

3.1 Functions, commuting and non-commuting. Drop statement.

Example ex2a.frm:

```
    CFunction f;
    Symbol x;
    Local F = f(x)+f(x^2)+f(x,x+1)+f;
    Print;
    .sort

Time =       0.00 sec    Generated terms =          4
               F         Terms in output =          4
                         Bytes used      =        114

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

```
Functions A,B;
Drop F;
Local G = (A+B)^3;
Print;
.end
```

```
Time =       0.00 sec    Generated terms =          8
                G        Terms in output =          8
                         Bytes used      =        162

   G =
      A*A*A + A*A*B + A*B*A + A*B*B + B*A*A + B*A*B + B*B*A + B*B*B;
```

We see here that functions can have arbitrary numbers of arguments. Zero arguments is also allowed. For noncommuting variables one uses noncommuting functions without arguments.

The drop statement erases existing expressions. It comes in two varieties: Without arguments it drops all existing expressions and with arguments it drops only the expressions mentioned in the arguments.

3.2 Vectors have indices. Contractions give dotproducts

Example ex2b.frm:

```
Vector p,q;
Indices mu,nu,rho;
Local F = p(mu)*q(nu)+p(mu)*q(mu)*p(nu)*q(nu)*q(rho);
Print;
.end
```

```
Time =       0.00 sec    Generated terms =          2
                F        Terms in output =          2
                         Bytes used      =         48

   F =
      p(mu)*q(nu) + q(rho)*p.q^2;
```

Here we see that vectors have indices. They have to be declared as such. Contracted indices are automatically summed over. In the next example we will see how this can be prevented.

3.3 Indices and their dimensions

Example ex2c.frm:

```
Symbol n,x;
Indices mu=4,nu,i=3,rho=n,a=0;
Local F = x*d_(mu,mu)+x^2*d_(nu,nu)+x^3*d_(i,i)+x^4*d_(rho,rho)
 +x^5*d_(a,a);
print +s;
.end
```

```
Time =       0.00 sec    Generated terms =          5
                F        Terms in output =          5
                         Bytes used      =         86
```

```
  F =
     + 4*x
     + 4*x^2
     + 3*x^3
     + n*x^4
     + d_(a,a)*x^5
    ;
```

The object `d_` is the kronecker delta. We declare the dimension of an index together with the index. If no dimension is specified the default dimension is taken which is 4. This can be changed with the dimension statement. The dimension can be a nonnegative integer or a symbol. If the dimension is zero the index is not summed over. Actually the dimension comes only into play when there is a Kronecker delta with two identical indices. The print statement here has the option +s which indicates that the output is printed in a mode in which each term starts on a new line.

3.4 Schoonschip notation for contracted indices

Example ex2d.frm:

```
    CFunction f,g;
    Indices mu,nu,ro,si,a=0;
    Vectors p,q,r,s;
    L F = f(mu)*p(mu)+f(a)*p(a)+f(mu)*g(mu)
     +e_(mu,nu,ro,si)*p(mu)*q(nu)*r(ro)*s(si);
    Print +s;
    .end
```

```
Time =       0.00 sec    Generated terms =          4
                F        Terms in output =          4
                         Bytes used      =         86

  F =
     + e_(p,q,r,s)
     + f(p)
     + f(mu)*g(mu)
     + f(a)*p(a)
    ;
```

The object `e_` is the Levi-Civita tensor. Schoonschip notation is the notation in which we write a vector in the place of an index if the index of that vector is contracted with the index that was originally in that position. Note that the index a is zero dimensional and hence does not get contracted. The Levi-Civita tensor does not have to have the same number of indices as the dimension of the indices. How this is to be interpreted is up to the user.

3.5 $-variables

Example ex2e.frm:

```
    Symbols a,b,c;
    Local F = (a+b+c)^3;
    #$c = 0;
    $c = $c + 1;
    print +f "<%$> %t",$c;
```

```
    Print +f;
    .end
<1>  + a^3
<2>  + 3*a^2*b
<3>  + 3*a^2*c
<4>  + 3*a*b^2
<5>  + 6*a*b*c
<6>  + 3*a*c^2
<7>  + b^3
<8>  + 3*b^2*c
<9>  + 3*b*c^2
<10>  + c^3

Time =       0.00 sec    Generated terms =         10
                F        Terms in output =         10
                         Bytes used      =        180

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

The $-variables are special systems variables that can be applied on a term by term basis. They don't really belong to the algebraic expression. Rather they can contain information about the expression or the terms. There are two basic ways to give them a value. The first is during compilation. In that case they should be preceeded by the character #. We see that in the above example we use this to initialize the variable $c. The second way to give them a value is during execution. Here for each term the value of $c is raised by one. We can use the value of the $-variable in a print statement by the control sequence %$ and mentioning which variable we want to use after the control string. $-variables can contain numbers, arguments, groups of arguments, single variables, complete terms or even complete expressions. One should be careful assigning expressions to them as they are kept in memory. Hence very big expressions might slow down execution considerably.

3.6 Preprocessor variables

Example ex2f.frm:

```
    #define MAX "5"
    Symbols x,y;
    #do i = 1,‘MAX’
    Local F‘i’ = (x+y)^‘i’;
    #enddo
    Print;
    .end

Time =       0.00 sec    Generated terms =          2
               F1        Terms in output =          2
                         Bytes used      =         32

Time =       0.00 sec    Generated terms =          3
               F2        Terms in output =          3
                         Bytes used      =         54

Time =       0.00 sec    Generated terms =          4
```

```
             F3       Terms in output =         4
                      Bytes used      =        70

Time =       0.00 sec    Generated terms =         5
             F4       Terms in output =         5
                      Bytes used      =        86

Time =       0.00 sec    Generated terms =         6
             F5       Terms in output =         6
                      Bytes used      =       102

   F1 =
      y + x;

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

   F3 =
      y^3 + 3*x*y^2 + 3*x^2*y + x^3;

   F4 =
      y^4 + 4*x*y^3 + 6*x^2*y^2 + 4*x^3*y + x^4;

   F5 =
      y^5 + 5*x*y^4 + 10*x^2*y^3 + 10*x^3*y^2 + 5*x^4*y + x^5;
```

Preprocessor variables are aids during compilation. They contain string values that sometimes are interpreted as numbers as is the case here in the preprocessor do-loop. We give preprocessor variables a value with the (re)define instruction. The value is given between double quotes. When we use a preprocessor variable its name is placed between a back-quote quote combination. This contruction can be nested. The loop variable in a do-loop construction is automatically also a preprocessor variable. Note that the do-loop generates the contents 'MAX' times, each time with the appropriate value for 'i'. This is different from loops in the languages C and Fortran where the code exists once and the loop passes through it several times during execution. Here the code is generated several times and then compiled. Note also that because of this the loop may contain .sort instructions.

The $-variables can also be used as preprocessor variables. For this one has to enclose them with a back-quote quote pair as in `` `$c' ``. They have then the 'value' that exists during compilation time and this 'value' is converted into a string. This allows the contents of the terms to influence the structure of the program.

3.7 Example of that the terms get treated by statements one by one

Example ex2g.frm:

```
    Symbols x,y;
    Local F = (x+y)^2;
    Print +f "<1> %t";
    Multiply 2;
    Print +f "<2> %t";
    Print;
    .end
<1>  + x^2
```

```
<2>  + 2*x^2
<1>  + 2*x*y
<2>  + 4*x*y
<1>  + y^2
<2>  + 2*y^2

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

   F =
      2*y^2 + 4*x*y + 2*x^2;
```

What we see here is that each term gets generated and then treated by the statement(s) until the end of the module is reached, and the term is stored away. Then the next term in the expansion of `(x+y)^2` is generated. Hence the treatement of terms in a module follows a giant tree structure in which the statements are at the potential splittings of the branches.

