FOUR

Depending on the CA system non-commutative algebra is either trivial or complicated. In FORM it is rather natural. General functions are considered non-commutative. This is demonstrated in the following little example:

```
    Functions A,B;
    CFunctions f,g;
    Local F1 = (A+B)^3;
    Local F2 = (f+g)^3;
    Print;
    .end

Time =       0.00 sec    Generated terms =          8
              F1         Terms in output =          8
                         Bytes used      =        324
```

```
Time =        0.00 sec    Generated terms =           4
              F2          Terms in output =           4
                          Bytes used      =         184

   F1 =
      A*A*A + A*A*B + A*B*A + A*B*B + B*A*A + B*A*B + B*B*A + B*B*B;

   F2 =
      f^3 + 3*f^2*g + 3*f*g^2 + g^3;
```

As you can see here, in the commutative case the working out of the power can be done directly with binomial coefficients, while in the non-commutative case the powers just have to be multiplied out.

Also operations on non-commuting objects are rather natural:

```
Functions P,X;
Local F = (P+X)^5;
repeat id P*X = 1+X*P;
Print;
.end
```

```
Time =       0.00 sec    Generated terms =         126
               F         Terms in output =          12
                         Bytes used      =         476

   F =
      15*P + 10*P*P*P + P*P*P*P*P + 15*X + 30*X*P*P + 5*X*P*P*P*P
       + 30*X*X*P + 10*X*X*P*P*P + 10*X*X*X + 10*X*X*X*P*P
       + 5*X*X*X*X*P + X*X*X*X*X;
```

We can use this to program all kinds of complicated algebras.

During this lecture we will study the Campbell Baker Hausdorff (CBH) relation. This relation should be familiar to all physicists and mathematicians. We want to solve for F in the relation

$$e^F \;=\; e^A\, e^B \tag{1}$$

in which A and B are non commutative objects, usually belonging to an algebra. We can also write it differently:

$$F \;=\; \ln(e^A\, e^B) \tag{2}$$

and the familiar result is

$$F \;=\; A + B + \frac{1}{2}[A,B] + \frac{1}{12}([A,[A,B]] - [B,[A,B]]) + \cdots \tag{3}$$

The series continues with more and more complicated commutators. Just imagine we need many more terms in this expansion. There exists a formula by Dynkin which gives nested commutators. The coefficients are multiple nested sums, many of which just add up to zero and it is complicated to work them out. Can we do better?

The problem can be split into 4 parts:

1. Determine F in terms of A and B.
2. Express F in terms of commutators.
3. Try to bring F to a minimal form in a "hand guided" way.
4. Try to do the last step automatically.

Considering that we would like the program to go as deep as possible, speed is rather important. It is also important to select a good notation for the commutators. We will define:

$$C = [A, B] \tag{4}$$
$$C(D) = [D, C] \tag{5}$$
$$C(D, X) = [D, C(X)] \tag{6}$$

in which X is any set of arguments and D any single argument.

The final answer we want to obtain is a collection of terms, each with a coefficient and a single occurrence of C with a number of arguments A and/or B. Step three is based on the fact that the collection of all those terms is not linearly independent. As an example we give for now

$$C(B, A) \;=\; C(A, B) \tag{7}$$

but there are many more of such relations. It is a simple exercise to prove the above one.

Let us now turn to step one. The straightforward, but rather brutish version would look like

```
#define MAX "4"
Functions A,B,C,D;
Symbols x,j;
Format 68;
Off Statistics;
Local FA = sum_(j,0,‘MAX’,A^j/fac_(j));
Local FB = sum_(j,0,‘MAX’,B^j/fac_(j));
.sort
Local FC = FA*FB-1;
if ( count(A,1,B,1) > ‘MAX’ ) Discard;
.sort
Local F = -sum_(j,1,‘MAX’,sign_(j)*D^j/j);
id  D = FC;
if ( count(A,1,B,1) > ‘MAX’ ) Discard;
Print +f F;
.end
```

```
F =
    A + 1/12*A*A*B + 1/24*A*A*B*B + 1/2*A*B - 1/6*A*B*A - 1/12*A*
    B*A*B + 1/12*A*B*B + B - 1/2*B*A + 1/12*B*A*A - 1/6*B*A*B + 1/
    12*B*A*B*A + 1/12*B*B*A - 1/24*B*B*A*A;
```

This program takes (on my laptop) 0.03 sec. If however I want to go to 5 powers it takes already 3.59 sec and for MAX being 6 the time is 573 sec. We notice that in the multiplications most terms that are generated have a power that is too high. Can we do away with those?

This is what the symbol x is for:

```
#define MAX "5"
Functions A,B,C,D;
Symbols x(:‘MAX’),i,j;
Local FA = sum_(j,0,‘MAX’,x^j*A^j/fac_(j));
Local FB = sum_(j,0,‘MAX’,x^j*B^j/fac_(j));
.sort
Local FC = FA*FB-1;
Bracket x;
.sort
Local F = -sum_(j,1,‘MAX’,sign_(j)*D^j/j);
repeat;
    id,once,D*x^i? = sum_(j,1,‘MAX’-i,x^j*FC[x^j])*x^i;
endrepeat;
Bracket x;
Print +f F;
.end
```

```
F =
    + x * ( A + B )

    + x^2 * ( 1/2*A*B - 1/2*B*A )

    + x^3 * ( 1/12*A*A*B - 1/6*A*B*A + 1/12*A*B*B + 1/12*B*A*A
       - 1/6*B*A*B + 1/12*B*B*A )

    + x^4 * ( 1/24*A*A*B*B - 1/12*A*B*A*B + 1/12*B*A*B*A - 1/24*
      B*B*A*A )

    + x^5 * (  - 1/720*A*A*A*A*B + 1/180*A*A*A*B*A + 1/180*A*A*A
      *B*B - 1/120*A*A*B*A*A - 1/120*A*A*B*A*B - 1/120*A*A*B*B*A
       + 1/180*A*A*B*B*B + 1/180*A*B*A*A*A - 1/120*A*B*A*A*B + 1/
      30*A*B*A*B*A - 1/120*A*B*A*B*B - 1/120*A*B*B*A*A - 1/120*A
      *B*B*A*B + 1/180*A*B*B*B*A - 1/720*A*B*B*B*B - 1/720*B*A*A
      *A*A + 1/180*B*A*A*A*B - 1/120*B*A*A*B*A - 1/120*B*A*A*B*B
       - 1/120*B*A*B*A*A + 1/30*B*A*B*A*B - 1/120*B*A*B*B*A + 1/
```

```
180*B*A*B*B*B + 1/180*B*B*A*A*A - 1/120*B*B*A*A*B - 1/120*
B*B*A*B*A - 1/120*B*B*A*B*B + 1/180*B*B*B*A*A + 1/180*B*B*
B*A*B - 1/720*B*B*B*B*A );
```

First we see a few new features. There is the bracket statement. This tells the system to place brackets in the output and because we specify

```
Bracket x;
```

powers of the variable x will be outside the brackets. The rest will be inside. One can mention any number of variables in the bracket statement. The main restriction is that they can only be functions, symbols, vectors or dotproducts. If you mention a vector, all dotproducts of that vector will be taken outside the brackets. Sorting will be according to the brackets and secondary according to the content of the brackets. The bracket statement has a counterpart: the antibracket statement. This statement is like the bracket statement except for that it specifies what should be inside the brackets. The whole rest will be outside. Of course the coefficient of a term is always inside the bracket. The sorting according to the brackets is only for the module in which the (anti)bracket statement is specified. In the next module this information is forgotten.

