THREE

Until now we have seen a number of statements at the algebraic level and the instructions at the preprocessor level. In this chapter we will have a look at ways to have the two levels communicate with each other.

The first way is a means to redefine the content of a preprocessor variable during algebraic execution.

Let us look at an example that will occur often during either integration or summation. Imagine we have to integrate

$$\int dx\ \frac{1}{(x+1)(x+2)\cdots(x+n)}\ =\ ????$$

for some big integer number n. The method to use is to "split the fractions" as we do in the next program:

```
    #define N "10"
    Symbol x,m1,m2;
    CFunction den;
    Local F = <den(x+1)>*...*<den(x+`N')>;
    Print "<1> %t";
    SplitArg,den;
    Print "<2> %t";
    repeat id den(m1?,x)*den(m2?,x) = (den(m1,x)-den(m2,x))/(m2-m1);
    Print +s;
    .end
<1>  + den(1 + x)*den(2 + x)*den(3 + x)*den(4 + x)*den(5 + x)*den(6
  + x)*den(7 + x)*den(8 + x)*den(9 + x)*den(10 + x)
<2>  + den(1,x)*den(2,x)*den(3,x)*den(4,x)*den(5,x)*den(6,x)*den(7,
 x)*den(8,x)*den(9,x)*den(10,x)

Time =       0.00 sec    Generated terms =        512
               F         Terms in output =         10
                         Bytes used      =        336
```

```
   F =
       + 1/362880*den(1,x)
       - 1/40320*den(2,x)
       + 1/10080*den(3,x)
       - 1/4320*den(4,x)
       + 1/2880*den(5,x)
       - 1/2880*den(6,x)
       + 1/4320*den(7,x)
       - 1/10080*den(8,x)
       + 1/40320*den(9,x)
       - 1/362880*den(10,x)
      ;
```

We see here a variety of the repeat-endrepeat construction. If there is only a single statement inside the loop, we need only

```
    repeat statement;
```

We also see a completely new type of statement: SplitArg. This statement can take the terms of a composite argument and make each of the terms into a separate argument. It is also possible to just select some of the terms, e.g. all terms that contain the symbol x. In our example it would have been more proper to use **`splitarg,(x),den;`**. This way it should split off only arguments that are a numerical multiple of x. The complete syntax is given in the manual.

Back to the output: It should be clear that at this point we can do the integral. What if n is bigger?

```
    #define N "20"
    Symbol x,m1,m2;
    CFunction den;
    Local F = <den(x+1)>*...*<den(x+`N')>;
    SplitArg,den;
    repeat id den(m1?,x)*den(m2?,x) = (den(m1,x)-den(m2,x))/(m2-m1);
    .end

Time =       2.83 sec    Generated terms =     524288
               F         Terms in output =         20
                         Bytes used      =        816
```

This is rapidly becoming unpractical. Can be place .sort instructions? Unfortunately we cannot put a .sort instruction inside a repeat-endrepeat loop, because the whole loop must be inside a single module. We could of course do the following:

```
    #define N "20"
    Symbol x,m1,m2;
    CFunction den;
    Local F = <den(x+1)>*...*<den(x+`N')>;
    SplitArg,den;
    #do i = 1,5
     id den(m1?,x)*den(m2?,x) = (den(m1,x)-den(m2,x))/(m2-m1);
     .sort

Time =       0.02 sec    Generated terms =       1024
               F         Terms in output =       1024
                         Bytes used      =      61400
    #enddo

Time =       0.28 sec    Generated terms =      32768
```

```
               F         Terms in output =       1024
                         Bytes used      =      42304

Time =      0.30 sec     Generated terms =       4096
               F         Terms in output =        256
                         Bytes used      =      10224

Time =      0.30 sec     Generated terms =        512
               F         Terms in output =         64
                         Bytes used      =       2800

Time =      0.30 sec     Generated terms =        128
               F         Terms in output =         20
                         Bytes used      =        816
    .end

Time =      0.30 sec     Generated terms =         20
               F         Terms in output =         20
```

```
                    Bytes used      =        816
```