4 Substitutions

Of course the essence of symbolic manipulation is that we can modify the terms. This can be done in many different ways. The most powerful one is the substitution. The general form of a substitution is

```
id,options,lhs = rhs;
```

For the moment we will forget about the options. In that case we have

```
id lhs = rhs;
```

The id stands for identify and the action is that when the lhs occurs in a term it will be replaced by the rhs.

Example ex3a.frm:

```
    Symbols x,y;
    Local F = (x+1)^3;
    Print;
    .sort

Time =       0.00 sec    Generated terms =          4
                F        Terms in output =          4
                         Bytes used      =         50

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

    id x = y;
    Print;
    .sort

Time =       0.00 sec    Generated terms =          4
                F        Terms in output =          4
                         Bytes used      =         50
```

```
   F =
      1 + 3*y + 3*y^2 + y^3;

    id y = x-1;
    Print;
    .end

Time =       0.00 sec    Generated terms =         10
                F        Terms in output =          1
                         Bytes used      =         18

   F =
      x^3;
```

There is a problem if one needs to do two replacements simultaneously as shown here Example ex3b.frm:

```
    S x,y,sinphi,cosphi;
    Local F = x^2+y^2;
    id x = x*cosphi-y*sinphi;
    id y = x*sinphi+y*cosphi;
    id sinphi^2 = 1-cosphi^2;
    Print;
    .sort

Time =       0.00 sec    Generated terms =         15
                F        Terms in output =         10
                         Bytes used      =        158

   F =
      2*y^2*cosphi^2 - y^2*cosphi^4 + 4*x*y*sinphi*cosphi - 2*x*y*sinphi*
      cosphi^2 - 2*x*y*sinphi*cosphi^3 + 2*x^2 - 2*x^2*cosphi - 2*x^2*cosphi^2
       + 2*x^2*cosphi^3 + x^2*cosphi^4;

    Drop;
    Local F = x^2+y^2;
    id x = x*cosphi-y*sinphi;
    al y = x*sinphi+y*cosphi;
    id sinphi^2 = 1-cosphi^2;
    Print;
    .end

Time =       0.00 sec    Generated terms =          8
                F        Terms in output =          2
                         Bytes used      =         32

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

The al stands for also and means that the lhs of this statement is taken out together with the lhs of the previous id statement and before the rhs of the previous id statement is inserted. Note also that id $\text{sinphi}^2 = \cdots$ takes out all integer powers of sinphi^2. sinphi itself is left untouched.

4.1 Patterns

Generally the lhs of a substitution is called a pattern. It describes what we have to substitute. This may involve generic variables, called wildcards. Example ex4a.frm:

```
    Symbols x,n;
    Local F = (x+2)^3;
    id x^n? = x^(n+1)/(n+1);
    Print;
    .end

Time =       0.00 sec    Generated terms =          4
                F        Terms in output =          4
                         Bytes used      =         54

   F =
      8*x + 6*x^2 + 2*x^3 + 1/4*x^4;
```

n is called a wildcard and is indicated in the pattern with a questionmark. `x^n?` will match any power of x, also `x^0`. Because we have no negative powers of x there is no problem wrt dividing by zero. What would happen in that case? Example ex4b.frm:

```
    Symbols x,n;
    Local F = (x+2)^3/x^2;
    id x^n? = x^(n+1)/(n+1);
    Print;
    .end
Division by zero during normalization
```

We see that FORM terminates with an error. We should have forseen this case and intercepted it: Example ex4c.frm:

```
    Symbols x,n,lnx;
    Local F = (x+2)^3/x^2;
    id x^n?!{,-1} = x^(n+1)/(n+1);
    al 1/x = lnx;
    Print;
    .end

Time =       0.00 sec    Generated terms =          4
                F        Terms in output =          4
                         Bytes used      =         58

   F =
       - 8*x^-1 + 12*lnx + 6*x + 1/2*x^2;
```

Note that we have to use the al statement because the other statement will generate a new term with 1/x. The construction n?!,-1 means anything except for the set that consists of -1. The reason of the comma will become clear at a later stage. For now the rule is that if a set consists only of a single number, we need an extra comma between the to indicate that it is really a set. The empty set is not allowed.

Wildcards can be restricted to sets or anything but a set. Sets can either be declared or be given dynamically. Declared set: Example ex4d.frm:

```
    Symbols x,a,b,c,d,e;
    Set abc:a,b,c;
```

```
CFunction f;
Local F = f(a)+f(b)+f(c)+f(d)+f(e);
id f(x?abc) = f(x,x);
id f(x?!abc) = f(x+1);
Print;
.end

Time =       0.00 sec    Generated terms =          5
                F        Terms in output =          5
                         Bytes used      =        118

   F =
      f(1 + e) + f(1 + d) + f(a,a) + f(b,b) + f(c,c);
```

This could also have been done with Example ex4e.frm:

```
Symbols x,a,b,c,d,e;
CFunction f;
Local F = f(a)+f(b)+f(c)+f(d)+f(e);
id f(x?{a,b,c}) = f(x,x);
id f(x?!{a,b,c}) = f(x+1);
Print;
.end

Time =       0.00 sec    Generated terms =          5
                F        Terms in output =          5
                         Bytes used      =        118

   F =
      f(1 + e) + f(1 + d) + f(a,a) + f(b,b) + f(c,c);
```

If a set has to be used many times it is better to declare it. If it has to be used only once one may as well use the dynamical definition. One can also refer to the elements of a set. Example ex4f.frm:

```
Symbols x,n,a,b,c,d,e;
Set abc:a,b,c;
CFunction f;
Local F = f(a)+f(b)+f(c)+f(d)+f(e);
id f(x?abc[n]) = f(x,n);
Print;
.end

Time =       0.00 sec    Generated terms =          5
                F        Terms in output =          5
                         Bytes used      =         86

   F =
      f(a,1) + f(b,2) + f(c,3) + f(d) + f(e);
```

This way sets can also be used in the rhs. Note however that the index can only be a single symbol or a single positive number, no larger than the number of elements in the set. Example ex4g.frm:

```
Symbols x,n,a,b,c,d,e;
Vector p,q,r;
```

```
    Set abc:a,b,c;
    Set pqr:p,q,r;
    CFunction f;
    Local F = f(a)+f(b)+f(c)+f(d)+f(e);
    id f(x?abc[n]) = f(x,pqr[n]);
    Print;
    .end

Time =       0.00 sec    Generated terms =          5
                F        Terms in output =          5
                         Bytes used      =         86

   F =
      f(a,p) + f(b,q) + f(c,r) + f(d) + f(e);
```

For more options with the sets one should consult the manual.

4.2 Functions

Let I be a matrix that is a function of a variable x. One matrix element is then for instance I(i1,i2,x). Example ex5a.frm:

```
    Symbols x,y,z,x1,...,x6;
    Indices i1,...,i6;
    CFunction I;
    Local F = I(i1,i2,x)*I(i2,i3,y)*I(i3,i4,z)
             *I(i4,i5,x)*I(i5,i6,z)*I(i6,i1,y);
    id  I(i1?,i2?,x?)*I(i2?,i3?,y?) = I(i1,i3,x,y);
    Print;
    .sort

Time =       0.00 sec    Generated terms =          1
                F        Terms in output =          1
                         Bytes used      =         76

   F =
      I(i1,i3,x,y)*I(i3,i5,z,x)*I(i5,i1,z,y);

    id  I(i1?,i2?,x1?,x2?)*I(i2?,i3?,x3?,x4?) = I(i1,i3,x1,x2,x3,x4);
    id  I(i1?,i2?,x1?,x2?,x3?,x4?)*I(i2?,i3?,x5?,x6?) = I(i1,i3,x1,...,x6);
    Print;
    .sort

Time =       0.00 sec    Generated terms =          1
                F        Terms in output =          1
                         Bytes used      =         48

   F =
      I(i1,i1,x,y,z,x,z,y);

    Drop;
    Local G = I(i1,i2,x)*I(i2,i3,y)*I(i3,i4,z)
             *I(i4,i5,x)*I(i5,i6,z)*I(i6,i1,y);
    repeat;
      id  I(i1?,i2?,?a)*I(i2?,i3?,?b) = I(i1,i3,?a,?b);
```