The other new feature is the use of the expression FC. If an expression like FC has been bracketted, in the next module `FC[x^2]` refers to the contents of the bracket that has exactly x^2 outside the brackets. The bracket that has nothing outside is referred to as `FC[1]`.

```
 Symbols a,b,c;
 Format nospaces;
 Local F = (a+b+c)^3;
 Bracket a;
 Print;
 .sort

F=
   +a*(3*c^2+6*b*c+3*b^2)
   +a^2*(3*c+3*b)
   +a^3*(1)
   +c^3+3*b*c^2+3*b^2*c+b^3;
 Local G = F[a];
 Print G;
 .end

G=
   3*c^2+6*b*c+3*b^2;
```

Back to our output. The program has an automatic power cutoff in x when computing FC. When we compute F we make sure that we never even consider terms that will have too high a power in x. In addition the brackets make that the output is nicely ordered. The program takes now less than 0.01 sec for up to 5 powers and 3.99 sec when we go to 10 powers. But it can still be better. The problem is with the loop below:

```
repeat;
    id,once,D*x^i? = sum_(j,1,'MAX'-i,x^j*FC[x^j])*x^i;
endrepeat;
```

```
Time =       3.32 sec    Generated terms =     183711
               F         Terms in output =       1100
                         Bytes used      =      57476
```

It generates all terms at once and only then sorts them. This means that many terms are generated more than once and before they are added they are used again for generating more terms. In the previous session we have seen how to deal with this:

```
#define MAX "10"
Functions A,B,C,D;
```

```
Symbols x(:‘MAX’),i,j;
*Off Statistics;
Local FA = sum_(j,0,‘MAX’,x^j*A^j/fac_(j));
Local FB = sum_(j,0,‘MAX’,x^j*B^j/fac_(j));
.sort
Local FC = FA*FB-1;
Bracket x;
.sort
Hide;
Local F = -sum_(j,1,‘MAX’,sign_(j)*D^j/j);
#do i = 1,1
    id,once,D*x^i? = sum_(j,1,‘MAX’-i,x^j*FC[x^j])*x^i;
    if ( count(D,1,x,1) > ‘MAX’ ) Discard;
    if ( count(D,1) ) Redefine i "0";
    .sort
#enddo
Bracket x;
Print +f F;
```

```
.end
```

Again we see some new features: The hide statement and the discard statement. The discard statement is easiest. It just means: throw the current term away. For the hide statement we need to know a bit more.

Under normal circumstances FORM uses two scratch files. Actually these files are only made if their contents exceed a certain size which can be set in the setup (either at the start of the program or in a special file called the setup file). This parameter has the name scratchsize. Smaller contents are kept in memory. One scratch file is called the input file and the other the output file. For each module FORM reads the terms of an expression from the input file, processes them, then, when all terms of that expression have been processed, it sorts the results and writes the sorted results to the output file. This way it processes all expressions that are active. In the end the input file is emptied and the input and output files are exchanged.

There are moments that we have one of more expressions that we do not want to take part in this process. With the skip statement we can tell the system to copy them directly from the input to the output (preserving the bracket information). This is nice if we need to do this for just a single module, but if we need this for a large part of the program this is annoying as well. For this we have a third scratch file: the hide file. We put the expression, with its

bracket information, in the hide file and this way we can refer to it as if it is still in the input file, but it will not take part in the processing in the modules. If we want to activate it again, we use the unhide statement.

The above has one weakness. If we have a number of expressions in the hide file and we remove one of them, this will leave 'holes' in that file. FORM will not attempt to reuse that space. The result is that sloppy use of the hide system will lead to an ever growing hide file. If this becomes problematic, it is best to empty the hide file by unhiding all expressions and in the next module hiding them again. When the hide file has no expressions in it at the end of the module, FORM will remove it and the next hide statement will then start with a file of zero length.

Hence, by hiding the expression FC, we preserve its brackets and do not have to carry it around all the time. Then we do one power of D at a time and sort afterwards. This minimizes the amount of double work as we see in the statistics:

```
Time =       0.01 sec    Generated terms =       2507
               F         Terms in output =       1542
                         Bytes used      =      74036

Time =       0.04 sec    Generated terms =       7728
               F         Terms in output =       3224
                         Bytes used      =     154796

Time =       0.07 sec    Generated terms =      11315
               F         Terms in output =       3871
                         Bytes used      =     185996

Time =       0.11 sec    Generated terms =      11552
               F         Terms in output =       3851
```

```
                          Bytes used      =     185060

Time =       0.14 sec     Generated terms =      10411
               F          Terms in output =       3656
                          Bytes used      =     176132

Time =       0.17 sec     Generated terms =       8776
               F          Terms in output =       3398
                          Bytes used      =     163772

Time =       0.19 sec     Generated terms =       6854
               F          Terms in output =       2882
                          Bytes used      =     140588

Time =       0.20 sec     Generated terms =       4674
               F          Terms in output =       2248
                          Bytes used      =     111620
```

```
Time =       0.21 sec    Generated terms =       2760
               F         Terms in output =       1100
                         Bytes used      =      55004
```

The result is that the whole program takes now 0.23 sec. For 12 it takes 1.34 sec. and for 14 it needs 7.25 sec. For 18 it needs 194 sec. This is more or less acceptable, specially if one considers that the next steps may be more expensive and that we had to manipulate many millions of terms.

It is possible to make the program a little bit faster yet. When we specify

```
id,once,D*x^i? = sum_(j,1,'MAX'-i,x^j*FC[x^j])*x^i;
if ( count(D,1,x,1) > 'MAX' ) Discard;
if ( count(D,1) ) Redefine i "0";
```

we first substitute `FC[x^j]` and then normalize the resulting terms to throw away the excess powers. If we change this order by defining an extra function FCC and putting

```
id,once,D*x^i? = sum_(j,1,'MAX'-i,x^j*FCC(j))*x^i;
if ( count(D,1,x,1) > 'MAX' ) Discard;
if ( count(D,1) ) Redefine i "0";
id FCC(j?) = FC[x^j];
```

we first resolve the cutoff and then expand the number of terms by substituting the contents of the proper bracket. The execution times become now 0.22 sec, 1.23 sec, 6.60 sec and 176 sec respectively.

For the next step we modify the program slightly. First we introduce the concept of a global expression. Untill now we have seen only local expressions. The difference between a local and a global expression is that when we encounter .store as the end of a module, the module is executed, expressions are printed, but after that all local expressions are deleted, while the global expressions are stored in the store file. Also all declarations till then are forgotten, unless they were made global with a .global instruction at the beginning of the program. In that case these declarations (and other settings) are kept over the .store.