As you see, this is much faster. In the first substitution there are 10 pairs, so we generate 2^{10} terms, etc. It can still be faster if we replace one pair at a time:

```
    #define N "20"
    Symbol x,m1,m2;
    CFunction den;
    Local F = <den(x+1)>*...*<den(x+‘N’)>;
    SplitArg,den;
    id,once,den(m1?,x)*den(m2?,x) = (den(m1,x)-den(m2,x))/(m2-m1);
    #do i = 1,‘N’-1
     id,once,den(m1?,x)*den(m2?,x) = (den(m1,x)-den(m2,x))/(m2-m1);
     .sort

Time =       0.00 sec    Generated terms =          4
                F        Terms in output =          3
                         Bytes used      =       1540
    #enddo
                    .
                    .
                    .
Time =       0.00 sec    Generated terms =         38
```

```
               F          Terms in output =         20
                          Bytes used      =        816

Time =       0.00 sec    Generated terms =         20
               F          Terms in output =         20
                          Bytes used      =        816
    .end

Time =       0.00 sec    Generated terms =         20
               F          Terms in output =         20
                          Bytes used      =        816
```

The option once in the id-statement tells FORM to make the match only once, even if the pattern fits more than once. In that case FORM takes the first match it encounters and ignores the others.

In the worst case there are 38 terms generated in a single step. But the problem is that we have to know how many times to go through the loop. Can we not let the program determine that by itself? For this we do something which in many languages is considered rather dirty:

```
#define N "20"
Symbol x,m1,m2;
CFunction den;
Local F = <den(x+1)>*...*<den(x+`N')>;
SplitArg,den;
id,once,den(m1?,x)*den(m2?,x) = (den(m1,x)-den(m2,x))/(m2-m1);
#do i = 1,1
 id,once,den(m1?,x)*den(m2?,x) = (den(m1,x)-den(m2,x))/(m2-m1);
 if ( count(den,1) > 1 );
   redefine i "0";
 endif;
 .sort
#enddo
.end
```

We see two new things here. The easy one is the redefine statement which overwrites the definition of the preprocessor variable i. Note that if we use the value of i inside the loop, it will have the old value until the .sort instruction, because until then no execution has taken place. After the .sort it has executed and it may have changed its value. If this value has become zero, the #enddo will raise the numerical value of i and see that it is inside the range of the loop and hence go through the loop again.

The other new feature is the if statement. We were still missing that one. It is a bit like the if in any other langauge and like the #if in the preprocessor, except that it acts on terms during the execution.

```
    if ( condition );
    elseif ( condition2 );
    else;
    endif;
```

and there is the variety when only a single statement is involved:

```
    if ( condition ) statement;
```

The question is now: what conditions do we have? Clearly not something as in calculational languages or in the preprocessor. The condition we see here is a form of power or occurrence counting:

```
    if ( count(den,1) > 1 );
```

means that counting the number of occurrences of den and giving each a weight of one, take the if when the resulting weight is more than one. This can become more complicated when more objects are involved as in

```
    if ( count(den,3,x,1) > 10 );
```

This would mean that the term `den(x+1)*den(x+3)*x^7` would have weight 13: two times 3 from the den functions and 7 times one from the x^7. Weights can also be negative. This count object is most frequent in if statements. One can put functions, symbols, vectors and dotproducts inside.

Now back to our program. The statements

```
if ( count(den,1) > 1 );
  redefine i "0";
endif;
.sort
```

force the loop to be repeated as long as there is still a term with more than one occurrence of the den function. Note that the .sort is essential, because i can only obtain the redefined value during execution.

Next we make the program a bit more complicated. We will define F with some powers of the denominators. Unfortunately our program will crash:

```
    #define N "6"
    Symbol x,m1,m2;
    CFunction den;
    Local F = <den(x+1)^1>*...*<den(x+`N')^`N'>;
    SplitArg,den;
    id,once,den(m1?,x)*den(m2?,x) = (den(m1,x)-den(m2,x))/(m2-m1);
    #do i = 1,1
     id,once,den(m1?,x)*den(m2?,x) = (den(m1,x)-den(m2,x))/(m2-m1);
     if ( count(den,1) > 1 );
       redefine i "0";
     endif;
     .sort
Division by zero during normalization
    #enddo
    .end
Program terminating at p3-6.frm Line 6 -->
```

Close inspection shows the following statement to be the culprit.

```
   id,once,den(m1?,x)*den(m2?,x) = (den(m1,x)-den(m2,x))/(m2-m1);
```

will give division by zero. This can only be avoided if we can force the wildcard variables `m1` and `m2` to be unequal. There are many ways by which this can be done, but the fastest and most direct way is with the statement

```
   id,once,den(m1?!{m2?},x)*den(m2?!{m1?},x) =
                                      (den(m1,x)-den(m2,x))/(m2-m1);
```

This looks a bit cryptic. Let us study this more closely.