```
endrepeat;
Print;
.end

Time =       0.00 sec    Generated terms =          1
                G       Terms in output =          1
                         Bytes used      =         48

   G =
      I(i1,i1,x,y,z,x,z,y);
```

We can string the matrices together and we see that the result is the trace over the product of six matrices. The first method however is rather laborious when things become general. In the second method we use:

- a new type of wildcard: ?a and ?b
- the repeat loop.

The repeat loop is very much related to a while statement in other languages because

```
repeat;
  if ( condition );
  endif;
endrepeat;
```

is equivalent to

```
while ( condition);
endwhile;
```

In our case the fact that the id statement catches something means that the condition is fulfilled. ?a means any sequence of arguments, including no argument. These variables don't have to be delared as they have only one interpretation.

Imagine we have a function den(x) which stands for 1/x. We can split fractions with: Example ex5b.frm:

```
CF den;
S  x,x1,x2;
L  F = den(x+1)*den(x+2)^2*den(x+3)^3*den(x+4)^4;
SplitArg,den;
Print;
.sort

Time =       0.00 sec    Generated terms =          1
                F       Terms in output =          1
                         Bytes used      =        150

   F =
      den(1,x)*den(2,x)^2*den(3,x)^3*den(4,x)^4;

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;
    Print +s;
    .end

Time =       0.00 sec    Generated terms =        181
                F        Terms in output =         10
                         Bytes used      =        216

   F =
       + 1/648*den(1,x)
       + 1/4*den(2,x)
       - 1/16*den(2,x)^2
       - 17/8*den(3,x)
       + 3/4*den(3,x)^2
       - 1/2*den(3,x)^3
       + 607/324*den(4,x)
       + 403/432*den(4,x)^2
       + 13/36*den(4,x)^3
       + 1/12*den(4,x)^4
      ;
```

First we see the SplitArg command that takes a multiterm argument and assigns one argument per term. There are variations in which it takes only a single specified term to make a new argument. In this program we like to take x out and hence then the statement would have been SplitArg,((x)),den; The ((x)) means only terms that are a numeric multiple of x. The option (x) means all terms that contain x become a separate argument. One can also specify in which functions (like den) or which arguments this should happen. The new wildcarding here is that we define a set x2? and a set x1?. If we would just say x2 it would look for exactly the object x2. With the questionmark it knows that this is the value that the wildcard x2 gets. Hence this construction means that x1 and x2 should not get the same value.

When we extend this example to 6 we start seeing that the amount of CPU time becomes nonnegligible.

Example ex5c.frm:

```
    L  F = den(x+1)*den(x+2)^2*den(x+3)^3*den(x+4)^4
        *den(x+5)^5*den(x+6)^6;
                      .
                      .
Time =       4.82 sec    Generated terms =      65973
                F        Terms in output =         21
                         Bytes used      =        520
```

Many terms are generated while there are actually few different terms. We can speed the process up by doing several steps and then sorting, after which we do the rest:

Example ex5d.frm:

```
    CF den;
    S  x,x1,x2;
    L  F = den(x+1)*den(x+2)^2*den(x+3)^3*den(x+4)^4
     *den(x+5)^5*den(x+6)^6;
    SplitArg,den;
    Print;
    .sort
```

```
Time =       0.00 sec    Generated terms =          1
                F        Terms in output =          1
                         Bytes used      =        304

   F =
      den(1,x)*den(2,x)^2*den(3,x)^3*den(4,x)^4*den(5,x)^5*den(6,x)^6;

      id  den(x1?!{x2?},x)*den(x2?!{x1?},x) =
     (den(x1,x)-den(x2,x))*den(x2-x1);
      id  den(x1?!{x2?},x)*den(x2?!{x1?},x) =
     (den(x1,x)-den(x2,x))*den(x2-x1);
      id  den(x1?!{x2?},x)*den(x2?!{x1?},x) =
     (den(x1,x)-den(x2,x))*den(x2-x1);
    .sort

Time =       0.25 sec    Generated terms =      10513
                F        Terms in output =       1612
                         Bytes used      =      87112
    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;
    .end

Time =       0.93 sec    Generated terms =       9745
                F        Terms in output =         21
                         Bytes used      =        520
```

Basically what we need is a repeat with a .sort but that is not possible

```
repeat;
   id .....
   .sort
endrepeat;
```

This would cause a syntax error because when the module is executed there is no endrepeat. For this we need the preprocessor. And some communication with the preprocessor. We want to make modules with one id statement and keep executing these modules as long as something can still be done. This gives the program Example ex5e.frm:

```
    CF den;
    S  x,x1,x2;
    L  F = den(x+1)*den(x+2)^2*den(x+3)^3*den(x+4)^4
     *den(x+5)^5*den(x+6)^6;
    SplitArg,den;
    Print;
    .sort

Time =       0.00 sec    Generated terms =          1
                F        Terms in output =          1
                         Bytes used      =        304

   F =
      den(1,x)*den(2,x)^2*den(3,x)^3*den(4,x)^4*den(5,x)^5*den(6,x)^6;
```

```
    #do i = 1,1
      id  den(x1?!{x2?},x)*den(x2?!{x1?},x) =
     (den(x1,x)-den(x2,x))*den(x2-x1);
      id  den(x?number_) = 1/x;
      if ( match(den(x1?!{x2?},x)*den(x2?!{x1?},x)) );
     redefine i "0";
      endif;
      .sort

Time =       0.01 sec    Generated terms =        144
                F        Terms in output =        144
                         Bytes used      =      12084
    #enddo

Time =       0.09 sec    Generated terms =       2248
                F        Terms in output =        259
                         Bytes used      =       8302

Time =       0.11 sec    Generated terms =       1155
                F        Terms in output =        162
                         Bytes used      =       3870

Time =       0.12 sec    Generated terms =        396
                F        Terms in output =         82
                         Bytes used      =       1860

Time =       0.12 sec    Generated terms =        157
                F        Terms in output =         59
                         Bytes used      =       1324

Time =       0.12 sec    Generated terms =         97
                F        Terms in output =         44
                         Bytes used      =       1020

Time =       0.13 sec    Generated terms =         67
                F        Terms in output =         35
                         Bytes used      =        796

Time =       0.13 sec    Generated terms =         49
                F        Terms in output =         26
                         Bytes used      =        616

Time =       0.13 sec    Generated terms =         31
                F        Terms in output =         21
                         Bytes used      =        520
    Print +s;
    .end

Time =       0.13 sec    Generated terms =         21
                F        Terms in output =         21
                         Bytes used      =        520

   F =
```

```
     + 1/10368000000*den(1,x)
     + 43/95551488*den(2,x)
     - 1/15925248*den(2,x)^2
     - 265/559872*den(3,x)
     + 1/7776*den(3,x)^2
     - 1/46656*den(3,x)^3
     - 1489/41472*den(4,x)
     + 53/3456*den(4,x)^2
     - 11/2304*den(4,x)^3
     + 1/768*den(4,x)^4
     + 374111/5971968*den(5,x)
     + 3109/497664*den(5,x)^2
     + 269/13824*den(5,x)^3
     + 5/3456*den(5,x)^4
     + 1/288*den(5,x)^5
     - 117653266057/4478976000000*den(6,x)
     - 1661734447/149299200000*den(6,x)^2
     - 1888673/466560000*den(6,x)^3
     - 49313/41472000*den(6,x)^4
     - 29/115200*den(6,x)^5
     - 1/34560*den(6,x)^6
    ;
```

We reset the do loop parameter as long as there is still work to do. Notice also the if statement. The condition here is that if there is a match, the answer is true. Actually the answer is a number that indicates the number of matches there are. The redefine statement is a way to redefine a preprocessor variable. In this case the do loop parameter.

Actually we can make this program even faster. We notice that the second time in the loop there are very many terms generated and only 10surviving:

```
Time =       0.09 sec    Generated terms =       2248
                F        Terms in output =        259
                         Bytes used      =       8302
```

This is because the id statement catches more than one combination at a time. If we force the id statement to make only a single substitution things go even faster: Example ex5e.frm:

```
                          .
  id,once,den(x1?!{x2?},x)*den(x2?!{x1?},x) =
(den(x1,x)-den(x2,x))*den(x2-x1);
                          .
```

This is done with the option once.

