SIX

Originally FORM was designed to be the spider in a web, meant to calculate crosssections at the one loop level automatically. This got sidetracked a bit. By the time the (3-loop QCD) smoke cleared others had started making such systems as well and hence I decided against starting yet another system but get involved in one of the early systems, called GRACE. Hence in this session we will encounter some things that come from that system.

Let us first have a look at what is needed when you do some crosssection calculations and you want to do them automagically.

Imagine the way you would like to program your matrix element is by typing the amplitude and then square it, take the traces and make some simplifications. Which steps should you take?

In principle this is all there is to it:

- Select a notation.
- Define some functions that represent the Feynman rules.
- Paste these functions together in defining the diagrams.
- Square the amplitude.
- String the gamma matrices together and take the traces.
- Redefine the variables into something that is acceptable for FORTRAN or C.
- Write the output to a proper subroutine.
- Integrate numerically.

For the sake of simplicity we will start with the reaction $e^-e^+ \to \mu^-\mu^+$. After that we will go to Bhabha scattering.

Most of the work will be in a procedure 'squareamplitude'. We put this procedure in a file amplitude.h and in that file we also put all declarations. This means that the beginning of our first program will look like:

```
#-
*
*   Program for working out the matrix element squared for the
*   reaction e-e+ -> mu-mu+ in QED.
*
#include amplitude.h
*
L   Amp = VB(i1,pp,me)*g(i1,i2,j1)*U(i2,pe,me)
         *phprop(j1,j2,q)
         *UB(i3,pmm,mm)*g(i3,i4,j2)*V(i4,pmp,mm)
        ;
#call squareamplitude(Amp,M)
.sort
```

Notation: Clearly U,UB,V,VB are the spinors in the obvious way. The indexes i are the spinor indexes and g is the gamma matrix as in `g(i1,i2,mu)` $= \gamma^{\mu}_{i_1 i_2}$. The function g is commuting. The electron mass is 'me' and the muon mass 'mm'. The momenta are pe (e^-), pp (e^+), pmm (μ^-) and pmp (μ^+). The j are Lorenz indexes. Finally the photon propagator is 'phprop' and q is the momentum of the photon.

The file amplitude.h starts with the declarations:

```
AutoDeclare Index i,j,k;
AutoDeclare Symbol m,x;
AutoDeclare Vector p,q;
CFunction UB,U,VB,V,g,gstring,e;
CFunction fprop,phprop,gprop,prop;
Index k5=0, k6=0, k7=0;
*
```

There are more declarations here than we need for our simple reaction, because we want to be able to handle more complicated cases as well. The k indexes are for dealing with γ_5. We will come to that.

The first problem we will run into when we square the amplitude is the doubling of the indexes. Whereas the amplitude has two occurrences of all indexes, the squared amplitude would have four of them if we do not take precautions. That means that first we have to figure out how many i and how many j indexes are present. This would be possible with code like

```
#$imax = 0;
#do i = 1,40
    if ( match(VB(i‘i’,?a)) || match(V(i‘i’,?a))
         || match(UB(i‘i’,?a)) || match(U(i‘i’,?a))
         || match(g(i‘i’,?a)) || match(g(i?,i‘i’,?a))
         || match(fprop(i‘i’,?a)) || match(fprop(i?,i‘i’,?a)) );
            $imax = ‘i’;
    endif;
#enddo
#$jmax = 0;
#do j = 1,20
    if ( match(g(?a,j‘j’)) || match(phprop(j‘j’,?a))
        || match(phprop(j?,j‘j’,?a)) );
        $jmax = ‘j’;
    endif;
#enddo
```

in which we assume that we have kept the numbering such that we have limited the i indexes to numbers below 41 and the j indexes to numbers below 21. If we make this assumption there

is however a simpler way with a single statement. Hence the construction of the conjugate is done with

```
Skip;
Local ‘Amp’C = ‘Amp’;
id  i_ = -i_;
*
*   Make a new set of summed over indexes
*
Multiply replace_(<i1,ii1>,...,<i40,ii40>,
                  <j1,jj1>,...,<j20,jj20>);
*
*   Exchange rows and columns
*
id  g(i1?,i2?,j?) = g(i2,i1,j);
id  fprop?{fprop,phprop,gprop}(i1?,i2?,?a) = fprop(i2,i1,?a);
*
*   and exchange U and UB, V and VBAR
*
```

```
Multiply replace_(UB,U,U,UB,VB,V,V,VB);
*
*   gamma5 gets a minus sign. Hence k6 <--> k7
*
Multiply replace_(k6,k7,k7,k6);
id  g(?a,k5) = -g(?a,k5);
.sort
```

We use k5 for γ^5, k6 for $\gamma^6 = 1+\gamma^5$ and k7 for $\gamma^7 = 1-\gamma^5$. Those are indexes we do not sum over. Hence they have dimension zero. If you were to use complex variables (like a complex mass or coupling constant) you would have to add some extra statements here to replace the mass or coupling constant by its conjugate.

The next step is to multiply the amplitude and its conjugate after which we can do the spin sums and substitute the propagators.

```
*
*   Now multiply Amp and AmpC to get the matrix element squared.
*
Skip;
Drop,‘Amp’,‘Amp’C;
Local ‘Mat’ = ‘Amp’*‘Amp’C;
*
*   Spin sums
*
id  U(i1?,p?,m?)*UB(i2?,p?,m?) =  g(i1,i2,p)+g(i1,i2)*m;
id  V(i1?,p?,m?)*VB(i2?,p?,m?) = -g(i1,i2,p)+g(i1,i2)*m;
id  e(j1?,p?)*e(j2?,p?) = -d_(j1,j2);
*
*   Propagators
*
id  fprop(i1?,i2?,p?,m?) = (g(i1,i2,p)+g(i1,i2)*m)*prop(p.p-m^2);
```

```
id  phprop(j1?,j2?,q?) = -d_(j1,j2)*prop(q.q);
id  gprop(j1?,j2?,q?) = -d_(j1,j2)*prop(q.q);
```

The function e is the polarization vector of an external vector particle. In this case we assume that it will be a photon in which case current conservation allows us to drop all terms in the projection operator except for the $-g^{j_1 j_2}$. With gluons and massive vector particles one has to be more careful. If there is a single external gluon one can use this projection operator, but if there is more than one it is necessary to either use the full physical projection operator, or to work with ghost contributions (this gives much shorter expressions). In the above code we use gprop for a possible gluon propagator.

Having substituted the Feynman rules we have to worry about the gamma matrices and how to take their traces.

```
*
*   String the gamma matrices together in traces.
*
repeat id g(i1?,i2?,?a)*g(i2?,i3?,?b) = g(i1,i3,?a,?b);
.sort
Skip;
NSkip ‘Mat’;
*
*   Now put the traces one by one in terms of the built in gammas
*
#do i = 1,10
    id,once,g(i1?,i1?,?a) = g_(‘i’,?a);
    id  g_(‘i’,k7) = g7_(‘i’);
    al  g_(‘i’,k6) = g6_(‘i’);
    al  g_(‘i’,k5) = g5_(‘i’);
#enddo
```

```
.sort
*
*   Finally take the traces
*
#do i = 1,10
Trace4,‘i’;
#enddo
```

We assume that there will be no more than 10 trace lines. This is already a gross overestimation. The replacement of the k5,k6,k7 needs some care. If we use id statements in all three substitutions, FORM will move the γ^7 matrices to the left in the string, assuming that the k6 and k5 represent regular gamma matrices. We do not want that. Hence we first have to take out all k7,k6,k5 before we insert the RHS's and that is done with the al(so) statements. Finally we just take the traces.

Now we have dealt with the complete squareamplitude procedure. If we run the program and print the output after the procedure we obtain

```
M =
     + 64*prop(q.q)^2*me^2*mm^2
     + 32*prop(q.q)^2*pp.pe*mm^2
     + 32*prop(q.q)^2*pp.pmm*pe.pmp
     + 32*prop(q.q)^2*pp.pmp*pe.pmm
     + 32*prop(q.q)^2*pmm.pmp*me^2
    ;
```

in which we define `prop(x) = 1/x`.

