# Contents

# Chapter 1

# The preprocessor

The preprocessor is a program segment that reads and edits the input, after which the processed input is offered to the compiler part of FORM. When a module instruction is encountered by the preprocessor, the compilation is halted and the module is executed. The compiler buffers are cleared and FORM will continue with the next module. The preprocessor acts almost purely on character strings. As such it does not know about the algebraic properties of the objects it processes. Additionally the preprocessor also filters out the commentary.

The commands for the preprocessor are called instructions. Preprocessor instructions start with the character # as the first non-blank character in a line. After this there are several possibilities.

**#:** Special syntax for setup parameters at the beginning of the program. See the chapter on the setup parameters.

**#-, #+** Turns the listing of the input off or on.

**#name** Preprocessor command. The syntax of the various commands will be discussed below.

**#$name** Giving a value to a dollar variable in the preprocessor. See the chapter on dollar variables.

## 1.1 The preprocessor variables

In order to help in the edit function the preprocessor is equipped with variables that can be defined or redefined by the user or by other preprocessor actions. Preprocessor variables have regular names that are composed of strings of alphanumeric characters of which the first one must be alphabetic. When they are defined one just uses this name. When they are used the name should be enclosed between a backquote and a quote as if these were some type of brackets. Hence `a2r' is the reference to a regular preprocessor variable. Preprocessor variables contain strings of characters. No interpretation is given to these strings. The backquote/quote pairs can be nested. Hence `a`i'r' will result in the preprocessor variable `i' to be substituted first. If this happens to be the string "2", the result after the first substitution would be `a2r' and then FORM would look for its string value.

The use of the backquotes is different from the earlier versions of FORM. There the preprocessor variables would be enclosed in a pair of quotes and no nesting was possible. FORM still understands this old notation because it does not lead to ambiguities. The user is however strongly advised to use the new notation with the backquotes, because in future versions the old notation may not be recognized any longer.

FORM has a number of built in preprocessor variables. They are:

| | |
|---|---|
| VERSION_ | The current version as the 3 in 3.1. |
| SUBVERSION_ | The sub-version as the 1 in 3.1. |
| NAMEVERSION_ | For special versions that are experimental. Otherwise empty. |
| NAME_ | The name of the program file. |
| DATE_ | The date of the current run. |
| CMODULE_ | The number of the current module. |
| SHOWINPUT_ | If input listing is on: 1, if off: 0. |

## 1.2 The preprocessor calculator

Sometimes a preprocessor variable should be interpreted as a number and some arithmetic should be done with it. For this FORM is equiped with what is called the preprocessor calculator. When the input reading device encounters a left curly bracket `{`, it will read till the matching right curly bracket `}` and then test whether the characters (after substitution of preprocessor variables) can be interpreted as a numerical expression. If it is not a valid numerical expression the whole string, including the curly brackets, will be passed on to the later stages of the program. If it is a numerical expression, it will be evaluated, and the whole string, including the curly brackets, will be replaced by a textual representation of the result. Example:

```
Local F`i' = F{`i'-1}+F{`i'-2};
```

If the preprocessor variable i has the value 11, the calculator makes this into

```
Local F11 = F10+F9;
```

Valid numerical expressions can contain the characters

```
0 1 2 3 4 5 6 7 8 9 + - * / % ( ) { } & | ^ !
```

The use of parentheses is as in regular arithmetic. The curly brackets fullfil the same role, as one can nest these brackets of course. Operators are:

+ Regular addition.

− Regular subtraction.

∗ Regular multiplication.

/ Regular (integer) division.

% The remainder after (integer) division as in the language C.

& And operator. This is a bitwise operator.

| Or operator. This is a bitwise or.

∧ Exponent operator.

! Factorial. This is a postfix operator.

∧% A postfix $^2$log. This means that it takes the $^2$log of the object to the left of it.

∧/ A postfix square root. This means that it takes the square root of the object to the left of it.

Note that all arithmetic is done over the integers and that there is a finite range. On 32 bit systems this range will be $2^{31}-1$ to $-2^{31}$, while on 64 bit systems this will be $2^{63}-1$ to $-2^{63}$. In particular this means that `{13^/}` becomes `3`. The preprocessor calculator is only meant for some simple counting and organization of the program flow. Hence there is no large degree of sophistication. Very important is that the comma character is not a legal character for the preprocessor calculator. This can be used to avoid some problems. Suppose one needs to make a substitution of the type:

```
id f(x?!{0}) = 1/x;
```

in which the value zero should be excluded from the pattern matching. This would not work, because the preprocessor would make this into

```
id f(x?!0) = 1/x;
```

which is illegal syntax. Hence the proper trick is to write

```
id f(x?!{,0}) = 1/x;
```

With the comma the preprocessor will leave this untouched, and hence now the set is passed properly.

