ONE

FORM is a program cq programming language for speedy and large scale manipulation of formulas.

FORM is NOT like Mathematica! They are completely different! This is very noticeable already in how formulas are treated:

- In Mathematica (and Maple and nearly all CA systems) formulas are represented as trees.
- In FORM a formula is a sequence of terms (a queue).

This difference in representation is largely responsible for the enormous difference in speed and storage needs. It also means that there are completely different types of restrictions.

For this course, please forget about Mathematica. Here you have to think in different terms.

FORM is a batch program. This means that you prepare a FORM program in your favorite text editor, let FORM execute the program and finally you study the results. Its installation is relatively simple. There are two ways to install it:

- You download one of the executables in the FORM homepage (www.nikhef.nl/∼form). Disadvantage: lags behind in development and debugging.
- You download the sources (same homepage) and use the standard procedures to compile and install it (./configure; make). This is a bit more work, but it gives you the latest version. It also means that it should run on your computer.

There are three versions of FORM: sequential FORM, TFORM and ParFORM. The first is the 'normal' FORM, the second is for when you want to use more than one core at the same time and the third is for clusters of computers. In this course we will be using sequential FORM.

For now, please use the first download method and do not forget to make the program executable. For **homework** you are to try to use the second method of installation.

Let us take a very simple FORM program. Let us assume that you prepare the following program in your text editor and save it under the name prog1.frm.

```
Symbols a,b;
Local F = (a+b)^2;
Print;
.end
```

The extension .frm is mandatory. FORM will not execute files that do not have this extension. If you put your FORM executable in a location where the shell can find it (indicated by the path variable), which is usually either in the bin directory in your home directory, or in /usr/local/bin, you can execute this program by typing in a terminal window:

```
form prog1
```

followed by a <return>. FORM will attach the .frm extension automatically if you do not type it. The system should react with:

```
FORM 4.0 (Mar  1 2013) 64-bits                Run: Wed Mar  6 12:33:16 2013
    Symbols a,b;
    Local F = (a+b)^2;
    Print;
    .end

Time =       0.00 sec    Generated terms =          3
               F         Terms in output =          3
                         Bytes used      =        108

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

  0.00 sec out of 0.01 sec
```

Now is the time to see what we did.

In FORM formulas contain variables and FORM knows several types of variables. Hence all variables have to be declared. This avoids that the system needs to guess. The first type consists of the regular symbolic objects, called symbols. Here we declare a and b to be symbols. By the way: in statements like this the comma's are optional. You could also type one of the following

```
Symbols,a,b;
Symbols,a b;
Symbols   , a    b;
```

and FORM will interpret it correctly.

Also some keywords may be abbreviated. This is the case with the symbols keyword. Anything that starts with a letter s and is contained in the word 'symbols' will do. In addition, all system commands are case insensitive (all user defined objects are case sensitive). Hence

```
S a,b;
Sym a b;
sym a b;
sYmB a b;
SYMBOL a,b;
```

are all equivalent. After a while people have a tendency to use the first representation. Note that `S A,B;` is something different, because the variables are user defined.

The second statement defines a formula, or as we call it in FORM, an expression. There are two types of expressions in FORM, local and global expressions. We will be using mostly local expressions and we will discuss global expressions when we need them. Also the keyword local can be abbreviated all the way to a single character L. I will try to use the full words in the examples to avoid confusion, but the above program could have looked like

```
S a,b;
L F=(a+b)^2;
P;
.end
```

which is what many experienced programmers would do. Readability is another matter of course. (Also the keyword 'print' can be abbreviated). The print command tells FORM that we want it to print the result once it is done with this formula and the .end tells it that this is the end of the program and it should start working.

When you look at the output, you see that FORM first tells which version of FORM you are running and when. This is for your own administration. Then it echoes the program. Next it tells what it has been doing. These are called the statistics (if you do not want them, they can be turned off, but that is not always advisable). Then it prints the result of working out the formula. Finally it prints the CPU time and the real time elapsed since the start of the program.

In the following examples we will often omit some parts of the output, like the first and last lines and most of the statistics. Unless they are relevant of course.

At this point it should be noted that the coefficients of the terms are always either integers of fractions. FORM does not know floating point numbers. The size of the fractions is in principle unlimited, but there are practical limits (we are talking of something of $\mathcal{O}(2^{100000})$). FORM would give an error message in that case.

The first extension of our program is to see how powerful FORM really is (prog2.frm).

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

```
Time =      10.46 sec    Generated terms =    2760681
               F         Terms in output =    2760681
                         Bytes used      =  144759752
```

and as you can see, it does not take much effort to create lots of terms.

Of course, creating terms is relatively useless if we cannot do anything with them.

The executable statements of FORM are all meant to act on individual terms. During execution expressions as such do not exist. Only before the start of the execution and after it is finished we can refer to expressions.