We have to convert the notation to something a numerical program can work with efficiently. First we convert to symbols

```
.sort
S   s,t,u;
id  prop(q.q) = 1/s;
id  pe.pp = s/2-me^2;
id  pmm.pmp = s/2-mm^2;
id  pe.pmm = me^2/2+mm^2/2-t/2;
id  pp.pmp = me^2/2+mm^2/2-t/2;
id  pe.pmp = me^2/2+mm^2/2-u/2;
id  pp.pmm = me^2/2+mm^2/2-u/2;
id  u = 2*me^2+2*mm^2-s-t;
Print +f;
.sort
```

which gives the output

```
M =
   8 + 16*s^-2*t^2 + 16*s^-1*t - 32*mm^2*s^-2*t + 16*mm^4*s^-2
    - 32*me^2*s^-2*t + 32*me^2*mm^2*s^-2 + 16*me^4*s^-2;
```

Next we would like to make C output. This gives a little problem, because the C language does not have a power operator. Hence the output commands

```
Format C;
Print +f;
.sort
```

give

```
M =
   8 + 16*pow(s,-2)*pow(t,2) + 16*pow(s,-1)*t - 32*pow(mm,2)*pow(
   s,-2)*t + 16*pow(mm,4)*pow(s,-2) - 32*pow(me,2)*pow(s,-2)*t + 32*
   pow(me,2)*pow(mm,2)*pow(s,-2) + 16*pow(me,4)*pow(s,-2);
```

which is extremely inefficient. This can be greatly improved with

```
S   s2,t2,si,si2,me2,mm2,me4,mm4;
Format C;
id  s^2 = s2;
id  1/s^2 = si2;
id  1/s = si;
id  t^2 = t2;
```

```
id  me^2 = me2;
id  mm^2 = mm2;
id  me2^2 = me4;
id  mm2^2 = mm4;
Print +f;
.sort
```

resulting in

```
M =
   8 + 16*si2*mm4 + 16*si2*me4 + 32*si2*me2*mm2 + 16*t2*si2 - 32*t*
   si2*mm2 - 32*t*si2*me2 + 16*t*si;
```

This is fast code.

For a final step we have to convert this into a C subroutine. Let us assume for the moment that our variables me, me2, s, s2, si2 etc. have all been defined in the calling routine and are available as global variables. Only the variable t is passed as an argument because it may be changed in the Monte Carlo routine. We add the following code after the .sort

```
#write <eemm.c> "#include \"ourvariables.h\"\n"
#write <eemm.c> "double eemm(double t) {"
#write <eemm.c> "   double t2 = t*t;"
#write <eemm.c> "   double M;"
#write <eemm.c> "   M = %e",M
#write <eemm.c> "   return(M);"
#write <eemm.c> "}"
.end
```

and we find the file eemm.c with the contents

```
#include "ourvariables.h"

double eemm(double t) {
   double t2 = t*t;
```

```
    double M;
    M = 8 + 16*si2*mm4 + 16*si2*me4 + 32*si2*me2*mm2 + 16*t2*si2 -
       32*t*si2*mm2 - 32*t*si2*me2 + 16*t*si;

    return(M);
}
```

This file is ready for compilation provided the file ourvariables.h has been prepared in advance.

Having constructed the squareamplitude procedure we can deal with more complicated reactions. The Bhabha scattering diagrams show us a new feature. The cross graphs, when we square the amplitude, will have only a single trace. This is why we have to be so careful with the gamma matrices. The new program is

```
#-
*
*   The matrix element for the reaction e-e+ -> e-e+ in QED
*   Note the - sign between the diagrams.
*
#include amplitude.h
Format 68;
*
Local Amp = VB(i1,pp,me)*g(i1,i2,j1)*U(i2,pe,me)
          *phprop(j1,j2,q)
          *UB(i3,p1,me)*g(i3,i4,j2)*V(i4,p2,me)
         - UB(i1,p1,me)*g(i1,i2,j1)*U(i2,pe,me)
          *phprop(j1,j2,qq)
          *VB(i3,pp,me)*g(i3,i4,j2)*V(i4,p2,me)
```

```
        ;
#call squareamplitude(Amp,M)
.sort
Symbol s,t,u;
id  prop(q.q) = 1/s;
id  prop(qq.qq) = 1/t;
id  pe.pp = s/2-me^2;
id  p1.p2 = s/2-me^2;
id  pe.p1 = me^2-t/2;
id  pp.p2 = me^2-t/2;
id  pe.p2 = me^2-u/2;
id  pp.p1 = me^2-u/2;
id  u = 4*me^2-s-t;
Print +f;
.sort
S   s2,t2,si,si2,ti,ti2,me2,mm2,me4,mm4;
Format C;
Format 68;
```

```
id  s^2 = s2;
id  1/s^2 = si2;
id  1/s = si;
id  1/t^2 = ti2;
id  1/t = ti;
id  t^2 = t2;
id  me^2 = me2;
id  mm^2 = mm2;
id  me2^2 = me4;
id  mm2^2 = mm4;
Print +f;
.sort
#write <eeee.c> "#include \"ourvariables.h\"\n"
#write <eeee.c> "double eeee(double t) {"
#write <eeee.c> "   double t2 = t*t, ti = 1/t, ti2 = ti*ti;"
#write <eeee.c> "   double M;"
#write <eeee.c> "   M = %e",M
#write <eeee.c> "   return(M);"
```

```
#write <eeee.c> "}"
.end
```

and the output is

```
M =
   48 + 16*s^-2*t^2 + 32*s^-1*t + 32*s*t^-1 + 16*s^2*t^-2 - 64*
   me^2*s^-2*t - 64*me^2*s*t^-2 + 64*me^4*s^-2 - 64*me^4*s^-1*
   t^-1 + 64*me^4*t^-2;


Time =       0.00 sec    Generated terms =          10
               M         Terms in output =          10
                         Bytes used      =         396

M =
   48 + 64*ti2*me4 + 64*si2*me4 - 64*si*ti*me4 + 16*t2*si2 + 16*
   s2*ti2 - 64*t*si2*me2 + 32*t*si - 64*s*ti2*me2 + 32*s*ti;
```

The file eeee.c contains

```
#include "ourvariables.h"

double eeee(double t) {
   double t2 = t*t, ti = 1/t, ti2 = ti*ti;
   double M;
   M = 48 + 64*ti2*me4 + 64*si2*me4 - 64*si*ti*me4 + 16*t2*si2 + 16
      *s2*ti2 - 64*t*si2*me2 + 32*t*si - 64*s*ti2*me2 + 32*s*ti;

   return(M);
}
```

Calculating more complicated reactions is now mainly a matter of adapting the substitutions of the kinematic variables, putting some statements in the file ourvariables.h and maybe adding a bit to the #write instructions. If we make the interactions more complicated we may have to extend the squareamplitude procedure, for instance to add color treatment. Of course for the color traces we can use similar techniques as for the gamma matrices. It is just that we need an extra procedure to take the actual traces. Such procedures exist (see for instance the color package in the FORM site which has a rather efficient SU(N) procedure).

When we do a one loop calculation there are more steps involved. The main problem is then how to bring the result of the traces in such a form that one can insert the loop integrals. If things are sufficiently simple the loop integrals can be done analytically. Unfortunately this is rarely the case. Hence what one does is prepare the formulas for processing the integrals numerically. There exist programs that have the loop integrals programmed (like FF and LoopTools). Usually the so-called scalar loop integrals have been worked out carefully for the 2- 3- and 4-point functions. The major algebraic job is to express the amplitudes or matrix elements in terms of these functions. Again there are various algorithms for this. The best known method is called after a paper by Pasarino and Veltman, although the method can be traced back to Brown and Feynman and apparently even further to a thesis of a student at Cornell university in the early 50's.

All methods have in common that they involve much work and are very hard to properly automatize, although it has been done. For the average graduate student it may be a few years work. And it requires much testing.

Here I will show the method used in the GRACE system. Conceptually it is rather simple, but the tradeoff is that it is slow in execution. And of course somebody has been spending a lot of time making the corresponding numerical libraries.

One way to do loop integrals is by introducing Feynman parameters. If the integral looks originally like

$$\begin{aligned} I &= \int d^D Q \frac{P(Q)}{(Q^2 - m_1^2)((Q+r_1)^2 - m_2^2)\cdots((Q+r_{n-1})^2 - m_n^2)} \\ &= \int dx_1 \cdots dx_n \delta(1 - x_1 - \cdots - x_n) \int \frac{d^D Q \;\; P(Q)}{((Q + x_2 r_1 + \cdots + x_n r_{n-1})^2 - x_1 m_1 - x_n m_n)^n} \end{aligned}$$

Here $P(Q)$ is a formula that can contain dotproducts involving Q. We make the shift $Q \to K - x_2 r_1 - \cdots - x_n r_{n-1}$ which makes the momentum integral trivial, but we are left with Feynman parameters in the numerator. The first step in doing the integrals is to apply the delta function and replace (for historical reasons) $x_2 = 1 - x_1 - x_3 - \cdots - x_n$.

The various combinations of the Feynman parameters fulfil the same role as the formfactors in the Passarino-Veltman method. Actually inside the numerical libraries they are converted into these formfactors and then expressed in combinations of the scalar loop integrals. The reason this method was selected is because it is easiest to work out the coefficients of the various monomials in the Feynman parameters. Hence this is what we will do:

- We take a one-loop diagram, multiplied by the conjugate of a tree graph.
- We substitute the Feynman rules.
- We make the change of variables as explained above.
- We sort out what is D dependent and what can be done in 4 dimensions.
- We do the traces.
- We make substitutions of the dotproducts and Levi-Civita tensor contractions into a minimal set of variables.
- We write everything in terms of symbols.
- We write the output to a FORTRAN routine.