When we have a wildcard in a substitution, it may be needed to restrict the values that the wildcard can take to a given set. Hence we can define sets in FORM:

```
Symbols a,b,c,d,x;
CFunction den;
Set ss:a,b;
Local F = den(a)+den(b)+den(c)+den(d);
id  den(x?ss) = den(6,x);
Print;
.end
```

```
F =
   den(c) + den(d) + den(6,a) + den(6,b);
```

We declare a set that contains two symbols a and b. By placing the set after the question mark of the wildcard we say that x can only match if it becomes a member of the indicated set. Similarly we can tell that it should not be a member of the set:

```
Symbols a,b,c,d,x;
CFunction den;
Set ss:a,b;
Local F = den(a)+den(b)+den(c)+den(d);
id  den(x?!ss) = den(6,x);
Print;
.end

F =
   den(a) + den(b) + den(6,c) + den(6,d);
```

There can be sets of symbols, sets of indexes, sets of vectors, sets of functions and sets of numbers. Numbers (integers only) can be mixed with symbols. One may also indicate sets dynamically. This is a notation that was added at a later stage.

```
    Symbols a,b,c,d,x;
    CFunction den;
    Local F = den(a)+den(b)+den(c)+den(d);
    id  den(x?{a,b}) = den(6,x);
    Print;
    .end

   F =
      den(c) + den(d) + den(6,a) + den(6,b);
```

The set should be enclosed in a $\{\}$ pair. If one uses a set of numbers as in $\{1, 2, 3\}$ there is no possibility of confusion with the preprocessor calculator, but $\{1\}$ would be 'stolen' by the preprocessor calculator and replaced by 1. Hence when there is a single number in the set one should place an extra comma to prevent such 'stealing' as in $\{, 1\}$.

The dynamic sets can also have conditions like $\{>3\}$, etc. In addition there are predefined sets that the user cannot define as in symbol_, number_, int_ (all integers), even_, odd_ and a few more. The pattern we were using in our example however was even more flexible:

```
    id,once,den(m1?!{m2?},x)*den(m2?!{m1?},x) =
```

Here we say that `m1` is not allowed to belong to the set that contains the wildcard variable `m2` et vise versa. Why do we need to specify both? Would one not be enough? We do not know in what order FORM does the matching. When `m2` has not been fixed yet the `!{m2?}` condition can only be ignored and hence the restriction on `m2` has to do the job. If somehow FORM would try to match `m2` first we need the restriction on `m1`. Hence both are needed.

Our program needs one more modification:

```
    if ( count(den,1) > 1 );
```

is not going to work when there are identical denominators. Hence we use another condition:

```
    if ( match(den(m1?!{m2?},x)*den(m2?!{m1?},x)) );
```

which says that if there is still a match, we will go through the loop again.

```
    #define N "4"
    Symbol x,m1,m2;
    CFunction den;
    Local F = <den(x+1)^1>*...*<den(x+`N')^`N'>;
    SplitArg,den;
    id,once,den(m1?!{m2?},x)*den(m2?!{m1?},x) =
             (den(m1,x)-den(m2,x))/(m2-m1);
    #do i = 1,1
     id,once,den(m1?!{m2?},x)*den(m2?!{m1?},x) =
             (den(m1,x)-den(m2,x))/(m2-m1);
     if ( match(den(m1?!{m2?},x)*den(m2?!{m1?},x)) );
       redefine i "0";
     endif;
     .sort

Time =       0.00 sec    Generated terms =          4
               F         Terms in output =          2
                         Bytes used      =        472
    #enddo

    Print +s;
    .end

Time =       0.00 sec    Generated terms =         10
               F         Terms in output =         10
                         Bytes used      =        432

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

```
  + 13/36*den(4,x)^3
  + 1/12*den(4,x)^4
 ;
```

If speed is at an absolute premium it is possible to improve this program a little bit more.