The first executable statement and at the same time the most import one is the id-statement. It is the basic substitution. Its full name is 'identify' and this is an inheritance of Schoonschip. It is nearly always used as the abbreviated 'id' (Do not use 'i' because that is the abbreviation of something different). Its form is

```
id lhs = rhs;
```

The left hand side is also called a pattern and FORM will look in the terms whether it can take out one or more instances of the LHS and then replace them by equally many instances of the RHS.

```
*  prog3.frm
Symbols a,b,c,d;
Local F=(a+b+c+d)^20;
id d = 1-a-b-c;
Print;
.end
```

```
Time =       0.33 sec    Generated terms =     230230
               F         Terms in output =          1
                         Bytes used      =         20

   F =
      1;
```

The first line is commentary. Commentary has a star in column one. We also see that FORM makes no attempt to be smart. It does exactly what you ask, no more and no less. In this example that may look stupid, but that is just because we created a simple test of which we can verify the answer.

General rule: FORM does not try to be smarter than the user.

```
 Symbols a,b,c;
 Local F = (a+b)^6;
 id  a^2*b = c;
 Print +s;
 .end
F =
    + 15*c^2
    + 15*b^3*c
    + b^6
    + 20*a*b^2*c
    + 6*a*b^5
    + 6*a^3*c
    + a^6
   ;
```

(prog5.frm) Here we see that the LHS can be more complicated. The main restriction is that it must be a single term only and should have no numerical coefficient. Later in the course we will see much more complicated patterns.
The print statement can have the option "+s" to start each term on a new line.

The examples we have seen till now are rather simple. To make more complicated programs the first thing we need is the .sort instruction (prog6.frm).

```
Symbols a,b,c,d;
Local F = (a+b+c+1)^6;
id  a = -c+d+1;
id  b = -d+1;
Print;
.end
```

```
Time =       0.00 sec    Generated terms =         924
               F         Terms in output =           1
                         Bytes used      =          20

   F =
      729;
```

The above program substitutes first a and then b and only after that it sorts the results. This is sometimes (not always!) a bit wasteful. We can see this in the statistics. It gets a bit better if we force FORM to sort after the first substitution:

```
    Symbols a,b,c,d;
    Local F = (a+b+c+1)^6;
    id  a = -c+d+1;
    .sort

Time =       0.00 sec    Generated terms =         462
               F         Terms in output =          28
                         Bytes used      =         800
    id  b = -d+1;
    Print;
    .end

Time =       0.00 sec    Generated terms =          84
               F         Terms in output =           1
                         Bytes used      =          20

   F =
      729;
```

(prog6a.frm) We see that the total number of generated terms is 462+84 which is less than the 924 of the first program. We will see better examples later. First more about the .sort

The .sort marks the end of a 'module'. FORM programs consist of modules. Modules are terminated by an instruction that starts with a period, followed by the name of the instruction. We have seen .sort and .end and we will also encounter .global and .store at later stages.

FORM reads the input until it encounters an end-of-module instruction. The statements it reads in the meanwhile are being compiled and put in the main compiler buffer. When the end-of-module instruction has been read and there are no compilation errors or previous execution errors the statements of the module are executed on all terms of all expressions that are active at the moment, unless otherwise specified. When all terms of an expression have been treated by all statements of the module, the resulting terms are combined and sorted. If requested the expression(s) will be printed. After this the compiler buffer is cleared and FORM will read, compile and execute the next module. Etc.

At the level of modules FORM acts as an interpreter. At the level of the statements inside the modules FORM is a compiled language.

Terminology: The interpreted lines are called instructions, the compiled lines are called statements.

One of the skills in writing efficient FORM programs is either knowing or being able to figure out where to put the .sort instructions.