Stored expressions are not active. This means they will not be operated upon. They can be used in the RHS of the definition of other expressions or of id statements. And they can be copied to external files for use in other programs. As a consequence we can keep the results of our program in some files and use them as input for a program that does the next step in our project.

The new program looks like

```
Functions A,B,C,D,FCC;
Symbols i,j;
.global
#do MAX = 6,20
  Symbols x(:‘MAX’);
  Off Statistics;
  Local FA = sum_(j,0,‘MAX’,x^j*A^j/fac_(j));
  Local FB = sum_(j,0,‘MAX’,x^j*B^j/fac_(j));
  .sort
  Local FC = FA*FB-1;
  Bracket x;
  .sort
  Hide;
  Global F = -sum_(j,1,‘MAX’,sign_(j)*D^j/j);
  #do i = 1,1
    id,once,D*x^i? = sum_(j,1,‘MAX’-i,x^j*FCC(j))*x^i;
    if ( count(D,1,x,1) > ‘MAX’ ) Discard;
```

```
      if ( count(D,1) ) Redefine i "0";
      id FCC(j?) = FC[x^j];
      .sort
    #enddo
    Bracket x;
    Print +f F;
    .store
    #write "Time at the end of MAX = `MAX' is `time_' sec"
    Save expr`MAX'.sav;
    Delete Storage;
    .sort
    Drop;
    .sort
  #enddo
  .end
```

We have suppressed the printing of the output. This is not really necessary as we can pick up the expressions at any time. You are strongly advised to give the saved file the extension .sav in order to have a fixed convention for such files.

When a stored expression is being used, it may contain variables that have been erased from the variable administration. Or the expression may come from a previous program and those variables never existed in the current program. To avoid problems all variables that are used in a stored expression are specified in the storage file and when the expression is used these variables will be declared automatically when necessary. If there is a conflict, FORM will tell so and execution will stop. During execution, when the expression is used, the variables in the expression may have to be renumbered. They have some internal notation, and the internal notation of the variables in the creating program, and hence the file, may be different from the notation in the program that uses them. The result is that the use of stored expressions takes more internal resources than the use of hidden expressions for which such renumbering is not needed.

The output of our program is now:

```
Time at the end of MAX = 6 is 0.00 sec
Time at the end of MAX = 7 is 0.02 sec
Time at the end of MAX = 8 is 0.05 sec
Time at the end of MAX = 9 is 0.14 sec
Time at the end of MAX = 10 is 0.36 sec
Time at the end of MAX = 11 is 0.88 sec
Time at the end of MAX = 12 is 2.12 sec
Time at the end of MAX = 13 is 5.02 sec
Time at the end of MAX = 14 is 11.62 sec
Time at the end of MAX = 15 is 26.71 sec
Time at the end of MAX = 16 is 61.39 sec
Time at the end of MAX = 17 is 139.83 sec
Time at the end of MAX = 18 is 314.18 sec
Time at the end of MAX = 19 is 702.05 sec
Time at the end of MAX = 20 is 1555.91 sec
```

and in the file system we see

```
-rw-r--r-- 1 jos users     4956 Nov  8 14:20 expr6.sav
-rw-r--r-- 1 jos users    10516 Nov  8 14:20 expr7.sav
-rw-r--r-- 1 jos users    17500 Nov  8 14:20 expr8.sav
-rw-r--r-- 1 jos users    35828 Nov  8 14:20 expr9.sav
-rw-r--r-- 1 jos users    58748 Nov  8 14:20 expr10.sav
-rw-r--r-- 1 jos users   148788 Nov  8 14:20 expr11.sav
-rw-r--r-- 1 jos users   263292 Nov  8 14:20 expr12.sav
-rw-r--r-- 1 jos users   625044 Nov  8 14:20 expr13.sav
-rw-r--r-- 1 jos users  1086924 Nov  8 14:20 expr14.sav
-rw-r--r-- 1 jos users  2438532 Nov  8 14:20 expr15.sav
-rw-r--r-- 1 jos users  4167660 Nov  8 14:21 expr16.sav
-rw-r--r-- 1 jos users 10327348 Nov  8 14:22 expr17.sav
-rw-r--r-- 1 jos users 18121404 Nov  8 14:25 expr18.sav
-rw-r--r-- 1 jos users 44197396 Nov  8 14:32 expr19.sav
-rw-r--r-- 1 jos users 76745788 Nov  8 14:46 expr20.sav
```

The idea is that when we want to do something we use just the file we need.

We can pick things up with a program like:

```
   #define MAX "6"
   Functions A,B,C,D;
   Symbols x(:‘MAX’),i,j;
   Format nospaces;
   Format 68;
   .global
   Load expr‘MAX’.sav;
 F loaded
   .sort
   Local FF = F;
   Print;
   B   x;
   .end

Time =       0.00 sec    Generated terms =         72
              FF         Terms in output =         72
                         Bytes used      =       3684
```

```
FF=
    +x*(A+B)
    +x^2*(1/2*A*B-1/2*B*A)
    +x^3*(1/12*A*A*B-1/6*A*B*A+1/12*A*B*B+1/12*B*A*A-1/6*B*A*B+1/
       12*B*B*A)
    +x^4*(1/24*A*A*B*B-1/12*A*B*A*B+1/12*B*A*B*A-1/24*B*B*A*A)
    +x^5*(-1/720*A*A*A*A*B+1/180*A*A*A*B*A+1/180*A*A*A*B*B-1/120*
       A*A*B*A*A-1/120*A*A*B*A*B-1/120*A*A*B*B*A+1/180*A*A*B*B*B+
       1/180*A*B*A*A*A-1/120*A*B*A*A*B+1/30*A*B*A*B*A-1/120*A*B*A
       *B*B-1/120*A*B*B*A*A-1/120*A*B*B*A*B+1/180*A*B*B*B*A-1/720
       *A*B*B*B*B-1/720*B*A*A*A*A+1/180*B*A*A*A*B-1/120*B*A*A*B*A
       -1/120*B*A*A*B*B-1/120*B*A*B*A*A+1/30*B*A*B*A*B-1/120*B*A*
       B*B*A+1/180*B*A*B*B*B+1/180*B*B*A*A*A-1/120*B*B*A*A*B-1/
       120*B*B*A*B*A-1/120*B*B*A*B*B+1/180*B*B*B*A*A+1/180*B*B*B*
       A*B-1/720*B*B*B*B*A)
    +x^6*(-1/1440*A*A*A*A*B*B+1/360*A*A*A*B*A*B+1/360*A*A*A*B*B*B
       -1/240*A*A*B*A*A*B-1/240*A*A*B*A*B*B-1/240*A*A*B*B*A*B-1/
```

```
1440*A*A*B*B*B*B+1/360*A*B*A*A*A*B-1/240*A*B*A*A*B*B+1/60*
A*B*A*B*A*B+1/360*A*B*A*B*B*B-1/240*A*B*B*A*A*B-1/240*A*B*
B*A*B*B+1/360*A*B*B*B*A*B-1/360*B*A*A*A*B*A+1/240*B*A*A*B*
A*A+1/240*B*A*A*B*B*A-1/360*B*A*B*A*A*A-1/60*B*A*B*A*B*A+1/
240*B*A*B*B*A*A-1/360*B*A*B*B*B*A+1/1440*B*B*A*A*A*A+1/240
*B*B*A*A*B*A+1/240*B*B*A*B*A*A+1/240*B*B*A*B*B*A-1/360*B*B
*B*A*A*A-1/360*B*B*B*A*B*A+1/1440*B*B*B*B*A*A);
```