Here FORTRAN is the language of choice because the FF library is written in FORTRAN. Additionally the KEK group has convinced Fujitsu to write a FORTRAN compiler that can work at quadruple precision which is very handy when one has numerical problems and wants to check gauge cancellations.

Let us see what this all looks like. The grace system generates lots of FORM programs like the one below for each reaction. In the case of the reaction $e^-e^+ \to \mu^-\overline{\nu}_\mu u\bar{d}$ we get 6094 one loop graphs and 44 tree graphs in a most general non-linear gauge. Some of the one-loop graphs are counter terms. Let us take a bad one. This is for instance loop diagram 2689 times tree diagram 3. It is named b2689x3. The automatically generated FORM program looks like

```
#define name "b2689y"
#define NAME "b2689x3"
#define NAMEX "c2689x3"
#define nloop "4"
#define FINITE "1"
#define FLINE  "3"
#include grcform.hh

* Internal momenta

Vector q7;  * z
Vector l8,q8;   * w-plus
Vector k9,q9;   * w-plus
Vector l10,q10; * u
Vector l11,q11; * z
Vector q12; * w-plus
Vector q19; * z
Vector q20; * muon
```

```
Vector q21; * w-plus

* Momentum conservation at vertices

* k9 : x2
* l11 : x3
* l10 : x4
* l8 : x1

* k9 : loop momentum

#procedure momsub
Id q7 =-p1-p2;
Id q8 =+p1+p2;
Id q10 =+p3+p4+p5;
Id q11 =+p3+p4;
Id q12 =-p3-p4;
Id q19 =-p1-p2;
```

```
Id q20 =+p1+p2-p3;
Id q21 =-p5-p6;
#endprocedure

#procedure preint
Id l11 = + q11+k9;
Id l10 = + q10+k9;
Id l8 = + q8+k9;
.sort

* Feynman PARAMETER

Id x2 = 1-x3-x4-x1;
* Momentum shift
Id kl = kl*zk;
Id k9 = kl*zk-lshift;
Id zk^x?odd_ = 0;
Id lshift = +x3*q11+x4*q10+x1*q8;
```

```
#endprocedure

#procedure TreeDenom(fil)

* Denominator of tree propatagors

#write <‘fil’> "*"
#write <‘fil’> "      ztd = cc1"
#write <‘fil’> "      CALL snprpdn(pphase,ztd,vn7,amz**2,amz*agz)"
#write <‘fil’> "      CALL snprpdn(pphase,ztd,vn24,amw**2,amw*agw)"
#write <‘fil’> "      CALL snprpdc(pphase,ztd,vn7,amz**2,amz*agz)"
#write <‘fil’> "      CALL snprpdc(pphase,ztd,vn14,ammu**2,0.0d0)"
#write <‘fil’> "      CALL snprpdc(pphase,ztd,vn31,amw**2,amw*agw)"
#if ( ‘CCCC’ == 0 )
#write <‘fil’> "      ztd = ‘XCP’1/ztd"
#else
#write <‘fil’> "      ztd = cc1/ztd"
#endif
```

```
#write <‘fil’> "*"
#endprocedure
*
*   Amplitude
*
L   Sigma = + 3
    *ufp(fl0,p1,‘amel’)
    *ffvc(‘dzel1’,‘dzel2’,fl0,p1,p2,q19,n12c)
    *vfp(fl0,p2,‘amel’)
    *ffvn(‘czel1’,‘czel2’,fl0,p2,p1,q7,n6c)
    *ufp(fl1,p3,‘ammu’)
    *ffvn(‘cwnm1’,‘cwnm2’,fl1,-p3,-p4,-q12,n11c)
    *vfp(fl1,p4,‘amnm’)
    *ffvc(‘dwnm1’,‘dwnm2’,fl1,-p4,q20,q21,n14c)
    *sfc(fl1,-q20,‘ammu’)
    *ffvc(‘dzmu1’,‘dzmu2’,fl1,-q20,-p3,-q19,n13c)
    *ufp(fl2,p5,‘amuq’)
    *ffvn(‘czuq1’,‘czuq2’,fl2,-p5,l10,-l11,m10c)
```

```
      *sfn(fl2,l10,‘amuq’)
      *ffvn(‘cwdq1’,‘cwdq2’,fl2,-l10,-p6,l8,m8c)
      *vfp(fl2,p6,‘amdq’)
      *ffvc(‘dwdq1’,‘dwdq2’,fl2,-p6,-p5,-q21,n15c)
      *nlzwwn(‘czww’,-q7,n7a,-l8,m7b,k9,m7c)
      *nlzwwn(‘czww’,l11,m9a,-k9,m9b,q12,n9c)
      *dvn(n7a,n6c,q7,‘amz’)
      *dvn(m7b,m8c,l8,‘amw’)
      *dvn(m9b,m7c,k9,‘amw’)
      *dvn(m10c,m9a,l11,‘amz’)
      *dvn(n11c,n9c,q12,‘amw’)
      *dvc(n13c,n12c,q19,‘amz’)
      *dvc(n15c,n14c,q21,‘amw’);
#call grcform(1)
.end:‘NAME’;
```

I have taken out a few non-relevant commentary lines. As all computer generated code, it is not really for reading, but in this case it is not too bad either, and it even contains commentary.

All declarations are in the file grcform.hh and so are all relevant procedures and calls to other header files like the one that describes the full standard model in a general non-linear gauge. The call to the grcform procedure does all the work. The file has the extension .hh because there are already many .h files in the calculation for the FORTRAN routines. This way they could be kept separated.

At the moment we reach the writing routines this diagram contains 639727 terms with about 4.8×10^6 arithmetic operations and it definitely is not the worst. The file is anywhere from 25 to 30 Mbytes depending on whether we can use tabulator characters. Our task is now to do better than this, because if we have to do more than 200,000 diagrams the eventual Monte Carlo program becomes a bit unwieldy.

Problem: write lengthy outputs with as few arithmetic operations as possible. This is called code simplification. It is not a very advanced science. Compilers typically try to do this but they operate at a handicap: For a compiler $(a + b) + c$ is not the same as $a + (b + c)$ due to numerical precision problems. Also, using a high level of optimization with the compiler may need very much time, and on some of the bigger routines I had it even crash by lack of address space.

Let us take this formula with its 639727 terms and see how much we can squeeze it. We have put the formula in a file named testx3.sav and we can read it with the program

```
    Load testx3.sav;
 Sigma loaded
    Local test = Sigma;
    .sort

Time =       0.66 sec    Generated terms =     639727
            test         Terms in output =     639727
                         Bytes used      =   27495480
    Delete storage;
    .end

Time =       0.92 sec    Generated terms =     639727
            test         Terms in output =     639727
                         Bytes used      =   27495480
```

When we print the output it looks like

```
Delete storage;
Format O0;
Bracket x1,x3,x4,xlevi,zk;
Print +f;
.end
```

```
Time =       1.40 sec    Generated terms =     639727
            test         Terms in output =     639727
                         Bytes used      =   22747648
```

```
test=
   +xlevi*(-24*amel2*amdq2^2*es1235*xcp5-24*amel2*amdq2^2*es1235
      *xcp1+48*amel2*amdq2^2*es1234*xcp5+72*amel2*amdq2^2*es1234
      *xcp1+48*amel2*amuq2*amdq2*es1235*xcp5+48*amel2*amuq2*
      amdq2*es1235*xcp1-48*amel2*amuq2*amdq2*es1234*xcp6-48*
      amel2*amuq2*amdq2*es1234*xcp5-24*amel2*amuq2^2*es1235*xcp5
      -24*amel2*amuq2^2*es1235*xcp1+48*amel2*amuq2^2*es1234*xcp6
```

```
-72*amel2*amuq2^2*es1234*xcp1+96*amel2*ammu2*amdq2*es2345*
xcp7+192*amel2*ammu2*amdq2*es2345*xcp5+48*amel2*ammu2*

and more than 400000 extra lines.
```

The format statement is a basically a dummy statement because O0 is the default. The output is bracketted in the Feynman parameters, xlevi (which indicates whether there is a Levi-Civita tensor and hence the terms are imaginary), and the parameter zk which indicates whether there are powers of Q^2 in the answer. Each two powers of zk indicate one power of Q^2. The denominator with its $Q.Q$ and its Feynman parameters is an overal factor and taken for granted.