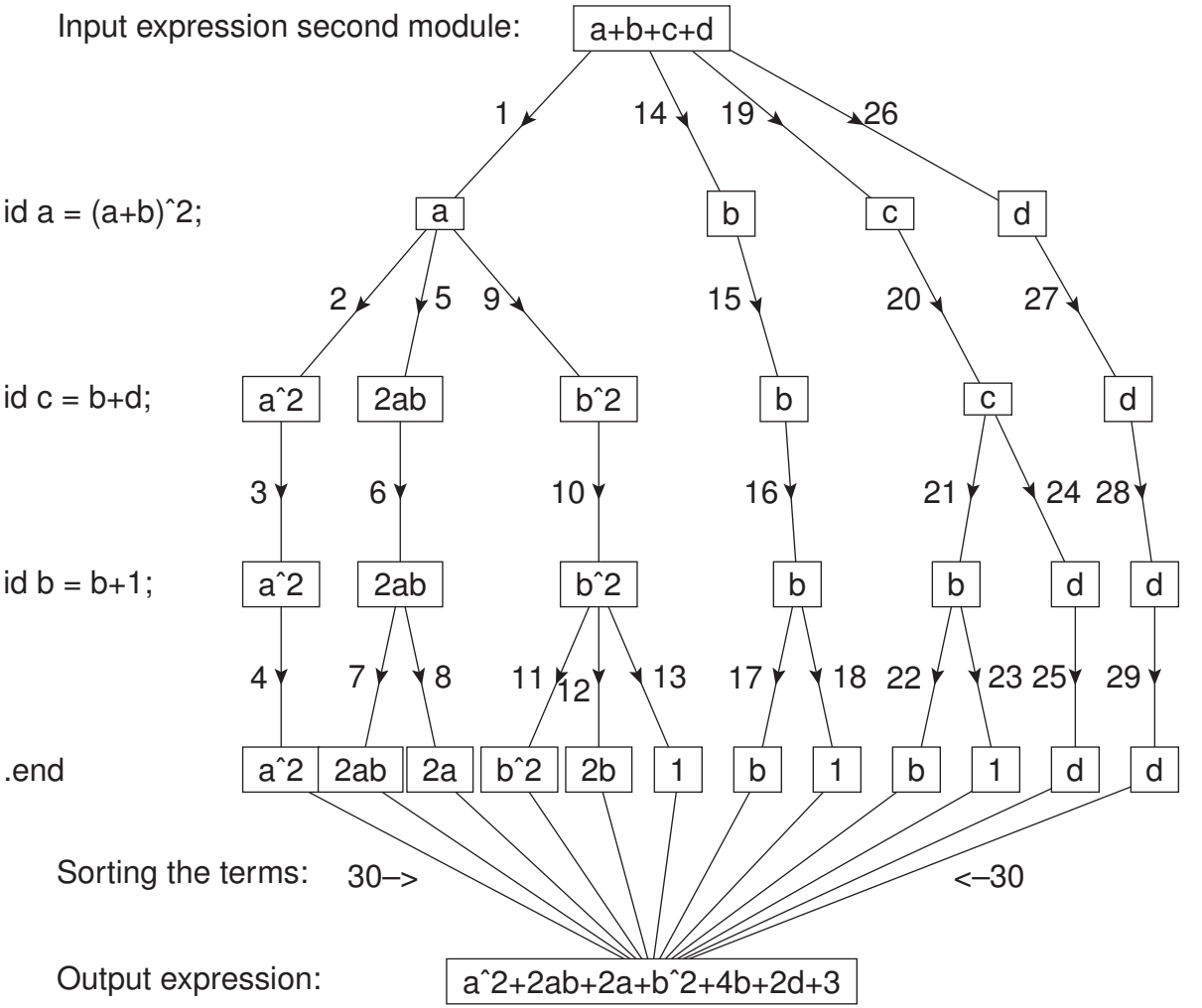
Let us have a look at the following program (prog7.frm):

```
    Symbols a,b,c,d;
    On HighFirst;
    Local F = a+b+c+d;
    .sort

Time =       0.00 sec    Generated terms =          4
               F         Terms in output =          4
                         Bytes used      =        120
    id a = (a+b)^2;
    id c = b+d;
    id b = b+1;
    Print;
    .end

Time =       0.00 sec    Generated terms =         12
               F         Terms in output =          7
                         Bytes used      =        204
```

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

The "On HighFirst;" statement controls the ordering of the variables. In this case it sorts in such a way that the high powers come first (default is low powers first).

```
S a,b,c,d;
On HighFirst;
L F = a+b+c+d;
.sort
```



How does FORM work? The picture represents the execution of the second module. At the start of the execution of a module the active expressions are sitting in the input scratch 'file'. The first term is taken and the LHS of the first statement tries a match. If there is a match the first term of the RHS is put in and the same will be done with the next statement, etc. until the end-of-module instruction is encountered. At this point the term is written to the 'small' sort buffer. We then backtrack to the last successful substitution and take there the second term. Etc. This way more and more terms end up in the small buffer. One of two things can happen now:

1. We reach the last term of the input and finish it before the small buffer is full.
2. At a given moment, before we are finished, the small buffer is full.

In either case the contents of the small buffer will be combined and sorted and statistics are written, but in the first case the results are written to the output scratch 'file', while in the second case the output is written to the 'large' sort buffer and the generation of terms is continued.

The above can be visualized in a different way by FORM itself (prog8.frm):

```
   Symbols a,b,c,d;
   On HighFirst;
   Local F = a+b+c+d;
   .sort
   Print " <1> %t";
   id a = (a+b)^2;
   Print "   <2> %t";
   id c = b+d;
   Print "     <3> %t";
   id b = b+1;
   Print "       <4> %t";
   Print;
   .end
<1>  + a
  <2>  + a^2
    <3>  + a^2
      <4>  + a^2
```