Now we are ready for step 2 in which we write everything in terms of commutators. Of course we practise with a small value for MAX.

```
    #define MAX "6"
    Functions A,B,C,D;
    Symbols x(:‘MAX’),y,i,j;
    Off Statistics;
    .global
    Load expr‘MAX’.sav;
 F loaded
    .sort
    Local FF = F;
    #do i = 1,1
        repeat;
            if ( count(y,1) == 0 ) id,once,B*A = A*B-C*y;
        endrepeat;
        id y = 1;
        .sort
        Repeat;
```

```
            id  C(?a)*D?{A,B} = D*C(?a)-C(D,?a);
            id,disorder,C(?a)*C(?b) = C(?b)*C(?a)-C(C(?b),?a);
        EndRepeat;
        if ( match(B*A) ) Redefine i "0";
        .sort
    #enddo
    B   x;
    Print +f;
    .end

FF =
      + x * ( A + B )

      + x^2 * ( 1/2*C )

      + x^3 * ( 1/12*C(A) - 1/12*C(B) )

      + x^4 * ( - 1/24*C(A,B) )
```

```
+ x^5 * ( - 1/720*C(A,A,A) - 1/360*C(A,A,B) - 1/120*C(A,B,A)
   - 1/90*C(A,B,B) + 1/180*C(B,A,A) + 1/40*C(B,A,B) - 1/120*
  C(B,B,A) + 1/720*C(B,B,B) + 1/360*C(C,A) + 1/120*C(C,B) )

+ x^6 * ( - 1/240*C(C(B),A) + 1/480*C(A,A,A,B) - 1/240*C(A,A,B,A)
   - 1/180*C(A,A,B,B) + 1/360*C(A,B,A,A) + 1/80*C(A,B,A,B)
   - 1/240*C(A,B,B,A) + 1/1440*C(A,B,B,B) + 1/720*C(C,A,A)
   + 1/240*C(C,A,B) );
```

We have several sophisticated constructions here. First there is the trick with the variable y. We want to introduce one commutator at a time. The problem is that one term has a commutator and the other does not. But because at a later stage we may need more commutators we cannot just say that we want to repeat as long as there are no commutators. We solve this problem with the variable y which we eliminate once we do not need it any longer. The next step is the most costly. We start building up the commutators. Hence we sort first to keep the number of terms at a minimum. Then we move the commutators to the right, moving the A and B elements to the left.

When two commutators meet each other the option disorder makes that the substitution is only made when the functions would be out of order if they were commuting functions. This option is needed because

```
Repeat;
    id  C(?a)*D?{A,B} = D*C(?a)-C(D,?a);
    id  C(?a)*C(?b) = C(?b)*C(?a)-C(C(?b),?a);
EndRepeat;
```

would result in an infinite loop. The answer comes rather fast. At 10 we need 0.57 sec and at 12 we need 10.68 sec. Yet it can be made somewhat faster:

```
#do j = 1,1
    id  C(?a)*D?{A,B} = D*C(?a)-C(D,?a);
    if ( match(C(?a)*D?{A,B}) ) Redefine j "0";
    .sort
#enddo
Repeat;
    id,disorder,C(?a)*C(?b) = C(?b)*C(?a)-C(C(?b),?a);
EndRepeat;
.sort
Repeat;
    id  C(?a)*D?{A,B} = D*C(?a)-C(D,?a);
    id,disorder,C(?a)*C(?b) = C(?b)*C(?a)-C(C(?b),?a);
EndRepeat;
if ( match(B*A) ) Redefine i "0";
```

This brings the time at 10 back to 0.32 sec, at 12 to 2.78 sec. and at 14 to 24.10 sec.

When we inspect the output we see that there are C-functions inside C-functions. This is not what we want. For this we need some extra relations. It will be homework to prove them. In FORM notation they are:

```
repeat id,C(?a,C(D?(?d),?c),?b) =
                        C(?a,D(?d),C(?c),?b)-C(?a,C(?c),D(?d),?b);
repeat id   C(?a,C,?b) = C(?a,A,B,?b)-C(?a,B,A,?b);
```

What it means effectively is that we can expand commutators inside commutators. If we add these two lines to the previous program with a .sort between them we get the answer as we want to have it. For the case of the program at 14 the time becomes now 31 sec.

```
FF =
    + x * ( A + B )

    + x^2 * ( 1/2*C )

    + x^3 * ( 1/12*C(A) - 1/12*C(B) )

    + x^4 * (  - 1/24*C(A,B) )

    + x^5 * (  - 1/720*C(A,A,A) - 1/360*C(A,A,B) - 1/180*C(A,B,A)
       - 1/360*C(A,B,B) + 1/360*C(B,A,A) + 1/60*C(B,A,B)
       - 1/120*C(B,B,A) + 1/720*C(B,B,B) )

    + x^6 * ( 1/480*C(A,A,A,B) - 1/240*C(A,A,B,A) - 1/180*C(A,A,B,B)
       + 1/240*C(A,B,A,A) + 1/60*C(A,B,A,B) + 1/1440*C(A,B,B,B)
       - 1/720*C(B,A,A,A) - 1/240*C(B,A,A,B) - 1/120*C(B,A,B,A)
       + 1/240*C(B,B,A,A) )
```

The next step is to store the results so that phase 3 can make use of the above results. For this we can make a similar construction as for the end of phase 1. Now however the program is more costly in resources. The tail of the program becomes

```
B   x;
.sort
Delete Storage;
#do i = 1,‘MAX’
    Global FF‘i’ = FF[x^‘i’];
#enddo
.store
Save CBHcom‘MAX’.sav;
.end
```

and we run it for $MAX = 18$. This is something to do just once and it takes 3510 sec. After this we can take any set of commutators.

This finishes basically step two of our project. The next question is whether we can make the output shorter. The answer is yes. The sixth order for instance can be written with 4 terms only.

How do we find relations? Let us have a look at the object $C(C)$. On the one hand this is $[C, C] = 0$ but we can also work it out as $C(A, B) - C(B, A)$. Hence we obtain the relations

$$\begin{aligned} C(A, B) - C(B, A) &= 0 \\ C(X, A, B) - C(X, B, A) &= 0 \end{aligned}$$

in which X is any set of additional arguments. We can play similar games with objects of the type $C(C(X), X)$. It must be zero, but we can also work out the inner $C(X)$ and hence end up with relations. For objects of the type $C(C(X), Y), X \neq Y$ we have to use

$$\begin{aligned} C(C(X), Y) &= C(X)\, C(Y) - C(Y)\, C(X) \\ &= -C(C(Y), X) \end{aligned} \tag{8}$$

and working out the inner C in $C(C(X), Y) + C(C(Y), X) = 0$ will give more relations.