```
Time =       0.01 sec    Generated terms =         21
                F        Terms in output =         21
                         Bytes used      =        520
```

As you see there are various ways of doing things and some are faster than others.

5 Preprocessor instructions

We have seen already some parts of the preprocessor. All instructions that start with the character # belong to the preprocessor except for what starts with #: which are settings at the startup of the program. Thusfar we have seen #define and #do/#enddo. Also the $\cdots$ operator belongs to the preprocessor and the preprocessor has its own variables and its own calculator.

Let us first study the complete $\cdots$ operator. It is between two separators which will be repeated: $,\cdots,\ +\cdots+\ -\cdots-\ \star\cdots\star\ /\cdots/\ +\cdots-\ -\cdots+$. The last two mean that there will be an alternatig sign, the first one being + resp -. The pattern that is to be set up is in general given between <>. Hence we have <pattern1>, *cdots*, <pattern2>, Form then looks how this can be generalized and would make pattern1,pattern2 in this case. In simple cases where there is a single number at the end the <> can be omitted as in x1, $\cdots$,x5 We can use this for the previous program Example ex6a.frm:

```
CF den;
S  x,x1,x2;
L  F = <den(x+(1))^1>*...*<den(x+(8))^8>;
SplitArg,den;
Print;
.sort
#do i = 1,1
  id  den(x1?!{x2?},x)*den(x2?!{x1?},x) =
 (den(x1,x)-den(x2,x))*den(x2-x1);
  id  den(x?number_) = 1/x;
  if ( match(den(x1?!{x2?},x)*den(x2?!{x1?},x)) );
 redefine i "0";
  endif;
  .sort
#enddo
Print +s;
.end
```

which gives for its final statistics

```
Time =       0.04 sec    Generated terms =         36
                F        Terms in output =         36
                         Bytes used      =       1160
```

Note: we have to be careful with x+1 as it picks the +1 as a number. It then reconstructs it without the leading sign. This can be circumvented with parentheses. Notice that it is now also easy to run the whole thing with a preprocessor variable:

```
#define MAX "10"
L  F = <den(x+(1))^1>*...*<den(x+(`MAX'))^`MAX'>;
```

When preprocessor variables are used their name should be enclosed between `' (back-quote, quote). This can be nested. The value of a preprocessor variable is a string. It will just paste strings together and look for new variables: Example ex6b.frm:

```
    #define i1 "x"
    #define i2 "y"
    #define i3 "z"
    #do j = 1,3
    #message i`j' = `i`j''
~~~i1 = x
    #enddo
~~~i2 = y
~~~i3 = z
    .end
```

Sometimes it is necessary to interpret the preprocessor variables numerically and do some arithmetic with them. For that we have the preprocessor calculator. It is invoked with and a purely numerical expression between the as in Example ex6c.frm:

```
#define MAX "4"
Symbols x1,...,x{2*‘MAX’+1};
CF f;
Local F = f(x1,...,x{2*‘MAX’+1});
Print;
.end

Time =       0.00 sec    Generated terms =          1
                F        Terms in output =          1
                         Bytes used      =         52

   F =
      f(x1,x2,x3,x4,x5,x6,x7,x8,x9);
```

The variable MAX is interpreted as a number, the arithmetic is done and the result is translated back into a string. This should explain now also why a set with only a number needs the comma to do as if there is another element. The comma blocks the invokation of the preprocessor calculator.

The preprocessor has also a #if #elseif #else #endif and a #switch #case #break #default #endswitch construction.

Maybe the most important feature of the preprocessor is to give structure to the program. There are procedures that can be specified externally like subroutines in calculational languages. Example ex6d.frm:

```
#procedure normden(x,den)
SplitArg,((‘x’)),‘den’;
id ‘den’(‘x’) = ‘den’(0,‘x’);
#do inormden = 1,1
id,once,‘den’(x1?!{x2?},‘x’)*‘den’(x2?!{x1?},‘x’) =
        (‘den’(x1,‘x’)-‘den’(x2,‘x’))*‘den’(x2-x1);
id ‘den’(x1?number_) = 1/x1;
if ( match(‘den’(x1?!{x2?},‘x’)*‘den’(x2?!{x1?},‘x’)) );
              redefine inormden "0";
endif;
.sort:normden;
#enddo
id ‘den’(x1?,x2?) = ‘den’(x1+x2);
#endprocedure

CF yden;
S  y,x1,x2;
L  F = <yden(y+(1))^1>*...*<yden(y+(8))^8>;
#call normden(y,yden)

                           .
                           .
                           .
Time =       0.04 sec    Generated terms =         42
                F        Terms in output =         36
                 normden Bytes used      =       1160
    .end
```

```
Time =       0.04 sec    Generated terms =         36
               F         Terms in output =         36
                         Bytes used      =       1852
```

This gives the result we had before. Note that the procedure has arguments which inside the procedure are preprocessor variables. Outside the procedure (after it is finished) these variables don't exist any longer. We could also have put the procedure inside a file normden.prc In that case the #procedure instruction should be the first line of the file and there should be no other characters before the #. Our program becomes simply: Example ex6e.frm:

```
#define MAX "8"
CF yden;
S  y,x1,x2;
L  F = <yden(y+(1))^1>*...*<yden(y+('MAX'))^'MAX'>;
#call normden(y,yden)
Print +s;
.end
```

```
                              .
                              .
                              .
Time =       0.07 sec    Generated terms =         55
               F         Terms in output =         55
                         Bytes used      =       3420
```

Another useful preprocessor instruction is the #include file.h which includes on the spot the contents of the file file.h.

It goes without saying that all these features can be nested. There is one restriction. A #if and it matching #endif must be inside the same procedure or the #case in a #switch construction. Similarly the complete #switch #endswitch construction must be inside the same procedure or the same part of a #if construction.

There is also a #write instruction that allows to write in files. It can write text and expressions. Example ex6f.frm:

```
Symbols x,y,n;
L F = (x+y)^3+(x-y+2)^4;
id x^n? = x^(n+1)/(n+1);
.sort
```

```
Time =       0.00 sec    Generated terms =         19
               F         Terms in output =         15
                         Bytes used      =        234
```

```
Format doubleFortran;
#write <fun.f> "      REAL*8 FUNCTION fun(x,y)"
#write <fun.f> "      REAL*8 x,y"
#write <fun.f> "*\n*      Routine created by FORM, 'DATE_'\n*"
#write <fun.f> "      fun = %E",F
#write <fun.f> "      RETURN"
#write <fun.f> "      END"
Print +f;
.sort
```

```
Time =       0.00 sec    Generated terms =         15
               F         Terms in output =         15
                         Bytes used      =        234
```

```
      F =
     & 16.D0*x - 32.D0*x*y + 24.D0*x*y**2 - 7.D0*x*y**3 + x*y**4 + 16.D0
     & *x**2 - 24.D0*x**2*y + 27.D0/2.D0*x**2*y**2 - 2.D0*x**2*y**3 + 8.
     & D0*x**3 - 7.D0*x**3*y + 2.D0*x**3*y**2 + 9.D0/4.D0*x**4 - x**4*y
     &  + 1.D0/5.D0*x**5

    Format C;
    #write <fun.c> "double fun(double x,double y)"
    #write <fun.c> "/*\n    Function created by FORM, `DATE_'\n*/\n{"
    #write <fun.c> "    double f;"
    #write <fun.c> "    f = %E;",F
    #write <fun.c> "    return(f);\n}"
    Print +f;
    .end

Time =       0.00 sec    Generated terms =         15
               F         Terms in output =         15
                         Bytes used      =        234

   F =
      16*x - 32*x*y + 24*x*pow(y,2) - 7*x*pow(y,3) + x*pow(y,4) + 16*
      pow(x,2) - 24*pow(x,2)*y + 27./2.*pow(x,2)*pow(y,2) - 2*pow(x,2)*
      pow(y,3) + 8*pow(x,3) - 7*pow(x,3)*y + 2*pow(x,3)*pow(y,2) + 9./4.
      *pow(x,4) - pow(x,4)*y + 1./5.*pow(x,5);
```

We see here also the use of the built in preprocessor variable DATE_. Format is a statement that controls the outputformat.