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

```
      <4>  + 1
    <3>  + d
      <4>  + d
<1>  + d
  <2>  + d
    <3>  + d
      <4>  + d

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

The print statement knows two forms: one without a string field, which controls which expressions will be printed after they have been processed, and one with a string field which works at the term level: each time a term passes by the print statement is executed. The string is a bit like the string of the printf statement in the C language. It has control sequences like the %t which indicates the current term. There are more control sequences, but we will treat them when needed. A full list is in the manual.

This version of the print statement is extremely useful when you have to debug your programs.

The sorting in FORM is only restricted by the size of the available disk space. There are many stages:

1. The small buffer is sorted. This is done in such a way that it would be very bad if even a small part of the small buffer would be swapped out. The results of this sorting is written as a 'sorted patch' into the large buffer.

2. If the large buffer is full, its contents are sorted. Because the buffer contains sorted patches it is much more robust against swapping. Its sorted contents are written as one sorted patch to the sort file.

3. When generation of terms is finished and all contents are in the sort file, the patches in the sort file are merged. Only a limited number of patches can be merged at the same time. If there are more, the result is written to another sort file. This is called 'stage 4 sorting'. If the sort can be done in one pass the result goes to the output scratch file and we are done.

4. In the case of stage 4 sorting, once the stage 4 file contains all terms we can eliminate the first sort file, rename the stage 4 file and repeat the previous step.

5. In the case of TFORM or ParFORM when processing takes place on more than one core

or processor, each will have an output and this output is then fed to the master processor who will merge these results and write to the master scratch file.

The user can tune the size of the various buffers to the size of the available memory. See the section on the setup parameters in the manual.

```
    #:SmallSize 2000
    #:LargePatches 4
    Symbols a,b,c,d;
    Local F1 = (a+b+c)^10;
    Local F2 = (a+b+c+d)^10;
    .end

Time =       0.00 sec    Generated terms =          44
              F1       1 Terms left      =          44
                         Bytes used      =        1576

Time =       0.00 sec    Generated terms =          66
              F1       1 Terms left      =          66
                         Bytes used      =        2324

Time =       0.00 sec    Generated terms =          66
              F1         Terms in output =          66
                         Bytes used      =        2320
```

```
Time =       0.00 sec    Generated terms =          43
              F2        1 Terms left      =          43
                          Bytes used      =        1708

Time =       0.00 sec    Generated terms =          82
              F2        1 Terms left      =          82
                          Bytes used      =        3268

Time =       0.00 sec    Generated terms =         121
              F2        1 Terms left      =         121
                          Bytes used      =        4788

Time =       0.00 sec    Generated terms =         159
              F2        1 Terms left      =         159
                          Bytes used      =        6284

Time =       0.00 sec    Generated terms =         197
```

```
             F2       1 Terms left      =         197
                        Bytes used      =        7764

Time =       0.00 sec
             F2         Terms active    =         197
                        Bytes used      =        7712

Time =       0.00 sec    Generated terms =         239
             F2       1 Terms left      =         239
                        Bytes used      =        9264

Time =       0.00 sec    Generated terms =         283
             F2       1 Terms left      =         283
                        Bytes used      =       10804

Time =       0.00 sec    Generated terms =         286
             F2       1 Terms left      =         286
                        Bytes used      =       10912
```

```
Time =       0.00 sec
              F2          Terms active    =         286
                          Bytes used      =       10852

Time =       0.00 sec    Generated terms =         286
              F2          Terms in output =         286
                          Bytes used      =       10832
```

(prog8a.frm) The statistics are illustrated here. Each time the small buffer is full, it is sorted and statistics are written. When the large buffer is full and sorted a different type of statistics is written. And after the last sort yet another type of statistics is written. They are characterized by the lines

$$\begin{aligned} \text{Terms left} &= 159 \\ \text{Terms active} &= 197 \\ \text{Terms in output} &= 286 \end{aligned}$$

The program is started with some settings of some of the buffers to make them artificially small for this example. We will get to them later.

FORM has a number of types of variables. There are symbols, vectors, indexes, commuting functions, non-commuting functions, tensors and sets. These are the algebraic variables. More variables of a completely different nature come in later sections. And there are the expressions, which are sequences of terms.

All algebraic variables have to be declared.

Let us start with the functions. They come in two varieties: commuting and non-commuting functions (prog9.frm).

```
    Functions A1,B1;
    CFunctions A2,B2;
    Local F1 = (A1+B1)^3;
    Local F2 = (A2+B2)^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 =
      A1*A1*A1 + A1*A1*B1 + A1*B1*A1 + A1*B1*B1 + B1*A1*A1 + B1*A1*B1
       + B1*B1*A1 + B1*B1*B1;

   F2 =
      A2^3 + 3*A2^2*B2 + 3*A2*B2^2 + B2^3;
```

This example shows a few things:

1. Generic functions are non-commuting. Only if we specify functions to be commuting with the CFunction declaration they will be taken as such.
2. We may define more than one expression at a time.
3. Functions do not have to have arguments.
4. The statistics show that if all objects in a power are commuting (one is allowed to be non-commuting) FORM uses binomial expansions, but for non-commuting objects that is

not possible.

5. Although internally these functions are stored in the same way, when commuting functions occur more than once FORM uses powers in the output. For non-commuting objects FORM does not use powers.