How do we generate the complete set of relations, taking into account that there may be many? Let us start with a simple program:

```
Functions A,B,C,D;
Symbol x;
Off Statistics;
.global
#define MAX "6"
#do i = 4,`MAX',2
Local F`i' = C((`i'-4)/2)^2;
repeat id C(x?pos_,?a) = C(x-1,A,?a)+C(x-1,B,?a);
id  C(0,?a) = C(?a);
id  C(?a)*C(?b) = C(C(?a),?b);
Print +f +s;
.store

F4 =
    + C(C)
   ;
```

```
 #enddo

F6 =
    + C(C(A),A)
    + C(C(A),B)
    + C(C(B),A)
    + C(C(B),B)
   ;

 .end
```

This program generates all nested commutators of the type we are after. We want the product of two C functions each with the same number of arguments, in such a way that if we work out all commutators we have a total of 6 objects A or B. The number of them grows as a power of 2 (or 4, depending on how one looks at it). We have to combine pairs though. We can do this with a little extra complication:

```
 Functions A,B,C,D;
 CFunction rel;
 Symbol x;
 Off Statistics;
 .global
 #define MAX "6"
 #do i = 4,`MAX',2
 Local F`i' = C((`i'-4)/2)^2;
 repeat id C(x?pos_,?a) = C(x-1,A,?a)+C(x-1,B,?a);
 id  C(0,?a) = C(?a);
 id  C(?a)*C(?b) = C(C(?a),?b);
 id  C(?a) = rel(C(?a));
 Print +f +s;
 .store

F4 =
    + rel(C(C))
   ;
```

```
 #enddo

F6 =
    + rel(C(C(A),A))
    + rel(C(C(A),B))
    + rel(C(C(B),A))
    + rel(C(C(B),B))
   ;

 .end
```

Now we create the relations with the extra statements (we only show the last part of the output):

```
id  C(?a) = rel(C(?a));
Argument rel;
    if ( match(C(C(?a),?a)) == 0 );
        id  C(C(?a),?b) = C(C(?a),?b)+C(C(?b),?a);
    endif;
EndArgument;
.sort
DropCoefficient;
Print +f +s;
.store

F4 =
    + rel(C(C))
   ;

#enddo
```

```
F6 =
     + rel(C(C(A),A))
     + rel(C(C(A),B) + C(C(B),A))
     + rel(C(C(B),B))
    ;

  .end
```

The Argument/EndArgument construction causes the statements inside to operate only on the terms in the argument(s) of the indicated function rel. What we do there is symmetrize the argument if it was not symmetric in the ?a and ?b arguments of C. The result is that there are two terms (for MAX=6) for which this is done and these terms become now identical. This factor two after the sorting we do not want. Hence we kill that coefficient of the term with the DropCoefficient statement.

The next step is to expand the inner commutators C:

```
  DropCoefficient;
  Argument rel;
     repeat id C(?a,C(D?,?b),?c) = C(?a,D,C(?b),?c)-C(?a,C(?b),D,?c);
     id C(?a,C,?b) = C(?a,A,B,?b)-C(?a,B,A,?b);
  EndArgument;
  Print +f +s;
  .store

 F4 =
     + rel(C(A,B) - C(B,A))
    ;

  #enddo

 F6 =
     + rel(C(A,A,B,A) - 2*C(A,B,A,A) + C(B,A,A,A))
     + rel(C(A,A,B,B) - 2*C(A,B,A,B) - C(A,B,B,A) + C(B,A,A,B) +
```

```
  2*C(B,A,B,A) - C(B,B,A,A))
   + rel( - C(A,B,B,B) + 2*C(B,A,B,B) - C(B,B,A,B))
  ;
.end
```

and in principle we have our relations. It is however possible to simplify the relations in F6 a bit by applying the relation in F4. In F8 we would have to apply the relations of F4 and F6. Etc. Doing this may actually make the number of relations less. For now we will just take the output and generate id statements from them by a text editor. It is possible to do this automatically, but for now that would be too complicated.

```
DropCoefficient;
Argument rel;
   repeat id C(?a,C(D?,?b),?c) = C(?a,D,C(?b),?c)-C(?a,C(?b),D,?c);
   id C(?a,C,?b) = C(?a,A,B,?b)-C(?a,B,A,?b);
   #if ( `i' > 4 )
      id  C(?a,B,A) = C(?a,A,B);
   #endif
   #if ( `i' > 6 )
      id  C(?a,A,B,A,A) = ( C(?a,B,A,A,A) + C(?a,A,A,A,B) )/2;
```

```
        id  C(?a,B,B,A,A) = C(?a,A,A,B,B) - 3*C(?a,A,B,A,B)
               + 3*C(?a,B,A,A,B);
        id  C(?a,B,A,B,B) = ( C(?a,A,B,B,B) + C(?a,B,B,A,B) )/2;
    #endif
 EndArgument;
 MakeInteger;
 .sort
 DropCoefficient;
 Print +f +s;
 .store

F4 =
    + rel(C(A,B) - C(B,A))
   ;

 #enddo

F6 =
```

```
    + rel(C(A,A,A,B) - 2*C(A,B,A,A) + C(B,A,A,A))
    + rel(C(A,A,B,B) - 3*C(A,B,A,B) + 3*C(B,A,A,B) - C(B,B,A,A))
    + rel(C(A,B,B,B) - 2*C(B,A,B,B) + C(B,B,A,B))
   ;


F8 =
    + rel(C(A,A,A,A,A,B) - 5*C(A,A,B,A,A,A) + 6*C(A,B,A,A,A,A)
    - 2*C(B,A,A,A,A,A))
    + rel(C(A,A,A,A,B,B) - 4*C(A,A,A,B,A,B) + 6*C(A,A,B,A,A,B)
    - 4*C(A,B,A,A,A,B) - C(A,B,B,A,A,A) + C(B,A,A,A,A,B) + 2*C(B
   ,A,B,A,A,A) - C(B,B,A,A,A,A))
    + rel(C(A,A,A,B,B,B) + 3*C(A,A,B,B,A,B) - 8*C(A,B,A,A,B,B)
    + 6*C(A,B,A,B,A,B) - 6*C(A,B,B,A,A,B) + 8*C(B,A,A,A,B,B) -
   18*C(B,A,A,B,A,B) + 18*C(B,A,B,A,A,B) - 3*C(B,B,A,A,A,B) - C(
   B,B,B,A,A,A))
    + rel(C(A,A,B,B,A,B) - 2*C(A,B,A,B,A,B) + 2*C(B,A,B,A,A,B)
    - C(B,B,A,A,A,B))
```

```
 + rel(C(A,A,B,B,B,B) - 2*C(A,B,A,B,B,B) - C(A,B,B,B,A,B) +
C(B,A,A,B,B,B) + 4*C(B,A,B,B,A,B) - C(B,B,A,A,B,B) - 3*C(B,B,
A,B,A,B) + C(B,B,B,A,A,B))
 + rel(2*C(A,B,B,B,B,B) - 6*C(B,A,B,B,B,B) + 5*C(B,B,A,B,B,B)
 - C(B,B,B,B,A,B))
;
.end
```

In this program we run to 8. The number of relations for 8 goes down from 10 to 6 due to the relations at 4 and 6. One can also see that these relations become rather complicated.