Now we switch to a higher format level. This may make the program crash, because it needs a rather big amount of workspace and it does not warn about that (still needs some attention). Hence we use a special file that defines a number of setup parameters to tune FORM to the computer at hand and to the problem:

```
LargeSize 1G
SmallSize 200M
ScratchSize 500M
TermsInSmall 1M
MaxTermSize 50K
WorkSpace 40M
```

The workspace of 40 Mbytes should be more than enough. We have now

```
    Delete storage;
    Format O1;
    Bracket x1,x3,x4,xlevi,zk;
    Print +f;
    .end

Time =       1.40 sec    Generated terms =     639727
            test         Terms in output =     639727
                         Bytes used      =   22747648
      Z1_=5*xnlb;
      Z2_=-6+Z1_;
      Z3_=xnlb*Z2_;
      Z4_=11+Z3_;
      Z5_=ammu2*Z4_;
      Z6_=3-xnlb;
      Z7_=xnlb*Z6_;
      Z8_=3*Z7_;
      Z9_=-8+Z8_;
```

```
Z10_=e3e1*Z9_;
    .
    .
    .   almost 350000 lines
    .
    .
Z50_=4*Z54_+Z50_;
Z50_=ammu2*Z50_;
Z30_=Z30_+Z37_+Z50_+Z48_;
Z30_=xcp5*Z30_;
Z2_=Z51_+Z44_+2*Z24_+2*Z43_+2*Z52_+Z47_+2*Z7_+4*Z49_+2*Z31_+2
*Z2_+4*Z14_+Z19_+2*Z8_+2*Z15_+Z28_+2*Z30_;
Z2_=24*Z2_;
test=xlevi*Z5_+x4*Z26_+x4*xlevi*Z226_+x4^2*Z167_+x4^2*xlevi*
Z53_+x4^3*Z166_+x4^3*xlevi*Z68_+x3*Z18_+x3*xlevi*Z35_+x3*x4*
Z82_+x3*x4*xlevi*Z126_+x3*x4^2*Z139_+x3*x4^2*xlevi*Z87_+x3^2*
Z96_+x3^2*xlevi*Z40_+x3^2*x4*Z25_+x3^2*x4*xlevi*Z23_+x3^3*
Z132_+x3^3*xlevi*Z111_+x1*Z9_+x1*xlevi*Z21_+x1*x4*Z69_+x1*x4*
```

```
xlevi*Z20_+x1*x4^2*Z57_+x1*x4^2*xlevi*Z39_+x1*x3*Z17_+x1*x3*
xlevi*Z11_+x1*x3*x4*Z29_+x1*x3*x4*xlevi*Z12_+x1*x3^2*Z41_+x1*
x3^2*xlevi*Z1_+x1^2*Z6_+x1^2*xlevi*Z10_+x1^2*x4*Z3_+x1^2*x4*
xlevi*Z34_+x1^2*x3*Z16_+x1^2*x3*xlevi*Z27_+x1^3*Z32_+x1^3*
xlevi*Z4_+zk^2*Z42_+zk^2*xlevi*Z13_+zk^2*x4*Z22_+zk^2*x4*
xlevi*Z36_+zk^2*x3*Z33_+zk^2*x3*xlevi*Z38_+zk^2*x1*Z45_+zk^2*
x1*xlevi*Z46_+Z2_;
```

```
20.09 sec out of 20.12 sec
```

If we have variables outside brackets, they are not part of the optimizations. This is needed like this if we need their coefficients. In that case we speak of 'simultaneous optimization'. The variables Z_ are temporary variables that have been introduced by FORM for this calculation. They are called 'extra symbols' and we do have control over their representation:

```
    ExtraSymbols,vector,w;
    Delete storage;
    Format O1,stats=on;
    Bracket x1,x3,x4,xlevi,zk;
    Print +f;
    .end

Time =       1.43 sec    Generated terms =     639727
            test         Terms in output =     639727
                         Bytes used      =   22747648
      w(1)=5*xnlb;
      w(2)=-6+w(1);
      w(3)=xnlb*w(2);
      w(4)=11+w(3);
      w(5)=ammu2*w(4);
      w(6)=3-xnlb;
      w(7)=xnlb*w(6);
          .
```

```
          .
          .
       w(16)+x1^2*x3*xlevi*w(27)+x1^3*w(32)+x1^3*xlevi*w(4)+zk^2*
       w(42)+zk^2*xlevi*w(13)+zk^2*x4*w(22)+zk^2*x4*xlevi*w(36)+zk^2
       *x3*w(33)+zk^2*x3*xlevi*w(38)+zk^2*x1*w(45)+zk^2*x1*xlevi*
       w(46)+w(2);
*** STATS: original  33660P 4130889M 639679A : 4837888
*** STATS: optimized 2P 281714M 252897A : 534615

  20.00 sec out of 20.02 sec
```

We have turned on the statistics of the optimization and we see that in the input there were 33660 exponent calls of at least a third power (a square counts as a multiplication), 4130889 multiplications and 639679 additions (subtractions counted as additions). In the output these numbers are 2, 281714 and 252897 respectively. This is an improvement of almost an order of magnitude. Actually the compiler does not do that well.

Let us have a look at this system of optimization. In FORM that is controled with the format statement. When setting the format the output will be in the indicated format until the format is changed. There are two output modes which are controled differently. Let us start with the easy mode: the print statement. We start with the O1 format:

```
    Symbols x,y,z;
    Off Statistics;
    Local F = 6*y*z^2+3*y^3-3*x*z^2+6*x*y*z-3*x^2*z+6*x^2*y;
    Format O1,stats=on;
    Print;
    .end
      Z1_=y*z;
      Z2_= - z + 2*y;
      Z2_=x*Z2_;
      Z3_=z^2;
      Z1_=Z2_ - Z3_ + 2*Z1_;
      Z1_=x*Z1_;
      Z2_=y^2;
      Z2_=2*Z3_ + Z2_;
      Z2_=y*Z2_;
      Z1_=Z2_ + Z1_;
      F=3*Z1_;
*** STATS: original  1P 16M 5A : 23
```

```
*** STATS: optimized 0P 10M 5A : 15
```

We see that FORM has generated a lot of very simple statements and because we also have specified that we want statistics from the optimization it tells us that originally there were 23 operations and now there are 15 operation. What does the program do?

When we have a polynomial in a single variable, one way to evaluate is with a Horner scheme:

$$a(x) = \sum_{i=0}^{n} a_i x^i = a_0 + x(a_1 + x(a_2 + x(\ldots + x \cdot a_n))).$$

For multivariate polynomials one can do something similar (assume that the a_i are polynomials in other variables. Unfortunately, with more than one variable the result is not unique. The number of operations may depend on the order in which we take the variables out. And when there are many variables it soon becomes impossible to try all orderings.

Traditionally one counts how many times each variable is used and orders them that way. We call this ordering 'occurrence'. This is not optimal.

In addition, after placing the brackets one can try other optimizations as looking for common subexpressions. This is what O1 does: it tries the canonical Horner scheme (and a second one in which we order exactly the opposite way) and looks for simple common subexpressions.

In O2 it tries for more complicated common subexpressions. This is called greedy optimization. This is a search that is quadratic in the sizes of the subexpressions and also has to traverse the system more than once. As a consequence it is much slower. For our toy sample that is irrelevant:

```
Symbols x,y,z;
Off Statistics;
Local F = 6*y*z^2+3*y^3-3*x*z^2+6*x*y*z-3*x^2*z+6*x^2*y;
Format O2,stats=on;
Print;
.end
  Z1_=z^2;
  Z2_=2*y;
  Z3_=z*Z2_;
  Z2_= - z + Z2_;
  Z2_=x*Z2_;
  Z2_=Z2_ - Z1_ + Z3_;
  Z2_=x*Z2_;
  Z3_=y^2;
```

```
      Z1_=2*Z1_ + Z3_;
      Z1_=y*Z1_;
      Z1_=Z1_ + Z2_;
      F=3*Z1_;
*** STATS: original  1P 16M 5A : 23
*** STATS: optimized 0P 9M 5A : 14
```

As you can see, thing have been arranged differently and the result is that there is one operation less.

When we apply this to our big formula the result is

```
ExtraSymbols,vector,w;
Delete storage;
Format O2,stats=on;
Bracket x1,x3,x4,xlevi,zk;
Print +f;
.end
```

```
Time =       1.40 sec    Generated terms =     639727
            test         Terms in output =     639727
                         Bytes used      =   22747648
      w(1)=2*xnlb;
      w(2)=w(1)+3;
      w(3)=2*amel2;
      w(4)=w(2)*w(3);
          .
          .
          .
```

```
      x1^2*w(21)+x1^2*xlevi*w(6)+x1^2*x4*w(7)+x1^2*x4*xlevi*w(25)+
      x1^2*x3*w(22)+x1^2*x3*xlevi*w(16)+x1^3*w(30)+x1^3*xlevi*w(23)
      +zk^2*w(33)+zk^2*xlevi*w(31)+zk^2*x4*w(41)+zk^2*x4*xlevi*
      w(45)+zk^2*x3*w(43)+zk^2*x3*xlevi*w(49)+zk^2*x1*w(26)+zk^2*x1
      *xlevi*w(50)+w(1);
*** STATS: original  33660P 4130889M 639679A : 4837888
*** STATS: optimized 2P 186598M 220298A : 406900

  3828.65 sec out of 3838.53 sec
```

and the number of operations has gone down from 534615 to 406900. But the execution time has shot up enormously. This may be worthwhile, but only if the function has to be evaluated very many times.