```
#define N "20"
Symbol x,m1,m2;
CFunction den;
Off Statistics;
Local F = <den(x+1)^1>*...*<den(x+`N')^`N'>;
SplitArg,den;
id,once,den(m1?!{m2?},x)*den(m2?!{m1?},x) =
         (den(m1,x)-den(m2,x))/(m2-m1);
#do i = 1,1
 id,once,den(m1?!{m2?},x)*den(m2?!{m1?},x) =
         (den(m1,x)-den(m2,x))/(m2-m1);
 if ( match(den(m1?!{m2?},x)*den(m2?!{m1?},x)) );
   redefine i "0";
 endif;
 .sort
#enddo
On Statistics;
```

```
    .end

Time =       4.39 sec    Generated terms =        210
               F         Terms in output =        210
                         Bytes used      =      37296
```

Actually, in the match condition FORM determines how many times the pattern matches. This is overkil and we could replace the if statement by

```
    if ( match(once,den(m1?!{m2?},x)*den(m2?!{m1?},x)) );
```

The match is like the pattern matcher of an id statement without the id keyword. Hence it accepts all options of the id statement. With this the statistics become:

```
Time =          0.82 sec    Generated terms =          210
                  F         Terms in output =          210
                            Bytes used      =        37296
```

If we need even more speed, we should observe that each time FORM has to do the patttern matching twice. It does it first in the if statement and then, when it repeats the loop it does it again in the id statement.

If we change the id statement by

```
#define N "20"
Symbol x,m1,m2;
CFunction den;
Off Statistics;
Local F = <den(x+1)^1>*...*<den(x+`N')^`N'>;
SplitArg,den;
id,once,den(m1?!{m2?},x)*den(m2?!{m1?},x) =
         (den(m1,x)-den(m2,x))/(m2-m1);
#do i = 1,1
 id,ifmatch->1,once,den(m1?!{m2?},x)*den(m2?!{m1?},x) =
         (den(m1,x)-den(m2,x))/(m2-m1);
 goto 2;
 label 1;
 redefine i "0";
```

```
     label 2;
     .sort
    #enddo
    On Statistics;
    .end

Time =       0.68 sec    Generated terms =         210
               F         Terms in output =         210
                         Bytes used      =       37296
```

execution is even faster because we do the pattern matching only once but we go through the loop one time too often. This is at the cost of having to use a goto statement and placing some labels. The ifmatch->label option of the id statement makes the control jump to that label if a match has occurred and the substitution has been made. It is a somewhat complicated construction and should only be used if the gain in time is very relevant.

Now we go to a whole new level of control.

Imagine we want to know the maximum power of a variable in a complete expression. This is a problem, because we can only measure this power during execution, but during execution expressions 'do not exist'. We have to be able to store information during execution so that it may be used during compilation. And the other way around. For this we have a third type of variables: the $-variables. $-variables are actually some kind of small expressions that are kept in memory and that can be set at nearly any moment.

```
    Symbols x,y;
    Local F = (x+1)^10-(x+3)^6*(x-2)^4;
    .sort

    #$maxx = 0;
    if ( count(x,1) > $maxx ) $maxx = count_(x,1);
    .sort
    #write "The maximum power of x is ‘$maxx’"
The maximum power of x is 8
    .end
```

The $-variable can be set, either in the preprocessor during compilation, or during execution in which case each term can influence it. Hence

```
#$maxx = 0;
```

sets the value during compilation and acts to initialize the variable. Had we written

```
$maxx = 0;
```

then for each term that passes during execution the variable would be set to zero. That is not what we want in this example!

We can use the $-variable in an if-statement in which case we compare it with the weight of the count condition. We also have a corresponding count_ function and hence if the weight of the term is bigger we can replace $maxx by the new weight. After the .sort the new value is complete and we can use it in the compiler/preprocessor. There it can be used as if it is a preprocessor variable: '$maxx'. It means that we can also use it in the parameter field of a #do loop. Or as the argument in a procedure call. The writing can be done in two ways. One is the way we saw before. The other is