As a final test, before we continue, we can check that these are indeed correct relations. We do this by expanding the commutators:

```
 DropCoefficient;
 Argument rel;
     repeat id C(D?,?a) = D*C(?a)-C(?a)*D;
     id  C = A*B-B*A;
 EndArgument;
 Print +f +s;
 .store

F4 =
    + rel(0)
   ;

 #enddo

F6 =
    + 3*rel(0)
```

```
    ;


F8 =
     + 6*rel(0)
    ;

  .end
```

and we see that all relations do indeed collapse to zero.

The above is a rather messy way to find good relations. Is there a better way? Just when I was considering this, I attended a talk by Gerard 't Hooft who, as luck would have it, needed CBH formulas for something. And he defined the objects $X = A + B$ and $Y = A - B$ to work with in order to get shorter relations. This is an interesting idea. Let us see how that works.

The first thing we have to do is define new objects:

$$\begin{aligned} A &= X + Y \\ B &= X - Y \\ Z &= [X, Y] \\ &= [A + B, A - B]/4 \\ &= -C/2 \end{aligned}$$

Next we have to create a procedure that rewrites our commutators to the Z, X,Y system. Such a procedure could be

```
#procedure CAB2ZXY
*
*   Procedure to rewrite commutators of the type
*   C(A,B,A,A,A,B,A,...) to commutators of the type
*   Z(X,X,Y,Y,X,Y,...) with
*       A = X+Y;
*       B = X-Y;
*   hence:
*       Z = [X,Y] = [A+B,A-B]/4 = -[A,B]/2 = -C/2;
*
id  C(?a) = -2*Z*C(?a);
id  A = X+Y;
id  B = X-Y;
#$CZcount = 0;
#do iCAB2ZXY = 1,1
    id,Z(?a)*C(A,?b) = Z(?a,X)*C(?b)+Z(?a,Y)*C(?b);
    al,Z(?a)*C(B,?b) = Z(?a,X)*C(?b)-Z(?a,Y)*C(?b);
    if ( match(C(A?,?a)) ) redefine iCAB2ZXY "0";
```

```
        #$CZcount = $CZcount + 1;
        .sort:CAB2ZXY-‘$CZcount’;
    #enddo
    id  C = 1;
    .sort:CAB2ZXY-finish;
    *
    #endprocedure
```

and we have a new feature again.

In the loop we have an id statement, but also a statement that starts with al. This is the also statement. What happens is that when we make a pattern matching with the id statement and if it is successful, actually we do not want to execute a second statement. But when the first does not match we do want to try for the second. This guarantees that the remaining C will have always the same number of arguments and the chance for cancellations is maximal. Hence what the id/al combination does is: It tries the id. If there is a match, it takes out the matching pattern, but does not insert the RHS yet. Then it offers the remains of the term to the following al(so) statement which also tries to take out its pattern. It keeps doing this till there are no more al(so) statements. Only then, all RHS's are inserted.

```
    Symbols x,y,cos,sin;
    Format 60;
    Local F = x^2+y^2;
    id  x = x*cos-y*sin;
    id  y = x*sin+y*cos;
    id  sin^2 = 1-cos^2;
    Print;
    .sort

Time =       0.00 sec    Generated terms =          15
               F         Terms in output =          10
                         Bytes used      =         332

   F =
      2*y^2*cos^2 - y^2*cos^4 + 4*x*y*cos*sin - 2*x*y*cos^2
      *sin - 2*x*y*cos^3*sin + 2*x^2 - 2*x^2*cos - 2*x^2*
      cos^2 + 2*x^2*cos^3 + x^2*cos^4;
```

```
    Drop;
    Local G = x^2+y^2;
    id  x = x*cos-y*sin;
    al  y = x*sin+y*cos;
    id  sin^2 = 1-cos^2;
    Print;
    .end

Time =       0.00 sec    Generated terms =          8
               G         Terms in output =          2
                         Bytes used      =         64

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

In what we are doing we could also have used the ifmatch construction as in

```
id,ifmatch->1,Z(?a)*C(A,?b) = Z(?a,X)*C(?b)+Z(?a,Y)*C(?b);
id,Z(?a)*C(B,?b) = Z(?a,X)*C(?b)-Z(?a,Y)*C(?b);
label 1;
```

If you really think about it, the id/al construction is in principle more powerful as shown in the example above which does not have an easy equivalent with ifmatch.

Let us run the following program

```
   #define MAX "6"
   Functions A,B,C,D,E,X,Y,Z;
   AutoDeclare Function A;
   AutoDeclare Symbols x;
   Symbols x(:‘MAX’),y,z,i,j;
   CFunction rel,fcount;
   Format 68;
   Off Statistics;
   .global
   Load CBHcom18.sav;
FF1 loaded
FF2 loaded
FF3 loaded
FF4 loaded
FF5 loaded
FF6 loaded
FF7 loaded
```

```
FF8 loaded
FF9 loaded
FF10 loaded
FF11 loaded
FF12 loaded
FF13 loaded
FF14 loaded
FF15 loaded
FF16 loaded
FF17 loaded
FF18 loaded
   .sort
   #do i = 1,‘MAX’
   L   G‘i’ = FF‘i’;
   #enddo
   .sort
   Delete Storage;
   *
```

```
 *   First we rewrite to X and Y.
 *
 #call CAB2ZXY
 Print +f;
 .end

G1 =
   2*X;

G2 =
    - Z;

G3 =
    - 1/3*Z(Y);

G4 =
   1/12*Z(X,X) - 1/12*Z(X,Y) + 1/12*Z(Y,X) - 1/12*Z(Y,Y);
```

```
G5 =
   1/20*Z(X,X,Y) - 11/180*Z(X,Y,X) + 1/20*Z(Y,X,X) - 2/45*Z(Y,X,
   Y) + 2/45*Z(Y,Y,X) - 1/60*Z(Y,Y,Y);

G6 =
    - 1/120*Z(X,X,X,X) + 11/360*Z(X,X,X,Y) - 7/90*Z(X,X,Y,X) +
   17/180*Z(X,Y,X,X) - 1/180*Z(X,Y,X,Y) + 7/360*Z(X,Y,Y,X) - 11/
   360*Z(X,Y,Y,Y) - 17/360*Z(Y,X,X,X) + 1/40*Z(Y,X,X,Y) - 1/20*
   Z(Y,X,Y,X) + 11/180*Z(Y,X,Y,Y) + 1/45*Z(Y,Y,X,X) - 2/45*Z(Y,Y
   ,X,Y) + 1/72*Z(Y,Y,Y,X) - 1/360*Z(Y,Y,Y,Y);
```

This seems to work rather well. We can however make one observation. The terms with an odd number of X's add up to zero. We can see this if we expand all those terms back into X,Y. For this we make a procedure that determines the number of X's and expands the commutators when this number is odd.