Good use of the preprocessor calculator can make life much easier for FORM. For example the following statements

```
id  f(`i') = 1/(`i'+1);
id  f(`i') = 1/{`i'+1};
```

are quite different in nature. In the first statement the compiler gets an expression with a composite denominator. The compiler never tries to simplify expressions by doing algebra on them. Sometimes this may not be optimal, but there are cases in which it would cause wrong results (in particular when noncommuting and commuting functions are mixed and wildcards are used). Hence the composite denominator has to be worked out during run time for each term separately. The second statement has the preprocessor work out the sum and hence the compiler gets a simple fraction and less time will be needed during running. Note that

```
id  f(`i') = {1/(`i'+1)};
```

would most likely not produce the desired result, because the preprocessor calculator works only over the integers. Hence, unless i is equal to zero or -2, the result would be zero (excluding of course the fatal error when i is equal to -1).

## 1.3 The triple dot operator

The last stage of the actions of the preprocessor involves the triple dot operator. It indicates a repeated pattern as in `a1+...+a4` which would expand into `a1+a2+a3+a4`. This operator is used in two different ways. First the most general way:

```
<pattern1>operator1...operator2<pattern2>
```

in which the less than and greater than signs serve as boundaries for the patterns. The operators can be any pair of the following:

+ + Repetitions will be separated by plus signs.

**− −** Repetitions will be separated by minus signs.

**+ −** Repetitions will be separated by alternating signs. First will be plus.

**− +** Repetitions will be separated by alternating signs. First will be minus.

**∗ ∗** Repetitions will be separated by ∗.

**/ /** Repetitions will be separated by /.

**, ,** Repetitions will be separated by comma's.

**: :** Repetitions will be separated by *single* dots.

For such a pair of operators FORM will inspect the patterns and see whether the differences between the two patterns are just numbers. If the differences are numbers and the absolute value of the difference of each matching pair is always the same (a difference of zero is allowed too; it leads to no action for the pair), then FORM will expand the pattern, running from the first to the last in increments of one. For each pair the counter can either run up or run down, depending on whether the number in the first pattern is greater or less than the number in the second pattern. Example:

```
Local F = <a1b6(c3)>-...+<a4b3(c6)>;
```

leads to

```
Local F = a1b6(c3)-a2b5(c4)+a3b4(c5)-a4b3(c6);
```

The second form is a bit simpler. It recognizes that there are special cases that can be written in a more intuitive way. If there is only a single number to be varied, and it is the end of the pattern, and the rest of the patterns consists only of alphanumeric characters of which the first is an alphabetic character, we do not need the less than/greater than combination. This is shown in

```
Symbol a1,...,a12;
```

There is one extra exception. The variables used this way may have a question mark after them to indicate that they are wildcards:

```
id  f(a1?,...,a4?) = g(a1,...,a4,a1+...+a4);
```

This construction did not exist in earlier versions of FORM. There one needed the #do instruction for many of the above constructions, creating code that was very hard to read. The ... operator should improve the readability of the programs very much.

## 1.4 #append

Syntax:
    #append <filename>
See also write (1.36), close (1.8), create (1.10, remove (1.30)
Opens the named file for writing. The file will be positioned at the end. The next #write instruction will add to it.

## 1.5 #break

Syntax:
 #break
See also switch (1.32), endswitch (1.19), case (1.7), default (1.11)
If the lines before were not part of the control flow (*i.e.* these lines are used for the later stages of the program), this instruction is ignored. If they are part of the control flow, the flow will continue after the matching #endswitch instruction. The #break instruction must of course be inside the range of a #switch/#endswitch construction.

## 1.6 #call

Syntax:
 #call procname(var1,...,varn)
See also procedure (1.28), endprocedure (1.18)
This instruction calls the procedure with the name procname. The result is that FORM looks for this procedure, first in its procedure buffers (for procedures that were defined in the regular text stream as explained under the #procedure instruction), then it looks for a file by the name procname.prc in the current directory, and if it still has not found the procedure it looks in the directories indicated by the -p option in the command that started FORM (see the chapter on running FORM). If this -p option has not been used FORM will see whether there is an environment variable by the name FORMPATH. The directories indicated there will be searched for the file procname.prc. If FORM cannot find the file, there will be an error message and execution will be stopped immediately.