The contents of the file fun.f are

```
      REAL*8 FUNCTION fun(x,y)
      REAL*8 x,y
*
*      Routine created by FORM, Mon Mar  5 13:12:31 2007
*
      fun = 16.D0*x - 32.D0*x*y + 24.D0*x*y**2 - 7.D0*x*y**3 + x*y**4
     &  + 16.D0*x**2 - 24.D0*x**2*y + 27.D0/2.D0*x**2*y**2 - 2.D0*x**2*
     & y**3 + 8.D0*x**3 - 7.D0*x**3*y + 2.D0*x**3*y**2 + 9.D0/4.D0*x**4
     &  - x**4*y + 1.D0/5.D0*x**5
      RETURN
      END
```

The contents of the file fun.c are

```
double fun(double x,double y)
/*
    Function created by FORM, Mon Mar  5 13:12:31 2007
*/
{
    double f;
    f = 16*x - 32*x*y + 24*x*pow(y,2) - 7*x*pow(y,3) + x*pow(y,4) + 16*
      pow(x,2) - 24*pow(x,2)*y + 27./2.*pow(x,2)*pow(y,2) - 2*pow(x,2)*
      pow(y,3) + 8*pow(x,3) - 7*pow(x,3)*y + 2*pow(x,3)*pow(y,2) + 9./4.
```

```
      *pow(x,4) - pow(x,4)*y + 1./5.*pow(x,5);
    return(f);
}
```

Another useful preprocessor instruction is #- which turns off the listing of the input. #+ turns it back on again. This can make lengthy programs much 'quieter'. If we put #- at the beginning of the previous example we have in the output example ex6g.log:

```
    #-

Time =       0.00 sec    Generated terms =         19
                F        Terms in output =         15
                         Bytes used      =        234

Time =       0.00 sec    Generated terms =         15
                F        Terms in output =         15
                         Bytes used      =        234

      F =
     & 16.D0*x - 32.D0*x*y + 24.D0*x*y**2 - 7.D0*x*y**3 + x*y**4 + 16.D0
     & *x**2 - 24.D0*x**2*y + 27.D0/2.D0*x**2*y**2 - 2.D0*x**2*y**3 + 8.
     & D0*x**3 - 7.D0*x**3*y + 2.D0*x**3*y**2 + 9.D0/4.D0*x**4 - x**4*y
     &  + 1.D0/5.D0*x**5


Time =       0.00 sec    Generated terms =         15
                F        Terms in output =         15
                         Bytes used      =        234

   F =
      16*x - 32*x*y + 24*x*pow(y,2) - 7*x*pow(y,3) + x*pow(y,4) + 16*
      pow(x,2) - 24*pow(x,2)*y + 27./2.*pow(x,2)*pow(y,2) - 2*pow(x,2)*
      pow(y,3) + 8*pow(x,3) - 7*pow(x,3)*y + 2*pow(x,3)*pow(y,2) + 9./4.
      *pow(x,4) - pow(x,4)*y + 1./5.*pow(x,5);
```

The instruction #include- file.h will include the file file.h without listing it.

6 Some more statements and functions

There are more than 100 types of statements so we will not treat all of them. Some are not so common and some can be learned from the manual just as well. Let us start with the if-statement which we have seen a few times already. The question is of course what conditions one can have in a symbolic program. We have seen match which gives the number of matches for a pattern. Another that is very useful is a 'powercount' as in

```
    if ( count(x,1,y,2,f,1,d,-2) ) > 0 );
```

Here we have each power of the symbol x count for 1, of the symbol y count for 2, of the function f count for 1 and of the function d count for -2. These weights are added to obtain the count.

Another condition can be whether a term belongs to a given expression as in

```
    if ( expresseion(F) );
```

In that case terms of other expressions are not considered. One can also specify integer numbers or the coefficient of the current term as in

```
if ( coefficient != 1 );
   Multiply 1/coeff_;
endif;
```

or $-variables which should evaluate to a numerical value. Example Example ex7a.frm:

```
    Symbols x,y,z;
    L F = (x+y+z)^5-(x+2*y)^5;
    .sort

Time =       0.00 sec    Generated terms =         27
                F        Terms in output =         20
                         Bytes used      =        364
    #$xc = 0;
    if ( count(x,1) > $xc ) $xc = count_(x,1);
    .sort

Time =       0.00 sec    Generated terms =         20
                F        Terms in output =         20
                         Bytes used      =        364
    #message The maximum power of x is `$xc'
~~~The maximum power of x is 4
    Print +f;
    Bracket x;
    .end

Time =       0.00 sec    Generated terms =         20
                F        Terms in output =         20
                         Bytes used      =        386

   F =
       + x * ( 5*z^4 + 20*y*z^3 + 30*y^2*z^2 + 20*y^3*z - 75*y^4 )

       + x^2 * ( 10*z^3 + 30*y*z^2 + 30*y^2*z - 70*y^3 )

       + x^3 * ( 10*z^2 + 20*y*z - 30*y^2 )

       + x^4 * ( 5*z - 5*y )

       + z^5 + 5*y*z^4 + 10*y^2*z^3 + 10*y^3*z^2 + 5*y^4*z - 31*y^5;
```

Note that dollar variables can be used as preprocessor variables if placed between ‘’. The function count_ works exactly like the count in the if but it gives a value that can be used to put for instance in a $-variable.

If the condition in an if statement is composite one should use brackets around each subcondition. If this is not done the compiler might get confused and interpret things its own way.

We also see an example of the bracket statement. It should be after the executable statements in the module. It controls some of the output format. The mentioned objects will be placed outside brackets. The rest inside.

As any good programming language form has a goto and a label statement. The label should have a name or a number and the goto must refer to a label inside the same module. We can use this to make our procedure normden twice as fast. File normden2.prc:

```
#procedure normden2(x,den)
```

```
SplitArg,((‘x’)),‘den’;
id ‘den’(‘x’) = ‘den’(0,‘x’);
#do inormden = 1,1
  id,once,ifmatch->1,‘den’(x1?!{x2?},‘x’)*‘den’(x2?!{x1?},‘x’) =
        (‘den’(x1,‘x’)-‘den’(x2,‘x’))*‘den’(x2-x1);
  goto 2;
 Label 1;
     redefine inormden "0";
 Label 2;
     id ‘den’(x1?number_) = 1/x1;
 .sort:normden;
#enddo
id ‘den’(x1?,x2?) = ‘den’(x1+x2);
#endprocedure
```

The option in the id statement says to jump to label 1 if there is a match and after the whole substitution has been completed. This way we don't have to do the pattern matching twice (once for the relabel statement). Notice however that it does cost us some readability of the program. But when speed is at a premium......

6.1 Arguments

Sometimes one would like to treat the arguments of some functions with some statements but nothing else. For this there is the Argument/EndArgument construction. In the argument statement we can specify what we want to treat:

```
Argument,f,2;
```

means just the 2-nd argument of the function f.

```
Argument f,g,1,h,3;
```

means the first argument of the functions f and g and the third of h.

```
Argument f;
```

means all arguments of the function f.