```
#procedure expoddX
*
*   We expand the Z that have an odd number of X's inside.
*   First count the X's;
*
id  Z(?a) = E*Z(?a);
repeat;
    id  E(?a)*Z(X,?b) = E(?a,X)*Z(?b)*y;
    id  E(?a)*Z(Y,?b) = E(?a,Y)*Z(?b);
endrepeat;
id  E(?a)*Z = Z(?a);
B   y,z;
.sort:expoddX: counting;
id  y = y*z;
id  z^2 = 1;
#$countexpoddX = 0;
#do jexpoddX = 1,1
    if ( count(z,1) );
```

```
        id Z(A?,?a) = A*Z(?a)-Z(?a)*A;
        if ( match(Z(A?,?a)) ) redefine jexpoddX "0";
    endif;
    #$countexpoddX = $countexpoddX+1;
    .sort:expoddX-`$countexpoddX';
#enddo
if ( count(z,1) );
    id  Z = X*Y-Y*X;
endif;
B   y,z;
.sort:expoddX-finish;
#endprocedure
```

By moving the arguments one by one to the function E, we can count the X's. Their number is the power of y. Next we copy y to $y \times z$ and take out the even powers of z. This means that if a term has a z left, it has an odd number of X's and if it has an even number of X's there is no z left. The rest is trivial. The result is

```
#call CAB2ZXY
*
*   Take out odd powers of X.
*
#call expoddX
Bracket y;
Print +f;
.end

G1 =
    + 2*X;

G2 =
    - Z;
```

```
G3 =
    - 1/3*Z(Y);

G4 =
    + y^2 * ( 1/12*Z(X,X) )

    - 1/12*Z(Y,Y);

G5 =
    + y^2 * ( 1/20*Z(X,X,Y) - 11/180*Z(X,Y,X) + 1/20*Z(Y,X,X) )

    - 1/60*Z(Y,Y,Y);

G6 =
    + y^2 * (  - 1/180*Z(X,Y,X,Y) + 7/360*Z(X,Y,Y,X) + 1/40*Z(Y,
      X,X,Y) - 1/20*Z(Y,X,Y,X) + 1/45*Z(Y,Y,X,X) )
```

```
+ y^4 * (  - 1/120*Z(X,X,X,X) )

- 1/360*Z(Y,Y,Y,Y);
```

and indeed the terms with odd powers of X (and hence y) are absent.

This program is rather fast. For MAX=14 it takes 2.65 sec, for MAX=16 15.83 sec. and for MAX=18 it takes 94 sec. Most of this time is spent in the expoddX procedure. Having checked that these odd powers all vanish, we may as well save this time and set these terms directly to zero by adding the statement

```
id  y = y*z;
id  z^2 = 1;
id  z = 0;
```

that makes the odd power of z equal to zero in the procedure expoddX. This brings the last time down to 16.12 sec.

Now we want to see how we can automatically create simplifying relations. As of now we have 2^{i-3} terms for G_i.

What we will do now is create a table with relations. Let us first discuss the tables we want to use. We will transform the X's and the Y's to 1's and 2's. Then the commutators will have only ones and twos and we can define proper tables. But we will need only a small subset of all possibilities. For this we have sparse tables. When we define a sparse table we only mention the number of table indexes (and then possibly extra arguments). FORM will then only allocate space for an element when it is defined with a fill statement.

The complete procedure that does this definition is given by

```
#procedure makeZtable(j)
*
*   Creates the relations of the type
*       Z(Z(A1,...,Aj),B1,...,Bj)+Z(Z(B1,...,Bj),A1,...,Aj)=0
*   Puts them in the table Ztable`j'.
*
*   First generate all relations:
*
Local relations = D(`j')^2;
repeat id D(i?pos_,?a) = D(i-1,X,?a)+D(i-1,Y,?a);
id,disorder,D(0,?a)*D(0,?b) = 0;
id  D(0,?a)*D(0,?b) = rel(Z(Z(?a),?b)+Z(Z(?b),?a));
makeinteger rel;
DropCoefficient;
.sort:makeZtable;
*
*   Next we expand the relations. This may give the problem that
*   they become too long to be kept inside a single term. Hence
```

```
*   we have to mark the relations with a function fcount(number)
*
#$Znumber = 0;
$Znumber = $Znumber+1;
Multiply fcount($Znumber);
id  rel(x?) = x;
repeat id   Z(?a,Z(A?,?b),?c) = Z(?a,A,Z(?b),?c)-Z(?a,Z(?b),A,?c);
id  Z(?a,Z,?b) = Z(?a,X,Y,?b)-Z(?a,Y,X,?b);
*
*   Next we apply the tables for as far as they exist yet.
*   We have to assume that the tables Ztable‘i’ with
*   0 <= i < j exist.
*
*   An extra complication is that for instance Z(X,Y) = Z(Y,X) will
*   work on all Z(?a,X,Y) and the table has only Ztable(X,Y).
*   Yet another complication is that tables work mostly with
*   numbers. *  This is why we transform first to 1,2 instead of X,Y
*
```

```
id  Z(?a) = D*Z(?a);
repeat;
    id  D(?a)*Z(X,?b) = D(?a,1)*Z(?b);
    id  D(?a)*Z(Y,?b) = D(?a,2)*Z(?b);
endrepeat;
id  D(?a)*Z = Z(?a);
B+  fcount;
.sort
#do imakeZtable = 2,2*‘j’,2
    id  Z(?a,x1?,...,x‘imakeZtable’?) =
          D(?a)*Ztable{‘imakeZtable’/2-1}(x1,...,x‘imakeZtable’);
    id  D(?a)*Ztable{‘imakeZtable’/2-1}(?b) = Z(?a,?b);
    id  D(?a)*Z(?b) = Z(?a,?b);
    Bracket+ fcount;
    .sort:makeZtable-Use tables-‘imakeZtable’;
#enddo
*
*   It could be that some relations have become identical.
```

```
*   This will get sorted out automatically.
*   We are now going to apply the relations one by one to the others.
*   It is a Gaussian elimination.
*
#do c = 1,`$Znumber'
    #$eq = relations[fcount(`c')];
    #if ( termsin($eq) > 0 )
        #$lhs = firstterm_($eq);
        #inside $lhs
            $coeff = coeff_;
            Dropcoefficient;
        #endinside
        #$rhs = -($eq/($coeff)-$lhs);
        if ( match(fcount(`c')) == 0 )
            id `$lhs' = `$rhs';
        Bracket+ fcount;
        .sort:makeZtable-relation `c';
    #endif
```