```
  Symbols x,y;
  Local F = (x+1)^10-(x+3)^6*(x-2)^4;
  .sort

  #$maxx = 0;
  if ( count(x,1) > $maxx );
     $maxx = count_(x,1);
     print "  $maxx adjusted to %$",$maxx;
  endif;
  .sort
$maxx adjusted to 1
$maxx adjusted to 2
$maxx adjusted to 3
$maxx adjusted to 4
$maxx adjusted to 5
$maxx adjusted to 6
$maxx adjusted to 7
$maxx adjusted to 8
```

```
    #write "The maximum power of x is %$",$maxx
The maximum power of x is 8
    .end
```

In the format string we can use the sequence `%$`. This will cause FORM to read after the format string for the next object. If this is a $-variable its contents will be printed.

$-variables are algebraic objects that can be translated into strings when the preprocessor wants to use them.

The $-variables can contain small expressions. One way to give these variables a value is directly as an expression.

```
    Symbols x,y;
    #$ex1 = (x+y)^2;
    #write "We start with %$",$ex1
We start with y^2+2*x*y+x^2
    #$ex2 = $ex1*x+1;
    #write "Next we have %$",$ex2
Next we have 1+x*y^2+2*x^2*y+x^3
    .end
```

Now before you become very enthousiastic, thinking that this way you can use FORM in the same way you would use Mathematica, Maple or Reduce by building up one expression after another, you should realize that \$-variables are living in allocated memory and hence become rather inefficient when they become big. Their internal manipulations are less efficient than those of expressions. They are meant for little things.

A completely different way to give them a value is by means of the wildcarding system.

```
  Symbols a1,...,a4;
  CFunction f,g;
  Local F = f(a1,a3,a4,a2)*g(a3,a4,a2,a1);
  Print;
  .sort

 F =
    f(a1,a3,a4,a2)*g(a3,a4,a2,a1);

  if ( match(f?$fun(?a,a1?$arg,?b)) );
   print "   --- The program took $fun = %$ and $arg = %$",$fun,$arg;
  endif;
  .end
 --- The program took $fun = f and $arg = a2
```

By attaching the \$-variable to the wildcard, it will get the same value at the one given to the wildcard. It will keep this value till the next term comes along. Now what happens if there is no match and we still try to print the \$-variable?

```
 Symbols a1,...,a4;
 CFunction f,g;
 Local F = f(a1,a3,a4,a2)*g(a3,a4,a2,a1)+2;
 Print;
 .sort

F =
   2 + f(a1,a3,a4,a2)*g(a3,a4,a2,a1);

 if ( match(f?$fun(?a,a1?$arg,?b)) );
  print "   --- The program took $fun = %$ and $arg = %$",$fun,$arg;
 endif;
  print "   +++ The program took $fun = %$ and $arg = %$",$fun,$arg;
 .end
+++ The program took $fun = *** and $arg = ***
--- The program took $fun = f and $arg = a2
+++ The program took $fun = f and $arg = a2
```

Now there are two terms and the first term does not match. This means that no value is assigned to the $-variables and using them would give some form of 'undefined'. If the constant term would be the second term we obtain

```
 Symbols a1,...,a4;
 CFunction f,g;
 On HighFirst;
 Local F = f(a1,a3,a4,a2)*g(a3,a4,a2,a1)+2;
 Print;
 .sort

F =
   f(a1,a3,a4,a2)*g(a3,a4,a2,a1) + 2;

 if ( match(f?$fun(?a,a1?$arg,?b)) );
  print "   --- The program took $fun = %$ and $arg = %$",$fun,$arg;
 endif;
  print "   +++ The program took $fun = %$ and $arg = %$",$fun,$arg;
 .end
```

```
--- The program took $fun = f and $arg = a2
+++ The program took $fun = f and $arg = a2
+++ The program took $fun = f and $arg = a2
```

and we see that now the $-variables are defined, because the previous term defined them. The term that did not give a match does not overwrite or invalidate the previous value.

One can use this also to collect values in \$-variables so that they may be used globally.