```
Argument;
```

means all arguments of all functions. Example (ex7b.frm)

```
CF f,g;
Symbol x,y;
L F = f(x+y,x+y,x+y)+g(x+y,x+y,x+y);
argument,f,2;
  id x = x+1;
endargument;
Print +s;
.end
```

```
Time =       0.00 sec    Generated terms =          2
                F        Terms in output =          2
                         Bytes used      =        278
```

```
F =
    + f(y + x,1 + y + x,y + x)
    + g(y + x,y + x,y + x)
   ;
```

Note that

```
Argument,2,f;
```

would mean that all functions have their second argument treated except for f which has all its arguments treated.

Argument 'environments' can be nested.

6.2 Replace_

There are many built in functions which do exactly what their name says like fac_ is a factorial, binom_ is a binomial, theta_ is a theta function and delta_ is the dirac delta function etc. One very useful function should be mentioned. Because `id x = y;` only affects occurrences of x outside functions it would be very complicated to replace all occurrence of x by y. For this we have the replace_ function:

```
multiply replace_(x,y,a,b,b,a);
```

acts like integration over dirac delta functions: all occurrences of x get replaced by y, all occurences of a by b and all of b by a (hence a and b are exchanged). This is the fastest way to do so. Be careful with using more than one replace_ function at the same time. It is hard to predict which one FORM will apply first.

Other built in functions include the dirac gamma matrices. We will leave them for study from the manual.

6.3 Tables

FORM is also equiped with tables. Tables are objects with at least one index and can also have (wildcard) arguments. They have to be declared and there are two types of tables. The regular tables are like arrays where one spot is reserved for each element. The sparse tables have no reserved spots and each element is taken as it comes. They are stored in a balanced tree to make searches for table elements not too slow. Let us look at a few examples. Example ex8a.frm:

```
Symbols x,y,n1,n2;
Table t(0:2,0:2);
Fill t(0,0) = 1;
Fill t(0,1) = 2;
Fill t(0,2) = 3;
Fill t(1,0) = 2;
Fill t(1,1) = 3;
Fill t(1,2) = 4;
Fill t(2,0) = 3;
Fill t(2,1) = 4;
Fill t(2,2) = 5;
Local F = (1+x+y)^2;
print;
.sort
```

```
Time =       0.00 sec    Generated terms =          6
```

```
               F       Terms in output =         6
                       Bytes used      =        88

   F =
      1 + 2*y + y^2 + 2*x + 2*x*y + x^2;

    id  x^n1?*y^n2? = t(n1,n2);
    Print;
    .end

Time =       0.00 sec    Generated terms =         6
               F         Terms in output =         1
                         Bytes used      =        10

   F =
      21;
```

As can be seen, the substitutions of the table elements take place immediately. What happens if an element is not in the table? In that case FORM leaves the element. Example ex8b.frm:

```
    Symbols x,y,n1,n2;
    Table t(0:2,0:2);
    Fill t(0,0) = 1;
    Fill t(0,1) = 2;
    Fill t(0,2) = 3;
    Fill t(1,0) = 2;
    Fill t(1,1) = 3;
    Fill t(1,2) = 4;
    Fill t(2,0) = 3;
    Fill t(2,1) = 4;
    Fill t(2,2) = 5;
    Local F = (1+x+y)^3;
    id  x^n1?*y^n2? = t(n1,n2);
    Print;
    .end

Time =       0.00 sec    Generated terms =        10
               F         Terms in output =         3
                         Bytes used      =        48

   F =
      73 + t(0,3) + t(3,0);
```

unless we apply the check option in the definition of the table. Example ex8c.frm:

```
    Symbols x,y,n1,n2;
    Table,check,t(0:2,0:2);
    Fill t(0,0) = 1;
    Fill t(0,1) = 2;
    Fill t(0,2) = 3;
    Fill t(1,0) = 2;
    Fill t(1,1) = 3;
    Fill t(1,2) = 4;
    Fill t(2,0) = 3;
    Fill t(2,1) = 4;
```

```
    Fill t(2,2) = 5;
    Local F = (1+x+y)^3;
    id  x^n1?*y^n2? = t(n1,n2);
    Print;
    .end
Table boundary check. Argument 1
      t(3,0)
```

After the message is printed execution is halted. An example of a table with an argument is in example ex8d.frm:

```
    Symbols x,y,n;
    Table,check,tlog(-2:3,x?);
    Fill tlog(-2) = -x^-1;
    Fill tlog(-1) = ln_(x);
    Fill tlog(0) = x;
    Fill tlog(1) = x^2/2;
    Fill tlog(2) = x^3/2;
    Fill tlog(3) = x^4/2;
    Local F = (1+y)^4/y^2;
    id y^n? = tlog(n,y);
    Print +s;
    .end

Time =       0.00 sec    Generated terms =          5
                F        Terms in output =          5
                         Bytes used      =         72

   F =
       - y^-1
       + 6*y
       + 2*y^2
       + 1/2*y^3
       + 4*ln_(y)
      ;
```

The right hand sides of the fill statements can also contain tables. The only rule here is that one should try to avoid loops as that will cause a crash. Example ex8e.frm:

```
    Symbols x,y;
    Table ch(0:6,x?);
    Fill ch(0) = 1;
    Fill ch(1) = x;
    Fill ch(2) = x*ch(1,x)+(x+1)*ch(0,x);
    Fill ch(3) = x*ch(2,x)+(x-1)*ch(1,x);
    Fill ch(4) = x*ch(3,x)+(x+1)*ch(2,x);
    Fill ch(5) = x*ch(4,x)+(x-1)*ch(3,x);
    Fill ch(6) = x*ch(5,x)+(x+1)*ch(4,x);
    Local F5 = ch(5,y);
    Local F6 = ch(6,y);
    Print +f;
    .end

Time =       0.00 sec    Generated terms =         21
               F5        Terms in output =          4
                         Bytes used      =         54
```

```
Time =       0.00 sec    Generated terms =         43
               F6        Terms in output =          7
                         Bytes used      =         86

   F5 =
      y + 3*y^3 + 4*y^4 + y^5;

   F6 =
      1 + 3*y + 5*y^2 + 5*y^3 + 7*y^4 + 5*y^5 + y^6;
```

When tables are multi-dimensional and/or not elements are known, it is usually better to use sparse tables. In sparse tables we tell FORM only the dimension of the table and the possible arguments. Example ex8f.frm:

```
Symbols x1,x2,x3,x4,n1,n2,n3,n4,N;
Table sparse,t(4);
Fill t(1,1,1,1) = N+1;
Fill t(1,2,1,2) = N+2;
Fill t(1,1,2,1) = N^2-1;
Local F = x1*x2*x3*x4*(1+x1)*(1+x2)*(1+x3)*(1+x4);
id x1^n1?*x2^n2?*x3^n3?*x4^n4? = t(n1,n2,n3,n4);
id t(n1?,n2?,n3?,n4?) = x1^n1*x2^n2*x3^n3*x4^n4;
Print +f +s;
.end
```

```
Time =       0.00 sec    Generated terms =         19
                F        Terms in output =         16
                         Bytes used      =        254

   F =
       + 2
       + 2*N
       + N^2
       + x1*x2*x3*x4^2
       + x1*x2*x3^2*x4^2
       + x1*x2^2*x3*x4
       + x1*x2^2*x3^2*x4
       + x1*x2^2*x3^2*x4^2
       + x1^2*x2*x3*x4
       + x1^2*x2*x3*x4^2
       + x1^2*x2*x3^2*x4
       + x1^2*x2*x3^2*x4^2
       + x1^2*x2^2*x3*x4
       + x1^2*x2^2*x3*x4^2
       + x1^2*x2^2*x3^2*x4
       + x1^2*x2^2*x3^2*x4^2
      ;
```

Here we try what is in the table. The elements that are not in the table are written back and will have to be dealt with by other means. Very often one puts the tables in separate files that can be extended when more table elements become known. If these files become very big (megabytes) there is a facility which is called the tablebase. It is a database feature for tables in which FORM at first only looks which table elements exist but does not compile the right hand sides yet. At a moment of the users choice FORM can then decide what elements are actually needed and only compile those. The use of this can be looked up in the manual. During a recent project we had tablebases containing more than 3 Gbytes of fill statements.

6.4 Collect,PolyFun

One of the reasons of the speed of FORM is the fact that it works its way through expressions one by one. Each operation has for its input just a single term. Those are called local operations. The major nonlocal operation is of course the sort that brings expressions to a standard form. It is amaizing what can be done with just this single nonlocal operation. Yet there are cases in which we could use, to great benefit, a nearly local operation. Have a look at the following. Example ex9a.frm:

```
    Symbols x,ep(:4);
    CFunction acc;
    Local F = (x+1+ep)^5;
    Bracket x;
    Print;
    .sort

Time =       0.00 sec    Generated terms =         20
                F        Terms in output =         20
                         Bytes used      =        308

   F =
       + x * ( 5 + 20*ep + 30*ep^2 + 20*ep^3 + 5*ep^4 )

       + x^2 * ( 10 + 30*ep + 30*ep^2 + 10*ep^3 )

       + x^3 * ( 10 + 20*ep + 10*ep^2 )

       + x^4 * ( 5 + 5*ep )

       + x^5 * ( 1 )

       + 1 + 5*ep + 10*ep^2 + 10*ep^3 + 5*ep^4;

    Collect acc;
    Print +s;
    .end

Time =       0.00 sec    Generated terms =          6
                F        Terms in output =          6
                         Bytes used      =        414

   F =
       + acc(1 + 5*ep + 10*ep^2 + 10*ep^3 + 5*ep^4)
       + acc(5 + 5*ep)*x^4
       + acc(5 + 20*ep + 30*ep^2 + 20*ep^3 + 5*ep^4)*x
       + acc(10 + 20*ep + 10*ep^2)*x^3
       + acc(10 + 30*ep + 30*ep^2 + 10*ep^3)*x^2
       + acc(1)*x^5
      ;
```

First we see in the declaration of ep a new option. This one indicates that we will consider powers of ep up to 4. Higher powers will be automatically removed. The bracket statement indicates that we want to print the output with x outside brackets. This may improve the readability. During the sorting FORM takes this into account and all terms inside the same bracket are put together. FORM prints the output in the order that the terms have after the sorting. The collect statement tells FORM put the contents of the brackets inside the indicated function acc. This means that the collect statement takes a number of adjacent (hence nearly local) terms for its input and gives a single term as its output. We could process the new terms further as in the next job. Example ex9b.frm:

```
    Symbols x,ep(:4),x1,x2;
    CFunction acc;
    Local F = (x+1+ep)^5;
    Bracket x;
    .sort

Time =       0.00 sec    Generated terms =         20
                F        Terms in output =         20
                         Bytes used      =        308
    Collect acc;
    Splitarg,acc;
    Print +s;
    .sort

Time =       0.00 sec    Generated terms =          6
                F        Terms in output =          6
                         Bytes used      =        430

   F =
       + acc(1)*x^5
       + acc(1,5*ep,10*ep^2,10*ep^3,5*ep^4)
       + acc(5,5*ep)*x^4
       + acc(5,20*ep,30*ep^2,20*ep^3,5*ep^4)*x
       + acc(10,20*ep,10*ep^2)*x^3
       + acc(10,30*ep,30*ep^2,10*ep^3)*x^2
      ;

    Repeat id acc(x1?,x2?,?a) = acc(x1)*acc(x2,?a);
    Print +s;
    .end

Time =       0.00 sec    Generated terms =          6
                F        Terms in output =          6
                         Bytes used      =        438

   F =
       + acc(1)*x^5
       + acc(1)*acc(5*ep)*acc(10*ep^2)*acc(10*ep^3)*acc(5*ep^4)
       + acc(5)*acc(5*ep)*x^4
       + acc(5)*acc(20*ep)*acc(30*ep^2)*acc(20*ep^3)*acc(5*ep^4)*x
       + acc(10)*acc(20*ep)*acc(10*ep^2)*x^3
       + acc(10)*acc(30*ep)*acc(30*ep^2)*acc(10*ep^3)*x^2
      ;
```

And whatever we want with it....

We can also consider the contents of the function as the coefficient of the term. This would be equivalent to not using the collect statement at all, but in that case we have more terms and in the sequel we may need much more pattern matching. Example ex9c.frm:

```
    Symbols ep(:3),x,y;
    CF acc,den;
    Local F = x*den(1+ep)+x^2*den(1+2*ep)+x^3*den(1-3*ep);
    Splitarg,((ep)),den;
    Print;
    .sort

Time =       0.00 sec    Generated terms =          3
               F         Terms in output =          3
                         Bytes used      =        102

   F =
      den(1, - 3*ep)*x^3 + den(1,2*ep)*x^2 + den(1,ep)*x;

    repeat id den(x?,y?) = den(x)-y*den(x)*den(x,y);
    id den(x?) = 1/x;
    Abracket ep;
    Print;
    .sort

Time =       0.00 sec    Generated terms =         12
               F         Terms in output =         12
                         Bytes used      =        186

   F =
       + x * ( 1 - ep + ep^2 - ep^3 )

       + x^2 * ( 1 - 2*ep + 4*ep^2 - 8*ep^3 )

       + x^3 * ( 1 + 3*ep + 9*ep^2 + 27*ep^3 );

    PolyFun acc;
    Collect acc;
    Print +s;
    .sort

Time =       0.00 sec    Generated terms =          3
               F         Terms in output =          3
                         Bytes used      =        240

   F =
       + x*acc(1 - ep + ep^2 - ep^3)
       + x^2*acc(1 - 2*ep + 4*ep^2 - 8*ep^3)
       + x^3*acc(1 + 3*ep + 9*ep^2 + 27*ep^3)
      ;

    id x = 1/2;
    Print +s;
    .end

Time =       0.00 sec    Generated terms =          3
               F         Terms in output =          1
                         Bytes used      =         76
```

```
F =
    + acc(7/8 - 5/8*ep + 21/8*ep^2 + 7/8*ep^3)
   ;
```

Here we use the function den as a denominator function and we expand it in ep. We see that this works well. Then we declare that acc is the PolyFun which stands for polynomial function. It means that its argument is the coefficient of the term. It automatically also means that `2*acc(x)` is replaced by `acc(2*x)`, etc. One can check that the addition (with the value for x) in the end went correctly. The advantage is when we have to first do a lot of work with the powers of x. Before each power had to be treated 4 times, now only once. At times this can cause great savings.

7 Massless propagator graphs

One of the very successful FORM programs is the MINCER program for three loop massless propagators. A description of how the routines work is given in the separate documentation file mincer.ps. It describes the various topologies of the diagrams and how each topology is solved, either directly or via reduction into simpler topologies. The answer is given as an expansion in $\epsilon = 2 - D/2$ in which D is the dimension of space-time. The three loop topologies LA (ladder), BE (benz) and NO (non-planar) have to be evaluated to order 1. The three loop topologies FA and BU should in principle be known to order ϵ and the three loop topologies O1, O2, O3 and O4 to order ϵ^2. Finally the three loop topologies Y1, Y2, Y3, Y4 and Y5 could be needed to order ϵ^3 although such cases are very rare. Basically the elimination of a line in a topology creates a diagram of a simpler topology. At the same time such an elimination can give a factor $1/\epsilon$, hence the needed accuracy because the following hierarchy exists when one line gets eliminated:

- NO $\rightarrow$ FA,BU.
- BE $\rightarrow$ FA,BU,O1,O2.
- LA $\rightarrow$ FA,O2,O3.
- FA $\rightarrow$ O1,O2,Y3,Y4.
- BU $\rightarrow$ O2,O4,Y1,Y3.
- O1 $\rightarrow$ Directly down to T1.
- O2 $\rightarrow$ Directly down to T1.
- O3 $\rightarrow$ Directly down to T1.
- O4 $\rightarrow$ Y2,Y3 or directly down to T1.

We see that for instance Y2 can be obtained via $BE \rightarrow BU \rightarrow O4 \rightarrow Y2$ with a factor $1/\epsilon^3$ but in practise we use the direct reduction to the two loop topology T1 and hence in the current setup we need the Y topologies effectively only to order $1/\epsilon^2$.

The two loop topologies are T1, T2 and T3 where we can reduce T1 to T2 and or T3 at the cost of a factor $1/\epsilon$. Each one loop (sub)integral that we do can give a factor $1/\epsilon$. Hence the T1 integral can have a factor $1/\epsilon^3$ and the T2,T3 integrals can have a factor $1/\epsilon^4$. The T2,T3 integrals can be reduced by integration to the one loop topology L1 at the cost of a potential factor $1/\epsilon$ and finally the L1 integral can give a factor $1/\epsilon$. In all, intermediate results may have to be expanded to order ϵ^6 and different integrals are needed to different accuracies. We obtain these accuracies by multiplying a given integral by $1/\epsilon^n$ with n the accuracy we need. Hence a T1 integral is multiplied by $1/\epsilon^3$. We multiply again by ϵ^3 in the end after the rounding.

For diagrams the above accuracies are too much. There we have to consider that if a three loop integral is needed to order 1, two loop integrals are needed to order ϵ, one loop integrals are needed to order ϵ^2 and tree graphs are needed to order ϵ^3. The excess accuracy that is needed for the individual integrals is due to the poles that are artifacts of the method.
So how do we use MINCER?
Any program that uses MINCER routines should have at or near the beginning the instruction