```
#enddo
NTable,sparse,Ztable‘j’({2*‘j’+2});
#do c = 1,‘$Znumber’
    #$eq = relations[fcount(‘c’)];
    #if ( termsin($eq) > 0 )
        #$lhs = firstterm_($eq);
        #inside $lhs
            $coeff = coeff_;
            Dropcoefficient;
        #endinside
        #$rhs = -($eq/($coeff)-$lhs);
        #inside $lhs
            Multiply replace_(Z,Ztable‘j’);
        #endinside
        Fill ‘$lhs’ = ‘$rhs’;
    #endif
#enddo
Drop relations;
```

```
.sort:makeZtable-finish;
*
#endprocedure
```

The start of the procedure is a bit like what we have seen before for creating relations. The makeinteger command multiplies the argument of the indicated function by a fraction such that all coefficients of the terms in the argument are integer and have a GCD of one. Then the outside of the function is divided by this fraction.

Next we multiply each relation by a function with a unique argument. We do this by raising $Zcount by one for each term. The reason we have to do this is because when we expand these relations for a large value of MAX there may be very many terms inside the function rel. More than FORM can handle in a single term. Hence we multply each of these relations by this function fcount with a unique argument and we can work at ground level. We can recover the relations by bracketting in fcount.

The Multiply statement does what it says: it multiplies the current term by the argument of the multiply statement.

The instruction (this takes place in the preprocessor)

```
#$eq = relations[fcount('c')];
```

puts the relation 'c' in the $-variable $eq. If this relation is of the type

```
coef1*term1+coef2*term2+...+coefn*termn
```

we want to create the equivalent of the substitution

```
id term1 = -(coef2*term2+...+coefn*termn)/coef1;
```

For this we have to pick up the first term which can be done with the function firstterm_, put it inside another $-variable, pick up its coefficient which we do by looking inside the $-variable and then we generate the LHS and the RHS of the id statement that we want to have. Inside the preprocessor we can use these to create the id statement. We apply it to all other relations. This is some kind of Gaussian elimination scheme.

Finally we define the table. This is a non-commuting table and hence we use the NTable statement. The table is sparse and has `{2*'j'+2}` arguments. We create the LHS and RHS again, but now the LHS is used to indicate the table element. This involves a new function: replace_. This function should have pairs of arguments. In the term in which it is encountered it causes the replacement of the odd arguments by their next even argument. It does so in the complete term, also inside function arguments. Sometimes this is the fastest way to make a number of replacements. It can also be used to exchange objects:

```
Multiply replace_(X,Y,Y,X);
```

Of course this is a rather sophisticated routine. Let us see how it works.

```
#call expoddX
Hide;
#do i = 0,‘MAX’/2-2
    #call makeZtable(‘i’)
#enddo

Unhide;

id  Z(?a) = D*Z(?a);
repeat;
    id  D(?a)*Z(X,?b) = D(?a,1)*Z(?b);
    id  D(?a)*Z(Y,?b) = D(?a,2)*Z(?b);
endrepeat;
id  D(?a)*Z = Z(?a);
#do i= 2,‘MAX’-2,2
    id  Z(?a,x1?,...,x‘i’?) = D(?a)*Ztable{‘i’/2-1}(x1,...,x‘i’);
    id  D(?a)*Ztable{‘i’/2-1}(?b) = Z(?a,?b);
    id  D(?a)*Z(?b) = Z(?a,?b);
```

```
    .sort:table subs-‘i’;
#enddo
Transform Z,replace(1,last)=(1,X,2,Y);
.sort
B   x,y,z;
Print +f +s;
.end
```

We start with hiding the expressions, so that they are not affected by what happens inside makeZtable. Once makeZtable has defined all its table elements we unhide the expressions. Next we have to transform X,Y to one, two.

In the loop we replace each Z by the corresponding table element. If this element exists it will be replaced, and if it does not exist the replace it back.

The Transform statement is a rather complicated statement. Here we use it to replace in all the arguments of Z the argument 1 by X and the argument 2 by Y. This finishes the program.

The results are:

```
G4=
   +y^2*(
      +1/12*Z(X,X)
      )
   -1/12*Z(Y,Y)
      ;
G5=
   +y^2*(
      -1/90*Z(X,Y,X)
      +1/20*Z(Y,X,X)
      )
   -1/60*Z(Y,Y,Y)
      ;
G6=
   +y^2*(
      +1/72*Z(X,Y,Y,X)
      -1/40*Z(Y,X,Y,X)
```

```
        +1/45*Z(Y,Y,X,X)
        )
    +y^4*(
        -1/120*Z(X,X,X,X)
        )
    -1/360*Z(Y,Y,Y,Y)
        ;
G7=
    +y^2*(
        -1/2520*Z(X,Y,X,Y,Y)
        -1/7560*Z(X,Y,Y,Y,X)
        +13/1512*Z(Y,X,Y,Y,X)
        -97/7560*Z(Y,Y,X,Y,X)
        +1/140*Z(Y,Y,Y,X,X)
        )
    +y^4*(
        -17/7560*Z(X,X,Y,X,X)
        +37/7560*Z(X,Y,X,X,X)
```

```
        -17/2520*Z(Y,X,X,X,X)
        )
    -1/2520*Z(Y,Y,Y,Y,Y)
        ;
G8=
    +y^2*(
        +1/2880*Z(X,Y,Y,X,Y,Y)
        -1/1344*Z(Y,X,Y,X,Y,Y)
        -11/30240*Z(Y,X,Y,Y,Y,X)
        +233/60480*Z(Y,Y,X,Y,Y,X)
        -31/6720*Z(Y,Y,Y,X,Y,X)
        +13/6720*Z(Y,Y,Y,Y,X,X)
        )
    +y^4*(
        -7/4320*Z(X,X,Y,X,Y,X)
        -1/320*Z(X,X,Y,Y,X,X)
        +19/1728*Z(X,Y,X,Y,X,X)
        -17/4320*Z(X,Y,Y,X,X,X)
```

```
    -71/20160*Z(Y,X,X,Y,X,X)
    +1/840*Z(Y,X,Y,X,X,X)
    -1/840*Z(Y,Y,X,X,X,X)
    )
+y^6*(
    +17/20160*Z(X,X,X,X,X,X)
    )
-1/20160*Z(Y,Y,Y,Y,Y,Y)
    ;
```

For the odd G's these are the shortest expressions I have encountered. For the even G's they are not the very shortest, but they come close. For G8 there are 15 terms, while in terms of A,B with a reduction that is hand guided it is possible to obtain 13 terms.

The execution times are

```
MAX =  6                                5       8       5
MAX =  7                                9      16      18
MAX =  8                               15      32      17
MAX =  9                               30      64
MAX = 10                               51     128      54
MAX = 11                               99     256
MAX = 12     0.79 sec                 178     512     295     0.69 sec
MAX = 13     1.00 sec                 351    1024
MAX = 14    21.51 sec                 621    2048     839    22.43 sec
MAX = 15    22.60 sec                1233    4096
MAX = 16   947.76 sec                2242    8192
```

The execution time goes up steeply. Nearly all of it is spent in the Gaussian elimination scheme.

Homework: Modify makeZtable such that it will work on our C,A,B representation and apply it there.