With O3 things become complicated. In this case we try to improve the order of the variables in the Horner scheme. There are N! orderings. When we first select the first variable, then the second etc. it is a bit like trying to find a path in a tree. In addition we can either start to select the outer most variable first (forward) or the innermost variable first (backward). The field of game theory has come up with a method to do this efficiently to find a good outcome. The method is called MCTS (Monte Carlo tree search). When we select a branch of the tree and that branch has never been visited before we give it a value by playing out that branch to the end randomly and looking at its 'value'. we assign that value to that branch. If we have visited the branch before we look at the values of its branches. There is a formula for which of these branches we should select which is called UCT (upper confidence level for trees):

$$UCT_i \;=\; \langle x_i \rangle + 2C_p \sqrt{\frac{2\log n}{n_i}},$$

and the child with the highest UCT value is selected. Here C_p is a constant that may be problem dependent. $\langle x_i \rangle$ is the average score of child i over the previous traversals, n_i is the number of times child i has been visited before, and n is the number of times the node itself has been visited.

Basically the first term uses knowledge that we have obtained in previous samplings of the tree (exploitation) and the second term goes into uncharted territory (exploration). C_p

determines the balance between the two. If there are various branches with the same value, a random number will determine which one we will take. This statistical procedure works amazingly well. But it also means that two runs with a different seed for the random number generator may obtain different results.

Let us see how that works for our toy sample:

```
Symbols x,y,z;
Off Statistics;
Local F = 6*y*z^2+3*y^3-3*x*z^2+6*x*y*z-3*x^2*z+6*x^2*y;
Format O3,stats=on;
Print;
.end
  Z1_=x + z;
  Z2_=2*y;
  Z3_=Z2_ - x;
  Z1_=z*Z3_*Z1_;
  Z3_=y^3;
  Z2_=x^2*Z2_;
  Z1_=Z1_ + Z3_ + Z2_;
  F=3*Z1_;
*** STATS: original  1P 16M 5A : 23
*** STATS: optimized 1P 6M 4A : 12
```

Because the default for O3 is to sample the tree up to 1000 times and because we have only three variables, it stops after 6 samplings because it has had all different trees. And indeed it found a better tree. In this case the statistical sampling did not do anything for us, but in the big formula the story is different:

```
#do i = 1,5
ExtraSymbols,vector,w;
#message mctsconstant = 0.‘i’
Format O3,mctsconstant=0.‘i’,hornerdirection=backward,
         method=csegreedy,mctsnumexpand=1000,stats=on;
#message CPU time till now: ‘time_’ sec.
Local test = Sigma;
B   x1,x3,x4,xlevi,zk;
.sort
#Optimize test
.store
#enddo
.end
```

The results are

```
*** STATS: optimized 2P 163477M 207304A : 370785
*** STATS: optimized 3P 175169M 209676A : 384851
*** STATS: optimized 2P 176962M 210156A : 387122
*** STATS: optimized 2P 172261M 203804A : 376069
*** STATS: optimized 3P 170929M 212534A : 383469
```

for C_p is 0.1, 0.2, 0.3, 0.4, and 0.5 respectively. The number of operations is bouncing around a bit and we do not see a pattern in which value of C_p is to be preferred. This also took much CPU time. The five determinations took a bit more than 9 hours on a slightly faster computer.

For a different and much smaller formula we have run a Monte Carlo program to see what happens (still more than 4000 lines of code):

```
Symbol amel2 zk xcp3 xcp1 x1 x5 x4 xcp2 x3
            e2e1 e3e2 e3e1 e4e2 e4e1 EFUN;
Off Statistics;
.global
#include- ReadSigma.h
.store
#setrandom 1021
#do i = 1,4000
  #redefine R "`random_(log,0.01,10.0)'"
  #message mctsconstant = `R'
  Format Optimize,mctsconstant=`R',
              hornerdirection=backward,
              method=cse,mctsnumexpand=3000,stats=on;
  L       FF = Sigma;
  .sort
  #Optimize FF
```

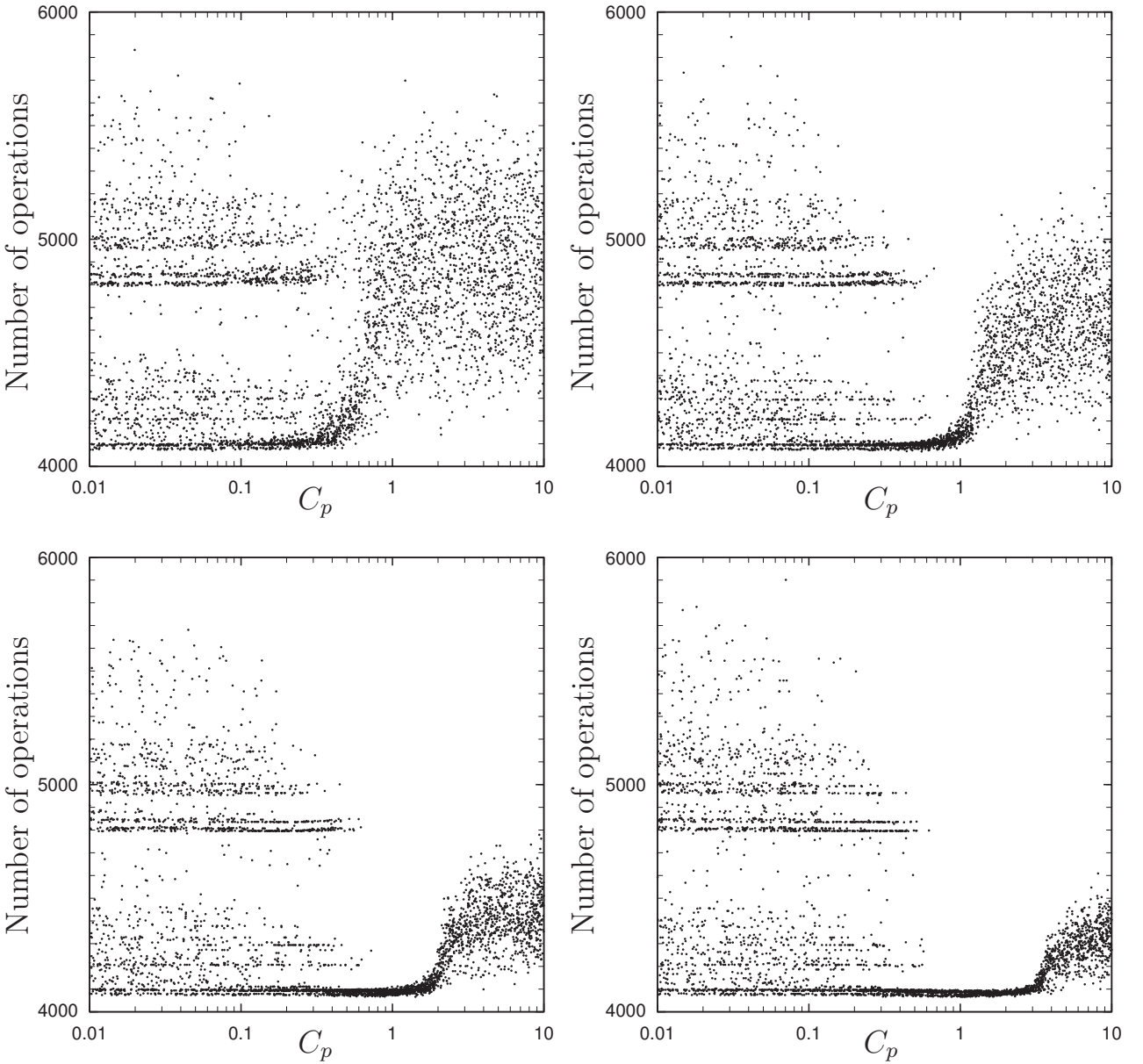
```
   .store
#enddo
.end
```

FORM has been equipped with a random number generator as mentioned before. In this case the preprocessor version of it can produce floating point numbers with a distribution. This floating point number is not really seen as a number. It is just seen as a string and is to be used as input for the format statement as a value for the mctsconstant C_p. Hence, in this program we run the optimization 4000 times, each with 3000 tree evaluations and for each of those 4000 times we determine the best outcome. We also see the second way to do the optimization. We have the #Optimize instruction. This one does the optimization and stores the result in a buffer. If needed we can then write the buffer contents later to a file. The expression is replaced by the final line of the optimized expression. If we optimize another expression after this the buffer is cleared and the previously optimized expression is removed. This is unlike the print statement that keeps the original of the expression and only prints the output, but does not keep it in a buffer.