```
    Functions A1,B1;
    CFunctions A2,B2;
    Local F1 = (A1+B1+A2+B2)^3;
    Local F2 = ((A1+B1)+A2+B2)^3;
    Print;
    .end

Time =       0.00 sec    Generated terms =          64
              F1         Terms in output =          26
                         Bytes used      =        1000

Time =       0.00 sec    Generated terms =          26
              F2         Terms in output =          26
                         Bytes used      =        1000

   F1 =
      A2^3 + 3*A2^2*B2 + 3*A2*B2^2 + B2^3 + 3*A1*A2^2 + 6*A1*A2*B2 + 3*
      A1*B2^2 + 3*A1*A1*A2 + 3*A1*A1*B2 + A1*A1*A1 + A1*A1*B1 + 3*A1*B1
```

```
*A2 + 3*A1*B1*B2 + A1*B1*A1 + A1*B1*B1 + 3*B1*A2^2 + 6*B1*A2*B2
 + 3*B1*B2^2 + 3*B1*A1*A2 + 3*B1*A1*B2 + B1*A1*A1 + B1*A1*B1 + 3*
B1*B1*A2 + 3*B1*B1*B2 + B1*B1*A1 + B1*B1*B1;

F2 =
   A2^3 + 3*A2^2*B2 + 3*A2*B2^2 + B2^3 + 3*A1*A2^2 + 6*A1*A2*B2 + 3*
   A1*B2^2 + 3*A1*A1*A2 + 3*A1*A1*B2 + A1*A1*A1 + A1*A1*B1 + 3*A1*B1
   *A2 + 3*A1*B1*B2 + A1*B1*A1 + A1*B1*B1 + 3*B1*A2^2 + 6*B1*A2*B2
    + 3*B1*B2^2 + 3*B1*A1*A2 + 3*B1*A1*B2 + B1*A1*A1 + B1*A1*B1 + 3*
   B1*B1*A2 + 3*B1*B1*B2 + B1*B1*A1 + B1*B1*B1;
```

(prog10.frm) Here we see that in F1 there are two non-commuting objects inside the bracket and hence FORM is not using binomial coefficients. In F2 we have the two non-commuting objects inside an extra pair of brackets. This means that when the outer level of brackets is worked out, there is only a single non-commuting object and the binomial expansion can be used. After that the inner bracket is worked out. This results in powers of `(A1+B1)` and those are then written in all detail. The answers are idential, but it should be clear that the second one is faster.

In prog11.frm you see an example where it makes quite a difference:

```
Functions A1,B1,C1,D1;
CFunctions A2,B2,C2,D2;
Local F1 = (A1+B1+C1+D1+A2+B2+C2+D2)^7;
Local F2 = ((A1+B1+C1+D1)+A2+B2+C2+D2)^7;
.end
```

```
Time =       1.78 sec    Generated terms =    2097152
              F1         Terms in output =      51720
                         Bytes used      =    1860560

Time =       1.83 sec    Generated terms =      51720
              F2         Terms in output =      51720
                         Bytes used      =    1860560
  1.83 sec out of 1.84 sec
```

FORM does not do such grouping by itself. It does not manipulate the input. This way you can have control over the order in which the terms are generated. Sometimes this can help you make your program much faster or using less disk space.

Functions can have any number of arguments as long as the term still fits inside a predefined (and user controlable) maximum internal size.

```
*   Example of nontrivial functions (prog12.frm)
CFunction f,S,R;
Symbol x,N;
Local F = f(x)+f(x^2)+f(x,x+1)+f;
Local G = S(R(3,1,-2),N+1);
Print;
.end

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

G =
   S(R(3,1,-2),1 + N);
```

The number of arguments and what they mean is totally up to the user.

Next we have vectors and indexes. The following is a bit tricky. We attach dimensions to indexes, not to vectors! For the indexes we have also a very special function: the Kronecker delta.

Vectors have an index and when we have contracted indexes the index is removed and we write the two vectors in a dotproduct (prog13.frm).

```
Index i1,i2,i3;
Vector p1,p2,p3;
Local F = p1(i1)*(p2(i1)+p3(i3))*(p1(i2)+p2(i3));
Print;
.end
```

```
F =
   p1(i1)*p1(i2)*p3(i3) + p1(i1)*p2.p3 + p1(i2)*p1.p2
    + p2(i3)*p1.p2;
```

The Einstein summation convention is applied whenever possible. However, be careful with too many identical indexes (prog13a.frm):

```
Index i1;
Vector p1,p2,p3,p4;
Local F = p1(i1)*p3(i1)*p2(i1)*p4(i1);
Print;
.end

F =
   p1.p3*p2.p4;
```

This goes purely by the order in which FORM encounters the vectors. There are no warnings. If you do something more complicated FORM may get an different answer than what you intend.

The dimension of an index can be specified in its declaration. The default dimension is 4.