```
 Symbols a1,...,a4;
 CFunction f,g;
 Local F = f(a1,a3,a4,a2);
 Print;
 .sort

F =
   f(a1,a3,a4,a2);

 $num = 0;
 repeat;
   if ( match(f(a1?,?a)) );
     $num = $num+1;
     id f(a1?,?a) = g($num,a1)*f(?a);
   endif;
 endrepeat;
 id f = 1;
 Print;
 .sort

F =
   g(1,a1)*g(2,a3)*g(3,a4)*g(4,a2);

 #do i = 1,`$num'
  id g(`i',a1?$arg`i') = 1;
 #enddo
 Print;
 .sort

F =
   1;

 Drop F;
 #do i = 1,`$num'
  #write "       Argument `i' = `$arg`i''"
```

```
      Argument 1 = a1
   #enddo
      Argument 2 = a3
      Argument 3 = a4
      Argument 4 = a2
   .end
```

Here you see that we first have to count the arguments. There are various ways of doing this. Below we give a second way. The idea in this example is to prepare the terms for putting the various arguments in the $arg‘i’ variables. This is not something we can do at running time (we cannot modify names of variables during execution). It has to be set up by the preprocessor. The drop statement tells the program to forget about all expressions after the current module, and to not treat them in the current module. If we mention one or more expressions this holds only for those expressions.

The other way to obtain our arguments is

```
 Symbols a1,...,a4,n;
 CFunction f,g;
 Off Statistics;
 Local F = f(a1,a3,a4,a2);
 Print;
 .sort

F =
   f(a1,a3,a4,a2);

 id f(?a) = f(nargs_(?a),?a);
 Print;
 .sort

F =
   f(4,a1,a3,a4,a2);
 id f(n?$num,?a) = f(?a);
 Print;
 .sort

F =
   f(a1,a3,a4,a2);

 id f(<a1?$arg1>,...,<a`$num'?$arg`$num'>) = 1;
 Print;
 .sort

F =
   1;

 Drop F;
 #do i = 1,`$num'
  #write "       Argument `i' = `$arg`i''"
    Argument 1 = a1
 #enddo
    Argument 2 = a3
    Argument 3 = a4
    Argument 4 = a2
 .end
```

Here we use the nargs_ function which returns a number that indicates the number of arguments it had. This program is much more compact.

Next I am going to show the ‘Sinterklaas’ program. The problem: There is a set of N people who all have to give one present to one random other person from the set. Nobody is to give a present to him- or herself. There may be other vetoes. Nobody is to know from whom their present is coming. In Spanish this is called ‘amigo invisible’. Hence the selection must be secret and people should be notified by personal email. Of course it is not very difficult to write a C program to do this, but in FORM it is even easier. Let us start with the basic program:

```
#define PEOPLE "10"
#define name1 "John"
#define name2 "Jane"
#define name3 "Carol"
#define name4 "Pete"
#define name5 "Fred"
#define name6 "Lizzy"
#define name7 "Mary"
#define name8 "Frank"
#define name9 "Bill"
#define name10 "Astrid"
Symbol x;
CFunction f;
#do ii = 1,1
    Local F = ranperm_(f,1,...,`PEOPLE');
    #do i = 1,`PEOPLE'
        id f(x?$to`i',?a) = deltap_(`i'-x)*f(?a);
    #enddo
    .sort
    #if ( termsin(F) == 0 )
        #redefine ii "0"
    #endif
#enddo
#do i = 1,‘PEOPLE’
    #write <> "      ‘name‘i’’ gives to ‘name‘$to‘i’’’"
  John gives to Astrid
#enddo
  Jane gives to John
  Carol gives to Pete
  Pete gives to Carol
  Fred gives to Frank
  Lizzy gives to Fred
  Mary gives to Lizzy
  Frank gives to Jane
  Bill gives to Mary
  Astrid gives to Bill
```

Any such program starts with defining the persons involved. In this case we have 10 people and we select their names in numbered preprocessor variables. This allows us to use them as an array.

FORM is equipped with a random number function random_ which takes a positive integer argument and then generates a random number in the range 1 to that number inclusive. Hence a die roll can be emulated with random_(6). There is also the ranperm_ function which generates a random permutation of its arguments, except for the first argument which must be a function. The random permutation of the arguments will be placed as arguments of this function. Hence `ranperm_(f,1,2,3,4)` could result in `f(3,1,4,2)`.

The problem of people not having to give a present to themselves is solved by the i-th argument not being allowed to be equal to i. We take out the arguments one by one, putting the i-th value in the $-variable $to‘i’, and multiplying the expression by the function deltap_(‘i’-x). FORM has two special functions delta_. theta_ which act on numerical arguments in the intuitive way: delta_(x) is one when x is zero and zero otherwise. Theta_(x) is one when $x \geq 0$ and zero otherwise. The functions deltap_ and thetap_ are variations: deltap_(x) is zero when x is zero and one otherwise. Thetap_(x) is one when $x > 0$ and zero otherwise. Here we use deltap_ to veto a person drawing her- or himself. Next we stay in the loop as long as the permutation gets vetoed.

Then we print the output. Note the nice nesting of the backquote-quote pairs.

This was the basic program. Now we have to make it a bit better. Imagine you have more vetoes. For example a thesis advisor is not allowed to draw his- or her student and the student is not allowed to draw her- or his advisor. Similarly with couples. We could add such vetoes as individual id statements but that is messy. For this we are going to use a table.

A table is a special function with an index field and possibly more arguments. For instance we can declare