The expression that we use here is of a similar type as what we used before. Such expressions give usually the best results when using the option backward for the searching through the Horner tree. To speed up the search we omit the greedy optimizations. It turns out that this does not alter the qualitative aspects of the result. Even so this is of course a rather big job. The output looks like

```
~~~mctsconstant = 1.482771
*** STATS: original  1270P 39168M 5716A : 47424
*** STATS: optimized 2P 1995M 2123A : 4122
~~~mctsconstant = 0.073660
*** STATS: original  1270P 39168M 5716A : 47424
*** STATS: optimized 2P 1916M 2287A : 4207
~~~mctsconstant = 0.135939
*** STATS: original  1270P 39168M 5716A : 47424
*** STATS: optimized 3P 1893M 2198A : 4097
                    .
                    .
```

and so on. We collect the optimized numbers in a scatterplot for 4 different values of the number of expansions: 300, 1000, 3000, 10000.


6000
5000
4000
Number of operations
0.01
0.1
1
10
C_p

We can see here that there is an optimal value for C_p and that if we know where this optimum is, we do not need excessive numbers of tree expansions.

The O2 and O3 options use algorithms that become relatively slow when the expressions are extremely big. It are however the extremely big expressions that need such optimizations most. Hence: can we do better?

The optimizations of FORM have no idea what the meaning of the formula is and what the variables mean. In a sense, FORM is just trying to create a bit of order in a chaos. If we know a bit about our problem, we may be able to do better. In AI terms this is called 'applying domain specific knowledge'. The best way to optimize an expression is to

1. Apply domain specific knowledge.

2. Use methods for chaos.

What can we do?

When we calculate reactions there are often combinations like $2p_1 \cdot p_2 + m^2$ that can make the expression shorter. This holds especially when applied selectively. If we apply it selectively however the number of variables will increase and that is not so good for the optimization system. Therefore we will try what happens when we make shifts in the variables.

Assume we have a set of variables xmin to xmax as in x4,...,x10. We want to see whether there are shifts of the type

```
id x5 = x5+a*x7+b;
```

that will make the expression shorter (a and b are numerical constants). We consider each time combinations of two variables and a potential constant. This means that when there are N variables the program will be quadratic in the number of variables.

The procedure is in the file linshift.prc. It is relatively lengthy, also because of the generous amount of commentary which we strip in the listings here. Because it cycles through the variables, it is possible that once it shifts variable x5, it should look at variable x4 again. Therefore we have to cycle through the variables and stop if a full cycle is unsuccessful. For this we use the variable $lasti.

The basic approach is that we select one variable, indicated by i. For this variable we generate for each other variable an expression in which we select the variables xi and xj renaming them into zlinshift1 and zlinshift2, shortly called z1 and z2 here. When $i = j$ we just select xi and look only for shifts of the type

```
id x5 = x5+b;
```

We have now the expressions Faaj with j running from some min to max.

We antibracket in z1 and z2 and collect the brackets in function f. Then comes the big trick.

```
#procedure linshift(F,f,x,minx,maxx,id,fname)
*
#$didshift = 0;
#$lasti = 0;
#do k = 1,1
 #do i = `minx',`maxx'
  .sort
  Format doublefortran;
  `OFFSTATS'
  On HighFirst;
  Hide `F';
  #do j = `minx',`maxx'
 #if ( `j' == `i' )
 L `F'aa`j' = `F'*replace_(`x'`i',z1);
 #else
 L `F'aa`j' = `F'*replace_(`x'`i',z1,`x'`j',z2);
```

```
#endif
 #enddo
 AB z1,z2;
 .sort
 Collect ‘f’;
 Normalize ‘f’;
 Dropcoefficient;
 DropSymbols;
 .sort
 $co = 0;
 $co1 = 0;
 Argument ‘f’;
if ( count(z1,1,z2,1) > $co ) $co = count_(z1,1,z2,1);
if ( count(z1,1) > $co1 ) $co1 = count_(z1,1);
 EndArgument;
 $co2 = coeff_;
 id  ‘f’(x?) = ‘f’(-$co2,x);
 Dropcoefficient;
```

```
if ( ( $co > 1 ) || ( $co1 == 0 ) ) id `f'(?a) = `f'`f'(?a);
id  `f'(x?,z1) = `f'`f'(x,z1);
id  `f'(x?,z2) = `f'`f'(x,z2);
id  `f'(x?,1)  = `f'`f'(x,1);
.sort
```

Once the dependence of z1 and z2 is inside f all we need to know is in how many terms the f with this argument occurs. Hence we drop the coefficient of all terms and we drop all symbols, making each term 1 times the function f. This means that after sorting the coefficient is the number of terms that were proportional to this combination of z1 and z2.

Next we only want linear shifts. Hence we select only functions f that have only terms linear in z1 and/or z2 or a constant. If a term does not satisfy this condition we rename the function into ff. Additionally we put the coefficient inside the function as its first argument, multiplied by -1, so that when we sort the expression the function with the largest occurrence comes first. We also rename the function if there is only a single term in the argument.

```
*
* Now we have the relevant functions in f and the irrelevant in ff.
* The first argument is minus the number of terms in the bracket.
*
  #$numskips = 0;
  #do j = `minx',`maxx'
 #$bra`j' = firstterm_(`F'aa`j');
 #$skip`j' = 1;
 #inside $bra`j'
 if ( count(`f',1) );
 $skip`j' = 0;
 SplitArg,(z1),`f';
 id `f'(x?,xb?$xb`j',z1) = 0;
 endif;
 #endinside
 #if ( `$skip`j'' )
 #$numskips = $numskips+1;
 Drop `F'aa`j';
```

```
#endif
 #enddo
 .sort
 #if ( `$numskips' < {`maxx'-`minx'+1} )
#do j = `minx',`maxx'
  #if ( `$skip`j'' == 0 )
if ( expression(`F'aa`j') );
id `f'(x?,xa?) = `f'(x,xa*replace_(z1,z1-($xb`j')));
id `f'`f'(x?,xa?) = `f'(x,xa*replace_(z1,z1-($xb`j')));
id `f'(x?,xa?) = -x*nterms_(xa);
endif;
  #endif
#enddo
.sort
```

After sorting we select the f of the first term and put it in $braj for expression j. Splitting the arguments so that z1 becomes a separate argument allows us to put the relevant part of the RHS of a substitution in the variable $xbj. $skipj is used for expressions Faaj that do not generate substitutions. If there are expressions remaining, in each of those expressions we make the substitution $z_1 \to z_1 - $xbj and we can calculate how many terms the new expression would have. This will be the new value of Faaj.

```
*
* Now find the minimum and look whether it is suitable.
*
 Drop;
 NDrop ‘F’;
 #$jmin = 0;
 #$nterms = termsin_(‘F’);
 #do j = ‘minx’,‘maxx’
   #if ( ‘$skip‘j’’ == 0 )
 #$t = ‘F’aa‘j’;
 #if ( ‘$t’ < ‘$nterms’ )
 #$jmin = ‘j’;
```

```
#$nterms = `$t';
#endif;
  #endif
#enddo
.sort
```

The minimum is suitable if it has fewer terms than the original expression F. The rest of the routine is to provide output about what we have done.

```
#if ( `$jmin' > 0 )
#inside $xb`$jmin'
id z2 = `x'`$jmin';
#endinside
#if ( `id' == 0 )
L `F'fortran = `$xb`$jmin'';
.sort
#endif
UnHide `F';
id `x'`i' = `x'`i' - (`$xb`$jmin'');
#write "      `x'`i' = `x'`i'+(`$xb`$jmin'');"
```

```
#write "* time = ‘time_’: new number of terms = ‘$nterms’"
#if ( ‘id’ > 0 )
#write <‘fname’> "id ‘x’‘i’ = ‘x’‘i’+(‘$xb‘$jmin’’);"
#else
#write <‘fname’> "      ‘x’‘i’ = ‘x’‘i’+(%E)",‘F’fortran
#endif
#redefine k "0"
#$lasti = ‘i’;
#$didshift = 1;
.sort
#else
#if ( ( ‘i’ == ‘$lasti’ ) && ( ‘k’ == 1 ) )
#breakdo 2
#endif
#endif
 #else
#if ( ( ‘i’ == ‘$lasti’ ) && ( ‘k’ == 1 ) )
#breakdo 2
```

```
 #endif
  #endif
 #enddo
#enddo
Unhide ‘F’;
Drop ‘F’fortran;
Format normal;
.sort
```

First we have to replace F by the new version. And we have to write somewhere what we have done. This can be in one of two ways:

- As a FORTRAN statement for a numerical program.
- As an id statement for when we want to reproduce this in FORM.

These transformations are put inside a file, because if we want to write a FORTRAN subroutine we first have to finish all declarations, including the statement in which we define the dimension of the extra symbols generated by the optimization (there is a preprocessor variable for that).

Finally we place a marker in $lasti to know for what value of i was the last substitution.

Let us see how this works. We make a small program that messes things up a bit with an id statement. Then we have linshift do its best.