```
Symbol x,D;
Index i1=3,i2 = 4,i3=D,i4=0,i5,i6,i7;
Local F = x*d_(i1,i1)
          +x^2*d_(i2,i2)+x^3*d_(i3,i3)
          +x^4*d_(i1,i2)*d_(i2,i1)+x^5*d_(i2,i1)*d_(i1,i2)
          +x^6*d_(i5,i6)*d_(i6,i7)
          +x^7*d_(i4,i4)
          +x^8*d_(i5,i4)*d_(i4,i7);
Print +s;
.end

F =
    + 3*x
    + 4*x^2
    + x^3*D
    + 4*x^4
    + 3*x^5
```

```
 + d_(i4,i4)*x^7
 + d_(i4,i5)*d_(i4,i7)*x^8
 + d_(i5,i7)*x^6
;
```

(prog14.frm) Here we see the first built in object: d_ which is the Kronecker delta. Because it is a systems object its name is case insensitive and you could also write D_. All systems defined objects have a name with a trailing underscore character. This character is not allowed to occur in user defined objects. This avoids confusion and gives the user free choice of names. There are no reserved names in the namespace of the user!

Back to the above program. The dimension of an index is specified after its name and an equal sign. Blank spaces are allowed. The dimension can be

- A positive integer, not bigger than a FORM word.
- A symbol.
- Zero.

An index with dimension zero will not be summed over. The dimension comes only in play when we have a Kronecker delta with two identical indexes as one can see in the above program. The program has a few funny things. The x^7 term has a Kronecker delta with indexes with zero dimension. They are not summed over. Neither are they in the x^8 term. Special attention should be given to the x^4 and x^5 terms. If you mix dimensions you can get unpredictable results. It all depends on the order in which the indexes are contracted. This order is not defined. It may be changed if the internal architecture asks for it. Hence try to keep your code unambiguous.

FORM does not complain about such 'dubious' things. It is totally up to the user to interpret what they mean.

The next built in function is the Levi-Civita tensor. It is indicated by e_ or E_. It is totally antisymmetric and FORM knows how to contract pairs of Levi-Civita tensors (prog15.frm):

```
Symbol D;
Index m1=2,m2=2,m3=2,m4=2;
Index n1=3,n2=3,n3=3,n4=3,n5=3,n6=3;
Index r1=D,r2=D,r3=D,r4=D,r5=D,r6=D;
Local F1 = e_(m1,m2)*e_(m3,m4);
Local F2 = e_(m1,m2)*e_(m2,m3);
Local F3 = e_(m1,m2)*e_(m1,m2);
Local G1 = e_(n1,n2,n3)*e_(n4,n5,n6);
Local G2 = e_(n1,n2,n3)*e_(n3,n4,n5);
Local G3 = e_(n1,n2,n3)*e_(n2,n3,n4);
Local G4 = e_(n1,n2,n3)*e_(n1,n2,n3);
Local H1 = e_(r1,r2,r3)*e_(r4,r5,r6);
Local H2 = e_(r1,r2,r3)*e_(r3,r4,r5);
Local H3 = e_(r1,r2,r3)*e_(r2,r3,r4);
Local H4 = e_(r1,r2,r3)*e_(r1,r2,r3);
Contract;
```

```
 Print;
 .end
F1 = d_(m1,m3)*d_(m2,m4) - d_(m1,m4)*d_(m2,m3);
F2 = - d_(m1,m3);
F3 = 2;

G1 = d_(n1,n4)*d_(n2,n5)*d_(n3,n6) - d_(n1,n4)*d_(n2,n6)*d_(n3,n5) -
     d_(n1,n5)*d_(n2,n4)*d_(n3,n6) + d_(n1,n5)*d_(n2,n6)*d_(n3,n4) +
     d_(n1,n6)*d_(n2,n4)*d_(n3,n5) - d_(n1,n6)*d_(n2,n5)*d_(n3,n4);
G2 = d_(n1,n4)*d_(n2,n5) - d_(n1,n5)*d_(n2,n4);
G3 = 2*d_(n1,n4);
G4 = 6;

H1 = d_(r1,r4)*d_(r2,r5)*d_(r3,r6) - d_(r1,r4)*d_(r2,r6)*d_(r3,r5) -
     d_(r1,r5)*d_(r2,r4)*d_(r3,r6) + d_(r1,r5)*d_(r2,r6)*d_(r3,r4) +
     d_(r1,r6)*d_(r2,r4)*d_(r3,r5) - d_(r1,r6)*d_(r2,r5)*d_(r3,r4);
H2 = - 2*d_(r1,r4)*d_(r2,r5) + d_(r1,r4)*d_(r2,r5)*D
     + 2*d_(r1,r5)*d_(r2,r4) - d_(r1,r5)*d_(r2,r4)*D;
```

```
H3 = 2*d_(r1,r4) - 3*d_(r1,r4)*D + d_(r1,r4)*D^2;
H4 = 2*D - 3*D^2 + D^3;
```

As you see, the dimension of the indexes is not bound to be the same as the number of arguments in the LC-tensor. If there is a contracted index that has a dimension that is different from the number of arguments, FORM does not use the 'shorter' formula, but writes all n! terms with Kronecker delta's and only then contracts the indexes.

When the index of a vector is contracted with an index of a LC-tensor the vector is written in the location of the index in the LC-tensor (prog16.frm):

```
Index i1,i2,i3,i4;
Vector p1,p2;
Local F = e_(i1,i2,i3,i4)*p1(i1)*p2(i2);
Print;
.end

F =
   e_(p1,p2,i3,i4);
```

This is called Schoonschip notation. All memory of the original indexes i_1 and i_2 is erased. It is much easier to put the contracted representation in the input. This saves much work and is much easier to read:

```
Index i1,i2;
Vector p1,p2,p3,p4;
Local F = e_(i1,i2,p1,p2)*e_(i1,i2,p3,p4);
Contract;
Print;
.end

F =
   2*p1.p3*p2.p4 - 2*p1.p4*p2.p3;
```

(prog17.frm) Just try to write this out 'properly', do the contraction with the indexes and then contract them with the vectors again. That is much more complicated.

If you are worried about upper and lower indexes, the important thing is to realize that you rarely need to worry about them. Only when you write out tensors explicitly you may need to study this. This is done in an example in another course (lecture five).

The last system function we are going to see in this session is the gamma matrix g_ or G_. This 'tensor' is in principle non-commuting because we indicate spin lines as the first argument as in (prog18.frm)

```
Index mu1,mu2,mu3,mu4;
Vector p1,p2,p3,p4;
Local F1 = g_(1,mu1)*g_(1,mu2)*g_(1,mu3)*g_(1,mu4);
Local F2 = g_(2,p1)*g_(2,p2)*g_(2,p3)*g_(2,p4);
Print;
.sort

F1 =
   g_(1,mu1,mu2,mu3,mu4);

F2 =
   g_(2,p1,p2,p3,p4);

Tracen,1;
Trace4 2;
```

```
 Print;
 .end

F1 =
   4*d_(mu1,mu2)*d_(mu3,mu4) - 4*d_(mu1,mu3)*d_(mu2,mu4)
    + 4*d_(mu1,mu4)*d_(mu2,mu3);

F2 =
   4*p1.p2*p3.p4 - 4*p1.p3*p2.p4 + 4*p1.p4*p2.p3;
```

The gamma's are combined into strings. This notation comes from Reduce. Basically what we do here is

$$\gamma^{\mu_1}_{i_1 i_2}\gamma^{\mu_2}_{i_2 i_3}\gamma^{\mu_3}_{i_3 i_4}\gamma^{\mu_4}_{i_4 i_5} \;=\; \gamma^{\mu_1\mu_2\mu_3\mu_4}_{i_1 i_5}$$

and we replace the i indexes by a spinline index to say which other matrices have to be pulled in.

There are two trace commands: one is an n-dimensional trace which does not allow γ_5 and the other is a 4-dimensional trace which does allow γ_5 and applies many 4-dimensional tricks like Chisholm identities to keep the expressions short. There are some varieties of the γ_5 that were introduced by Veltman in Schoonschip:

$\gamma_5 \rightarrow$ `g5_(1)` or `g_(1,5_)`

$\gamma_6 = (1 + \gamma_5) \rightarrow$ `g6_(1)` or `g_(1,6_)`

$\gamma_7 = (1 - \gamma_5) \rightarrow$ `g7_(1)` or `g_(1,7_)`

In addition there is the unit matrix `gi_(1)`.

Let us see whether you can do the following trace by hand (prog19.frm):

```
Vector p1,p2,p3,p4,q1,q2,q3,q4;
Index i1,i2,i3,i4;
Local F = g_(1,p1,i1,7_,p2,i2,7_,p3,i3,7_,p4,i4,7_)
         *g_(2,q1,i1,7_,q2,i2,7_,q3,i3,7_,q4,i4,7_);
Trace4,1;
Trace4,2;
Print;
.end
```

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

   F =
      65536*p1.q1*p2.q2*p3.q3*p4.q4;
```

and whether you can do it the way FORM does it by generating only a single term.

The next important thing is 'pattern matching'. When we make a substitution we might need some form of wildcarding: having generic arguments (prog20.frm).

```
Symbol x,y,z,a,b;
CFunction f;
Local F = f(x)+f(y)+f(4)+f(a+b);
id f(z?) = z^2;
Print;
.end
```

```
F =
   16 + b^2 + 2*a*b + a^2 + y^2 + x^2;
```

Wildcards are indicated by placing a questionmark after the object. Wildcard symbols can match symbols, numbers or scalarlike expressions. Wildcard indexes can match indexes or vectors (making the assumption that there was an index there that contracted with the index of the vector). Wildcard vectors can match only vectors, the negative of a vector or a vectorlike expression.

If the same wildcard occurs more than once in the LHS (the pattern) it indicates that both must match the same thing (prog20a.frm).

```
Symbol x,y,z,a,b;
CFunction f;
Local F = f(x)*f(y)*f(y)*f(4)*f(a+b)*f(a+b);
id f(z?)*f(z?) = z^2*f(3,z);
Print +s;
.end

F =
    + f(3,y)*f(3,b + a)*f(4)*f(x)*y^2*b^2
    + 2*f(3,y)*f(3,b + a)*f(4)*f(x)*y^2*a*b
    + f(3,y)*f(3,b + a)*f(4)*f(x)*y^2*a^2
  ;
```

There is a special wildcard that can pick up fields of arguments. They are indicated by a questionmark, followed by a name (prog21.frm).

```
Symbol a1,a2,a3,a4,a5,a6;
CFunction f,g;
Local F = f(a1,a2,a3,a4)*f(a1,a3,a2,a1,a4,a5,a6);
id  f(?a,a3,?b) = g(?b,a3,?a);
Print +s;
.end
```

```
F =
    + g(a2,a1,a4,a5,a6,a3,a1)*g(a4,a3,a1,a2)
   ;
```