```
#include- mincer.h
```

The file mincer.h contains

- All necessary declarations of variables.
- All physics independent procedures that are needed.
- Some programs for the extension of tables, should the need arise.

All internal variables of MINCER have names that start with mnc to minimize the possibilities of name conflicts with the parts of the program defined by the user. In addition a few external variables are defined for communication. They include the vectors p1, $\cdots$,p8, Q, P, $[P \pm Q]$ and $[P \pm p1] \cdots [P \pm p8]$. For the moment we will ignore the vector P.

An example of a program that would run the ladder integral

$$LA(2,2,2,2,2,2,2,2,0) \quad = \quad d^Dp_1d^Dp_2d^Dp_3Q \cdot Q^10/p_1 \cdot p_1^2/\cdots/p_8 \cdot p_8^2$$

would be (ex10a.frm)

```
    #define TOPO "la"
    #define SCHEME "0"
    #include- mincer.h
    off statistics;
    .global
    Local F = Q.Q^10/p1.p1^2/p2.p2^2/p3.p3^2/p4.p4^2
              /p5.p5^2/p6.p6^2/p7.p7^2/p8.p8^2;
    Multiply ep^3;
    #call integral('TOPO')
~~~Answer in MS-bar
    .sort
    On Statistics;
    Print +s;

    .end

Time =       0.47 sec    Generated terms =          5
                F        Terms in output =          5
                         Bytes used      =         80

   F =
       - 389662969/13500
       - 4964/3*ep^-3
       - 80739/5*ep^-2
```

```
      - 56411291/1350*ep^-1
      + 325708/3*z3
     ;
```

Usually integrals are not this complicated though.

To do an integral of the type O1 as in

$$O1(2,2,2,2,2,2,2,0,0) \quad = \quad d^D p_1 d^D p_2 d^D p_3 Q \cdot Q^8/p_1 \cdot p_1^2/\cdots/p_7 \cdot p_7^2$$

would give the program (ex10b.frm)

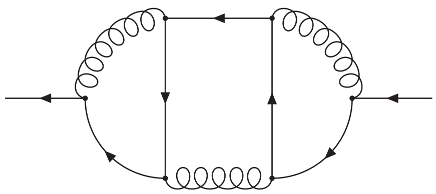
```
    #define TOPO "o1"
    #define SCHEME "0"
    #include- mincer.h
    off statistics;
    .global
    Local F = Q.Q^8/p1.p1^2/p2.p2^2/p3.p3^2/p4.p4^2
              /p5.p5^2/p6.p6^2/p7.p7^2;
    Multiply ep^3;
    Multiply 1/ep^2;
    #call integral('TOPO')
~~~Answer in MS-bar
    .sort
    On Statistics;
    Multiply ep^2;
    Print +s;
    .end

Time =       0.14 sec    Generated terms =         12
                F        Terms in output =         12
                         Bytes used      =        212

   F =
      - 3404689/20250
      - 244/3*ep^-3
      - 25871/45*ep^-2
      - 207466/675*ep^-1
      + 18956/3*z3
      + 572858047/30375*ep
      + 8624*ep*z4
      + 1804357/45*ep*z3
      + 1402010104781/18225000*ep^2
      + 86352*ep^2*z5
      + 177698/3*ep^2*z4
      + 48870626/675*ep^2*z3
     ;
```

Note now the multiplication with $1/\epsilon^2$ before the integration and with ϵ^2 afterwards.

The program for a complete diagram we do somewhat differently. Here we feed the diagram and the topology etc in via an include file. The reason is that we can generate such files for each diagram in a calculation without having to change the actual program. We need now also an extra procedure that substitutes the Feynman rules and that makes projections if needed. This is the procedure treat.prc. It calls the necessary procedures in the MINCER library. Hence the program looks like (calcdia.frm):

```
#-
#define GAUGE "0"
#define SCHEME "0"
#include mincer.h
*
* Now some variables that are calcdia specific
*
S sgn,n,[n-4],s,proexp,eq;
AutoDeclare Symbol xx, SGN, sgn, z, x, k, y, xdia;
AutoDeclare CFunction DL;
AutoDeclare index ii;
F fp,vqg;
CF signs,del,fxn,qpow,vgh,Dg,Dgh,v3g,epexp;
V Q,p;
S fermi1,fermi2,fermi3,gluon1,gluon2,gluon3,ghost1,ghost2,ghost3;
CF Dg,v2gp,v2gi,v2gc,v3g,v3gp,Ds,DL,v4g,V4G,v3gc,v3gi,v4g,v4gp,v4gc,withp;
I K1,K2;
Off Statistics;
.global
#include diagram.h
;
multiply ep^3;
multiply i_;
#call treat
.sort
#call integral(`TOPO')
id mncxi = 0;
id  xi = 0;
.sort
On Statistics;
#call trim(`TOPO')
id xi = 0;
print;
.end
```

There are many declarations here. There are the functions for one, two and three loop propagator subgraphs. There are functions for propagators and vertices etc. The parameter xi is a gauge parameter. If we put the parameter GAUGE equal to -1 we get all powers of the gauge parameter. If we want to calculate the diagram



this would lead to the file diagram.h with the contents

```
L  d1c=+vqg(1,mu1)*fp(1,p6)*vqg(1,mu2)*fp(1,p7)*vqg(1,mu3)*fp(1,p2)*
      vqg(1,mu4)*fp(1,p8)*vqg(1,mu5)*fp(1,p4)*
      vqg(1,mu6)*Dg(mu1,mu3,p1)*Dg(mu2,mu5,p5)*Dg(mu4,mu6,p3)
#define NAME "d1c"
#define TOPO "la"
```

and the run would give

```
    #-
~~~Answer in MS-bar

Time =       0.18 sec    Generated terms =          5
              d1c        Terms in output =          5
                         Bytes used      =         64

   d1c =
       - 1411/6 - 8/3*ep^-3 - 22/3*ep^-2 - 121/3*ep^-1 + 544/3*z3;
```

Having all this it becomes clear that taking the color factors into account when we add the diagrams it should be easy now to calculate the three loop quark, gluon and ghost diagrams, even with a gauge parameter. Actually the three loop quark and gluon propagators are already included in the MINCER library. All steps in these calculations can be automatized:

- The Feynman diagrams are generated with a program named QGRAF.
- They can be given a notation with a FORM program.
- The color factor can be computed with the procedures in the library color.h.
- The ready to run diagrams are stored by a database program minos.
- The minos program then runs the diagrams one by one each time storing the result in a database.
- Finally minos generates the sum of the diagrams taking into account the color factor and other symmetry factors.