```
    Table tab(1:10,1:10,x?);
    Table easytab(1:3,-2:4);
```

In this case the first table will have two indexes and one extra argument. The first table has two indexes each running from 1 to 10. The second has two indexes of which the first runs from 1 to 3 and the second from -2 to 4. The first table has an extra argument which is a wildcard. This means that it will have to be matched. We define the table elements with the Fill statement (here the extra arguments are taken for granted):

```
    Fill tab(1,1) = x;
    Fill tab(1,2) = x^2;
    Fill tab(2,1) = x+1;
    Fill tab(3,1) = x^2+tab(1,1,2*x);
```

etc. We need to define 100 elements here. The use will be that whenever FORM encounters an element of the table it will try to make a match. For instance `tab(1,1,a)` will be replaced by `a` and `tab(1,2,b+c)` will be replaced by `b^2+2*b*c+c^2`. This is basically automatic knowledge. We will see more examples of tables. The extra arguments are not necessary and in our case can be omitted. In our program we define a 10x10 table and there is a shortcut for filling many table elements at the same time:

```
    Fill Allowed(1,1) =
          0,0,1,1,1,1,1,1,1,1
         ,0,0,1,1,1,1,1,1,1,1
         ,1,1,0,0,1,1,1,1,1,1
         ,1,1,0,0,1,1,1,1,1,1
         ,1,1,1,1,0,0,1,1,1,1
         ,1,1,1,1,0,0,1,1,1,1
         ,1,1,1,1,1,1,0,0,0,0
         ,1,1,1,1,1,1,0,0,0,0
         ,1,1,1,1,1,1,0,0,0,0
         ,1,1,1,1,1,1,0,0,0,0;
```

The elements are separated by comma's and the order is (1,1),(1,2),(1,3), ...,(1,10),(2,1),...

In this case we do not need the function deltap_, because we let the table do the vetoing. The complete program is

```
#define PEOPLE "10"
*
*   First we need names
*
#define name1 "John"
#define name2 "Jane"
#define name3 "Carol"
#define name4 "Pete"
#define name5 "Fred"
#define name6 "Lizzy"
#define name7 "Mary"
#define name8 "Frank"
#define name9 "Bill"
#define name10 "Astrid"
*
*   Then we need e-mail addresses.
```

```
*
#define email1 "John@xyz.zz"
#define email2 "Jane@xyz.zz"
#define email3 "Carol@xyz.zz"
#define email4 "Pete@xyz.zz"
#define email5 "Fred@xyz.zz"
#define email6 "Lizzy@xyz.zz"
#define email7 "Mary@xyz.zz"
#define email8 "Frank@xyz.zz"
#define email9 "Bill@xyz.zz"
#define email10 "Astrid@xyz.zz"
*
*   Next we need to set the random number generator
*   Run program with: form -d INI=somenumber friend2
*
#ifdef `INI'
  #setrandom `INI'
#else
```

```
  #setrandom 13
#endif
*
Symbol x;
CFunction f;
CTable Allowed(1:‘PEOPLE’,1:‘PEOPLE’);
Fill Allowed(1,1) =
     0,0,1,1,1,1,1,1,1,1
    ,0,0,1,1,1,1,1,1,1,1
    ,1,1,0,0,1,1,1,1,1,1
    ,1,1,0,0,1,1,1,1,1,1
    ,1,1,1,1,0,0,1,1,1,1
    ,1,1,1,1,0,0,1,1,1,1
    ,1,1,1,1,1,1,0,0,0,0
    ,1,1,1,1,1,1,0,0,0,0
    ,1,1,1,1,1,1,0,0,0,0
    ,1,1,1,1,1,1,0,0,0,0;
Off Statistics;
```