This can become rather powerful when combined with a new set of statements: the repeat loop (prog22.frm).

```
Symbol a1,a2,a3,a4,a5,a6;
CFunction f,g;
Local F = f(a1,a2,a3,a4)*f(a1,a3,a2,a1,a4,a5,a6);
repeat;
  id f(a1?,a2?,?a) = f(a1)*f(a2,?a);
endrepeat;
Print +s;
.end

F =
    + f(a1)^3*f(a2)^2*f(a3)^2*f(a4)^2*f(a5)*f(a6)
   ;
```

What is between the repeat and the endrepeat will be repeated as a block of statements as long as any of the statements in that block did anything.

It is easy to do very nice things with this, provided you have something that can stop the loop (prog26.frm):

```
Symbol n,x1,x2;
CFunction f,fib;
Local F = f(10,1,1);
repeat;
    id  f(0,?a) = fib(?a);
    id  f(n?,?a,x1?,x2?) = f(n-1,?a,x1,x2,x1+x2);
endrepeat;
Print;
.end
```

```
F =
   fib(1,1,2,3,5,8,13,21,34,55,89,144);
```

It is of course easy to make infinite loops this way (prog23.fm):

```
   Symbol x;
   Local F = x;
   id x = x+1;
   id x = x+1;
   Print;
   .sort

  F =
     2 + x;

   repeat;
     id x = x+1;
   endrepeat;
   print;
   .end
=== Workspace overflow. 4000000 bytes is not enough.
=== Change parameter WorkSpace in form.set
```

```
Program terminating at prog23.frm Line 10 -->
```

As you see, the program ends suddenly with an error message because it had a workspace overflow. Sometimes, when you do very complicated things and have many statements inside a single module, this is a problem that can be repaired by increasing the size of the workspace, but in this case it is because of an infinite loop, hence there will never be enough workspace.

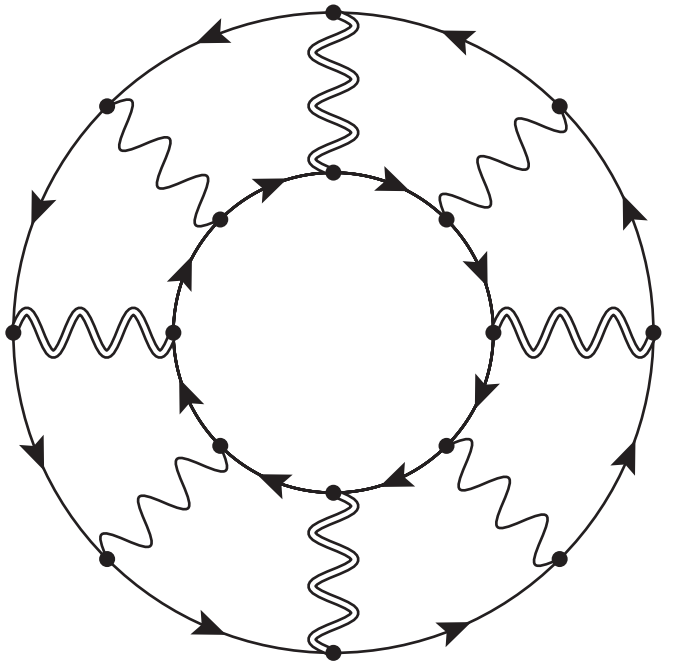
You can also see that an id-statement is ordinarily only executed once.

To show that we can also play and that function arguments can contain functions again:

```
CFunction O;    * This is prog24.frm
Symbol o;
Local F = O;
id O(?a) = O(O(?a,o),0,O(?a));
id O(?a) = O(O(?a,o),0,O(?a));
id O(?a) = O(O(?a,o),0,O(?a));
id O(?a) = O(O(?a,o),0,O(?a));
id O(?a) = O(O(?a,o),0,O(?a));
Print;
.end
```

```
F =
   O(O(O(O(O(O(o),0,O,o),0,O(O(o),0,O),o),0,O(O(O(o),0,O,o),0,O(O(o)
   ,0,O)),o),0,O(O(O(O(o),0,O,o),0,O(O(o),0,O),o),0,O(O(O(o),0,O,o),
   0,O(O(o),0,O))),o),0,O(O(O(O(O(o),0,O,o),0,O(O(o),0,O),o),0,O(O(
   O(o),0,O,o),0,O(O(o),0,O)),o),0,O(O(O(O(o),0,O,o),0,O(O(o),0,O),o
   ),0,O(O(O(o),0,O,o),0,O(O(o),0,O)))));
```

Homework:



The double wiggles are W-bosons (with V-A coupling) and the normal wiggles are photons.
Try to work out this trace in 4 dimensions (forget about the denominators).
Hint: use γ_7. Assume that photons do not change the mass of a particle, but W-bosons do.
If you use also other systems, try this also on the other systems and explain the differences.