```
    Symbols x1,...,x10;
    Symbols zlinshift1,zlinshift2,x,xlinshifta,xlinshiftb;
    CFunction f,ff;
    Local F = x6^4+x6^3*x7+x6^2*x7^2+x6*x7^3;
    id x6 = x6-x2+2*x3+x5-x4/2+2*x7;
    .sort

Time =       0.00 sec    Generated terms =         209
               F         Terms in output =         126
                         Bytes used      =        4816
    #call linshift(F,f,x,2,6,0,ex6file)

Time =       0.00 sec    Generated terms =         126
               F         Terms in output =         126
                         Bytes used      =        4816
      x2 = x2+(-2*x3);
* time = 0.00: new number of terms = 70
      x4 = x4+(2*x2);
```

```
* time = 0.00: new number of terms = 35
      x5 = x5+(-1/2*x4);
* time = 0.00: new number of terms = 15
      x6 = x6+(x5);
* time = 0.00: new number of terms = 5
    Print;
    .end

   F =
      x6^4 + 9*x6^3*x7 + 31*x6^2*x7^2 + 49*x6*x7^3 + 30*x7^4;

  0.01 sec out of 0.01 sec
```

Although it does not give the original expression back but a different, slightly longer expression, it is close to optimal. In the file ex6file we find

```
      x2 = x2+( - 2.D0*x3)
      x4 = x4+(2.D0*x2)
      x5 = x5+( - 1.D0/2.D0*x4)
      x6 = x6+(x5)
```

In the case of our complicated formula we need some more care. We have several groups of variables that cannot mix on dimensional grounds. Gauge parameters and Feynman parameters should not mix either. Neither should they mix with dimensionless coupling constants. Hence we split them in groups. This makes things much faster. But this way we need an extra routine to call the linshift procedure for the various groups.

```
#procedure doshifts(expr,name,feyn,kin,levi,gauge,coupl)
*
*   Procedure determines the order of the shifts to be tried
*   Variables:
*       x:   Feynman parameters.
*       yk:  regular kinematic variables (masses and dotproducts).
*       ye:  Levi-Civita tensors.
*       yg:  Gauge parameters.
*       XCP: Coupling constants.
*
#$lastshift = 0;
#$numpass = 1;
#$startterms = termsin_(`expr');
#write "Starting doshift with `$startterms' terms"
#do kdoshifts = 1,1
#write "doshift(`NAME') pass `$numpass' at time = `time_' sec"
#$numshift = 0;
*--------------------------x----------------------------
```

```
#if ( `$lastshift' == 1 )
    #breakdo
#endif;
#call linshift(`expr',fls,x,1,`feyn',1,`name'.fh)
#if ( `$didshift' > 0 )
    #$numshift = $numshift+$didshift;
    #$lastshift = 1;
#endif
*-------------------------yk-----------------------------
#if ( `$lastshift' == 2 )
    #breakdo
#endif;
#call linshift(`expr',fls,yk,1,`kin',0,`name'.ff)
#if ( `$didshift' > 0 )
    #$numshift = $numshift+$didshift;
    #$lastshift = 2;
#endif
*-------------------------ye-----------------------------
```

```
#if ( ‘$lastshift’ == 3 )
    #breakdo
#endif;
#call linshift(‘expr’,fls,ye,1,‘levi’,0,‘name’.ff)
#if ( ‘$didshift’ > 0 )
    #$numshift = $numshift+$didshift;
    #$lastshift = 3;
#endif
*-------------------------yg--------------------------
#if ( ‘$lastshift’ == 4 )
    #breakdo
#endif;
#call linshift(‘expr’,fls,yg,1,‘gauge’,0,‘name’.ff)
#if ( ‘$didshift’ > 0 )
    #$numshift = $numshift+$didshift;
    #$lastshift = 4;
#endif
*-------------------------xcp--------------------------
```

```
#if ( ‘$lastshift’ == 5 )
    #breakdo
#endif;
#call linshift(‘expr’,fls,xcp,1,‘coupl’,0,‘name’.ff)
#if ( ‘$didshift’ > 0 )
    #$numshift = $numshift+$didshift;
    #$lastshift = 5;
#endif
#if ( ‘$numshift’ > 0 )
    #redefine kdoshifts "0"
    #$numpass = $numpass+1;
#endif
#enddo
#write "doshift finished at time = ‘time_’ sec"
#endprocedure
```

Most of the code is calling the linshift routines for the various categories and setting cq checking the $lastshift variable so that we keep cycling till there are no more changes.

In the expression the coupling constants were all collected in each term and their product is represented by a single variable. There are usually not too many different variables inside a single diagram, although we have seen cases with 144 different combinations. Their names start with xcp.

The program we run for testing this is

```
#define NAME "test"
AutoDeclare Symbols x,y,z;
CFunction fls,flsfls;
Format nospaces;
Format 68;
Off statistics;
Load testx3.sav;
Global ‘NAME’ = Sigma;
Multiply replace_(e5e1,yk1,e5e2,yk2,e5e3,yk3,e4e1,yk4,e4e2,yk5,
                                e4e3,yk6,e3e1,yk7,e3e2,yk8,e2e1,yk9);
Multiply replace_(es1234,ye1,es1235,ye2,es1245,ye3,es1345,ye4,
                                es2345,ye5);
Multiply replace_(amel2,yk10,ammu2,yk11,amuq2,yk12,amdq2,yk13);
```

```
Multiply replace_(xnlb,yg1);
.sort
ExtraSymbols,vector,w;
Delete storage;
.sort
#call doshifts(‘NAME’,testout,4,13,5,1,8)
.store
```

We use the replace_ function to substitute all kinematic variables into the yk and ye variables. This is the fastest way to do this. The call to doshifts has the parameters testout as a basename for all files produced, and the number of the variables in each class. We ignore that x2 is missing. The output of the program till this point is

```
Starting doshift with 639727 terms
doshift(test) pass 1 at time = 3.24 sec
      x1 = x1+(x4-1);
* time = 14.17: new number of terms = 613303
      x3 = x3+(x4);
* time = 32.57: new number of terms = 575635
      x1 = x1+(-x4+1);
* time = 52.46: new number of terms = 521351
      yk1 = yk1+(yk2);
* time = 102.91: new number of terms = 510930
      yk2 = yk2+(-yk1);
* time = 121.44: new number of terms = 509714
      yk4 = yk4+(yk5);
* time = 157.51: new number of terms = 478955
      yk5 = yk5+(-yk4);
* time = 173.77: new number of terms = 472827
      yk6 = yk6+(1/2*yk11);
* time = 190.47: new number of terms = 456715
```

```
      yk7 = yk7+(yk8);
* time = 208.02: new number of terms = 436623
      yk8 = yk8+(-yk7);
* time = 224.09: new number of terms = 419341
      yk9 = yk9+(yk10);
* time = 240.12: new number of terms = 310354
      yk13 = yk13+(-yk12);
* time = 282.00: new number of terms = 280356
      yk1 = yk1+(-yk9);
* time = 292.44: new number of terms = 271808
      yk3 = yk3+(-yk1);
* time = 311.43: new number of terms = 269988
      yk4 = yk4+(yk7);
* time = 321.18: new number of terms = 248060
      yk5 = yk5+(yk8);
* time = 329.57: new number of terms = 238818
      yk6 = yk6+(-yk4);
* time = 338.18: new number of terms = 234193
```

```
      yk13 = yk13+(-2*yk6);
* time = 393.03: new number of terms = 225054
      yk6 = yk6+(yk4);
* time = 438.26: new number of terms = 217480
      yk13 = yk13+(2*yk1);
* time = 488.10: new number of terms = 212990
      yk1 = yk1+(yk3);
* time = 495.85: new number of terms = 212766
      yk3 = yk3+(-yk9);
* time = 510.14: new number of terms = 188704
      yk1 = yk1+(-yk3);
* time = 577.70: new number of terms = 187297
      yk3 = yk3+(yk1);
* time = 590.33: new number of terms = 167771
      ye3 = ye3+(ye2);
* time = 666.32: new number of terms = 164617
      ye4 = ye4+(ye5);
* time = 668.13: new number of terms = 162230
```

```
      ye5 = ye5+(-ye4);
* time = 669.89: new number of terms = 158089
      yg1 = yg1+(-1);
* time = 678.37: new number of terms = 147146
      xcp1 = xcp1+(xcp5);
* time = 681.74: new number of terms = 141857
      xcp2 = xcp2+(xcp6);
* time = 684.17: new number of terms = 140127
      xcp3 = xcp3+(-1/2*xcp1);
* time = 686.57: new number of terms = 138841
      xcp4 = xcp4+(-1/2*xcp2);
* time = 688.87: new number of terms = 138748
      xcp5 = xcp5+(-xcp1);
* time = 691.34: new number of terms = 133886
      xcp6 = xcp6+(-xcp2);
* time = 693.69: new number of terms = 133418
      xcp7 = xcp7+(xcp3);
* time = 696.02: new number of terms = 130440
```

```
      xcp8 = xcp8+(xcp4);
* time = 698.27: new number of terms = 129879
      xcp6 = xcp6+(-xcp5);
* time = 711.00: new number of terms = 129822
      xcp7 = xcp7+(-1/2*xcp1);
* time = 713.14: new number of terms = 127720
      xcp8 = xcp8+(-1/2*xcp2);
* time = 715.22: new number of terms = 127396
doshift(test) pass 2 at time = 731.48 sec
doshift finished at time = 798.36 sec
```

We see the number of terms going down steadily and in the end there is about 20% left of what we started with. That we also shift the Feynman parameters is for the optimization procedure in which we have those bracketted out. After optimization we will undo those shifts and for that we have a file with id statements. The only thing is that the id statements have to be reversed in order and for that we have written a separate C program that can be executed from FORM with the #system instruction (as with the program in session 3).

After this we can see what happens next in the optimizations.