```
    #do ii = 1,1
        Local F = ranperm_(f,1,...,‘PEOPLE’);
        #do i = 1,‘PEOPLE’
            id f(x?$to‘i’,?a) = Allowed(‘i’,x)*f(?a);
        #enddo
        .sort
        #if ( termsin(F) == 0 )
            #redefine ii "0"
        #endif
    #enddo
    #do i = 1,‘PEOPLE’
        #write <‘name‘i’’.sint> "This mail contains the name of \
the person to whom"
        #write <‘name‘i’’.sint> "you should give the secret present."
        #write <‘name‘i’’.sint> "Keep this name secret."
        #write <‘name‘i’’.sint> "It is part of the mystery of \
Sinterklaas."
        #write <‘name‘i’’.sint> " "
        #write <‘name‘i’’.sint> " ‘name‘i’’ gives to ‘name‘$to‘i’’’ "
        #write <‘name‘i’’.sint> " "
        #close <‘name‘i’’.sint>
*        #system mail -s "Sinterklaas" ‘email‘i’’ < name‘i’.sint
        #remove <‘name‘i’’.sint>
    #enddo
    .end
```

It shows the new instructions #close and #remove which close the indicated file and remove it from the system. The #system instruction executes a shell command. In this case it sends a mail to the proper person. The removal of the files makes that the sender cannot spy on who gets what.

The final touch is the setting of the random number generator. This is done with the #setrandom instruction which is followed by a number. We arrange it to be such that somebody types the FORM command with some random initialization number. If you do not look at what you type and it has a lot of digits and afterwards you type 'clear' to clear the screen and close the terminal window, you have almost erased all traces (how about the 'sent' part of your mailbox?). On the other hand, there are always people who loose mails, so it may be a good idea to have something floating around in the system that is hard to get.

A warning: when you are practicing with this program, please comment out the #system instruction, because otherwise you may be spamming the people on your list with lots of names of people that they should give presents to. Enable the #system command only in the single final run.

Finally (if we still have time): What is the chance that the drawing of the championsleague eights finals in December 2012 gave the same results as the practise run? This is nontrivial statistics!

The problem: There are 8 group winners and 8 numbers two. We want a draw in which each pairing has a number one and a number two, but not from the same group, and they are also not allowed to be from the same country.

Again it is not very complicated to this in a C program, but in FORM it is even easier.

We start with generating all possible pairings. There are 8! ways to couple the numbers one to the numbers 2, if we ignore the order of the pairings. An easy way is to generate such pairings is by contracting two Levi-Civita tensors with 8 indexes. After that we veto whatever is not allowed:

```
Tensor f;
Index  i1,...,i8;
Local  F = f(i1,...,i8)*e_(i1,...,i8)*e_(1,...,8);
Contract;
*  A    PSG       Porto
*  B    Schalke   Arsenal
*  C    Malaga    Milan
*  D    Dortmund  Real Madrid
*  E    Juventus  Donetsk
*  F    Bayern    Valencia
*  G    Barcelona Celtic
*  H    Mancester Galatasaray
id f(1,?a) = 0;
id f(i1?,2,?a) = 0;
id f(i1?,i2?,i3?{3,4,6},?a) = 0;
id f(i1?,i2?,i3?,4,?a) = 0;
id f(i1?,i2?,i3?,i4?,i5?{3,5},?a) = 0;
id f(i1?,i2?,i3?,i4?,i5?,6,?a) = 0;
```

```
    id f(i1?,i2?,i3?,i4?,i5?,i6?,i7?{4,6,7},?a) = 0;
    id f(i1?,i2?,i3?,i4?,i5?,i6?,i7?,i8?{2,8}) = 0;
    .end

Time =       0.07 sec    Generated terms =       5463
               F         Terms in output =       5463
                         Bytes used      =     184456
```

The number of the argument in the function f will indicate which group winner, and its value which number 2. The id statements veto effectively what is not allowed. Of the 40320 possibilities there are 5463 left, which means that the chance to get identical pairings is 1/5463.