```
.store
Format O1,stats=on;
Local test1 = test;
Bracket x1,x3,x4,xlevi,zk;
.sort
#optimize test1
.store
Format O2,stats=on;
Local test2 = test;
Bracket x1,x3,x4,xlevi,zk;
.sort
#optimize test2
.store
Format O3,hornerdirection=backward,mctsconstant=0.05,
          mctsnumexpand=1000,debugflag=1,stats=on;
Local test3 = test;
Bracket x1,x3,x4,xlevi,zk;
```

```
.sort
#optimize test3
Format O0;
#write <testout.fcp> "L SigmaC = %e",test3
Format O3,hornerdirection=backward,mctsconstant=0.05,
          mctsnumexpand=1000,debugflag=1,stats=on;
#write <testout.fcp> "%O"
.store
Print +f;
.end
```

We see here the sequel of our test program in which we do the optimizations

```
Time =     792.66 sec    Generated terms =     127396
           test1         Terms in output =     127396
                         Bytes used      =    6926180
    #optimize test1

Time =     796.49 sec    Generated terms =         48
           test1         Terms in output =         48
```

```
                          Bytes used      =       2324
*** STATS: original  1089P 810229M 127348A : 939755
*** STATS: optimized 2P 89186M 60724A : 149914
    .sort

Time =     796.50 sec    Generated terms =         48
           test1         Terms in output =         48
                         Bytes used      =       2240
    #clearoptimize
    .store
    Format O2,stats=on;
    Local test2 = test;
    Bracket x1,x3,x4,xlevi,zk;
    .sort

Time =     796.64 sec    Generated terms =     127396
           test2         Terms in output =     127396
                         Bytes used      =    6926180
```

```
    #optimize test2

Time =     987.25 sec    Generated terms =          48
           test2         Terms in output =          48
                         Bytes used      =        2324
*** STATS: original  1089P 810229M 127348A : 939755
*** STATS: optimized 2P 54032M 55813A : 109849
    .sort

Time =     987.26 sec    Generated terms =          48
           test2         Terms in output =          48
                         Bytes used      =        2240
    #clearoptimize
    .store
    Format O3,hornerdirection=backward,mctsconstant=0.05,
              mctsnumexpand=1000,debugflag=1,stats=on;
    Local test3 = test;
    Bracket x1,x3,x4,xlevi,zk;
```

```
    .sort

Time =     987.40 sec    Generated terms =     127396
           test3         Terms in output =     127396
                         Bytes used      =    6926180
    #optimize test3

Time =    2913.43 sec    Generated terms =         48
           test3         Terms in output =         48
                         Bytes used      =       2324
*** STATS: original  1089P 810229M 127348A : 939755
*** STATS: optimized 2P 47642M 55572A : 103218
    Format O0;
    #write <testout.fcp> "L SigmaC = %e",test3
    Format O3,hornerdirection=backward,mctsconstant=0.05,
              mctsnumexpand=1000,debugflag=1,stats=on;
    #write <testout.fcp> "%O"
    .end
```

```
Time =    2913.57 sec    Generated terms =          48
           test3         Terms in output =          48
                         Bytes used      =        2240
  2913.57 sec out of 2918.72 sec
```

and we see that the number of operations has decreased dramatically, as well as the time for the optimization. In this case the time for O2 is quite acceptable and O3 is only interesting for very many function evaluations. On the whole we have reduced the number of operations by about a factor 40. And of course we will gain time back by having much shorter compilation times.

The debugflag option in the format statements dumps the output in reverse order with id in front and a ; at the end. This can be used to substitute everything back and test whether we still have the same formula. The contents of the various files are:

testout.fh:

```
id x1 = x1+(x4-1);
id x3 = x3+(x4);
id x1 = x1+(-x4+1);
```

These statements should be reversed in order and used on the final expression to obtain the proper brackets for the Feynman parameters.

testout.ff

```
yk1 = yk1+(yk2)
yk2 = yk2+(-yk1)
yk4 = yk4+(yk5)
yk5 = yk5+(-yk4)
yk6 = yk6+(1.D0/2.D0*yk11)
yk7 = yk7+(yk8)
yk8 = yk8+(-yk7)
yk9 = yk9+(yk10)
yk13 = yk13+(-yk12)
yk1 = yk1+(-yk9)
yk3 = yk3+(-yk1)
yk4 = yk4+(yk7)
yk5 = yk5+(yk8)
yk6 = yk6+(-yk4)
yk13 = yk13+(-2.D0*yk6)
yk6 = yk6+(yk4)
yk13 = yk13+(2.D0*yk1)
```

```
yk1 = yk1+(yk3)
yk3 = yk3+(-yk9)
yk1 = yk1+(-yk3)
yk3 = yk3+(yk1)
ye3 = ye3+(ye2)
ye4 = ye4+(ye5)
ye5 = ye5+(-ye4)
yg1 = yg1+(-1.D0)
xcp1 = xcp1+(xcp5)
xcp2 = xcp2+(xcp6)
xcp3 = xcp3+(-1.D0/2.D0*xcp1)
xcp4 = xcp4+(-1.D0/2.D0*xcp2)
xcp5 = xcp5+(-xcp1)
xcp6 = xcp6+(-xcp2)
xcp7 = xcp7+(xcp3)
xcp8 = xcp8+(xcp4)
xcp6 = xcp6+(-xcp5)
xcp7 = xcp7+(-1.D0/2.D0*xcp1)
```

```
xcp8 = xcp8+(-1.D0/2.D0*xcp2)
```

These are statements for the FORTRAN subroutine to obtain the numerical value of the eventual variables.

testout.fcp

```
L SigmaC = w(1)+xlevi*w(5)+x4*w(92)+x4*xlevi*w(93)+x4^2*w(134)+x4^2*
      xlevi*w(226)+x4^3*w(1456)+x4^3*xlevi*w(679)+x3*w(190)+x3*xlevi*
      w(8)+x3*x4*w(237)+x3*x4*xlevi*w(23)+x3*x4^2*w(345)+x3*x4^2*xlevi*
      w(787)+x3^2*w(507)+x3^2*xlevi*w(97)+x3^2*x4*w(102)+x3^2*x4*xlevi*
      w(51)+x3^3*w(647)+x3^3*xlevi*w(45)+x1*w(76)+x1*xlevi*w(24)+x1*x4*
      w(66)+x1*x4*xlevi*w(28)+x1*x4^2*w(47)+x1*x4^2*xlevi*w(86)+x1*x3*
      w(35)+x1*x3*xlevi*w(11)+x1*x3*x4*w(39)+x1*x3*x4*xlevi*w(169)+x1*
      x3^2*w(85)+x1*x3^2*xlevi*w(53)+x1^2*w(2)+x1^2*xlevi*w(34)+x1^2*x4
      *w(41)+x1^2*x4*xlevi*w(21)+x1^2*x3*w(54)+x1^2*x3*xlevi*w(13)+x1^3
      *w(32)+x1^3*xlevi*w(62)+zk^2*w(33)+zk^2*xlevi*w(6)+zk^2*x4*w(70)+
      zk^2*x4*xlevi*w(37)+zk^2*x3*w(15)+zk^2*x3*xlevi*w(19)+zk^2*x1*
      w(26)+zk^2*x1*xlevi*w(18);

id w(1)=24*w(1);
id w(1)=w(17)+w(3)+4*w(4)+w(1);
id w(4)=w(7)+w(9)+w(4);
id w(9)=w(9)*w(2108);
```

```
id w(9)=w(10)+w(9);
          .
          .
          .
id w(6)=w(2)+9;
id w(5)=w(3)*w(4);
id w(4)=2*yk5;
id w(3)=w(2)+6;
id w(2)=w(1)*yg1;
id w(1)=yg1+6;
```

This should expand to the expression before we optimized it. By lack of time we will not show that here but the GRACE implementation does indeed do this. This was needed because we used this while the optimization code was still in its test phase and 100000+ expressions to be optimized give of course a good testing ground. It did indeed help us to find two nasty bugs. Since then this code has not revealed any new bugs.

One extreme example was diagram b2689x7. Before the shifting there were 111582 terms. After the shifting there were only 938 terms remaining and after an O2 optimization we were down to no more than 1284 operations (down from 8256) plus what the reexpansion of the Feynman parameters gives.