Once the procedure has been located, FORM reads the whole file and then determines whether the number of parameters is identical in the #call instruction and the #procedure instruction. A difference is a fatal error.

The parameter field consists of strings, separated by commas. If a string contains a comma, this comma should be preceded by a backslash character (\). If a string should contain a linefeed, one should 'escape' this linefeed by putting a backslash and continue on the next line.

In old versions of FORM the syntax was different here. The parentheses were curly brackets and the separators the symbol `|`. This was made to facilitate the use of strings that might contain commas. In practise however, this turned out to be far from handy. In addition the new preprocessor calculator is a bit more active and hence an instruction of the type

```
#call test{1}
```

will now be intercepted by the preprocessor calculator and changed into

```
#call test1
```

Because there are many advantages to the preprocessor calculator treating the parameters of the procedures before they are called (in the older versions it did not do this), the notation has been changed. FORM still understands the old notation, provided that there is no conflict with the preprocessor calculator. Hence

```
#call test{1|a}
#call test{1,a}
#call test(1|a)
#call test(1,a)
```

are all legal and give the same result, but only the last notation will work in future versions of FORM.

Nowadays also the use of the argumentfield wildcard is allowed as in the regular functions:

```
    #define a "1"
    #define bc2 "x"
    #define bc3 "y"
    #define b "c`~a'"
    #procedure hop(c,?d);
    #redefine a "3"
    #message This is the call: `c',`?d'
    #endprocedure

    #redefine a "2"
    #message This is b: `b'
~~~This is b: c2

    #call hop(`b`!b''`!b'`b'`!b'`b',`~a',`b',`a')
~~~This is the call: xc2c3c2c3,3,c3,2

    .end
```

We also see here that the rules about delayed substitution (see also the #define instruction on page 1.12) apply. The use of ‘!b’ cancels the delayed substitution that is asked for in the definition of b.

## 1.7 #case

Syntax:

#case string

See also switch (1.32), endswitch (1.19), break (1.5), default (1.11)

The lines after the #case instruction will be used if either this is the first #case instruction of which the string matches the string in the #switch instruction, or the control flow was already using the lines before this #case instruction and there was no #break instruction (this is called fall-through). The control flow will include lines either until the next matching #break instruction, or until the matching #endswitch instruction.

## 1.8 #close

Syntax:

#close <filename>

See also write (1.36), append (1.4, create (1.10, remove (1.30)

This instruction closes the file by the given name, if such a file had been opened by the previous #write instruction. Normally FORM closes all such files at the end of execution. Hence the user would not have to worry about this. The use of a subsequent #write instruction with the same file name will remove the old contents and hence start basically a new file. There are times that this is useful.

## 1.9 #commentchar

Syntax:

#commentchar character

The specified character should be a single non-whitespace character. There may be white space (blanks and/or tabs) before or after it. The character will take over the role of the comment character. *i.e.* any line that starts with this character in column 1 will be considered commentary. This feature was provided because output of some other algebra programs could put the multiplication sign in column 1 in longer expressions.

The default commentary character is $*$.

## 1.10 #create

Syntax:

#append <filename>

See also write (1.36), close (1.8), append (1.4), remove (1.30)

Opens the named file for writing. If the file existed already, its previous contents will be lost. The next #write instruction will add to it. In principle this instruction is not needed, because the #write instruction would create the file if it had not been opened yet at the moment of writing.

## 1.11 #default

Syntax:

#default

See also switch (1.32), endswitch (1.19), case (1.7), break (1.5)

Control flow continues after this instruction if there is no #case instruction of which the string matches the string in the #switch instruction. Control flow also continues after this instruction, if the lines before were included and there was no #break instruction to stop the control flow (fall-through). Control flow will stop either when a matching #break instruction is reached, or when a matching #endswitch is encountered. In the last case of course control flow will continue after the #endswitch instruction.

## 1.12 #define

Syntax:

#define name ”string”

See also redefine (1.29), undefine (1.35)

in which name refers to the name of the preprocessor variable to be defined and the contents of the string will form the value of the variable. The double quotes are mandatory delimiters of the string.

The use of the #define instruction creates a new instance of the preprocessor variable with the given name. This means that the old instance remains. If for some reason the later instance becomes undefined (see for instance #undefine), the older instance will be the one that is active. If the old definition is to be overwritten, one should use the #redefine instruction.

As of version 3.2 preprocessor variables can also have arguments as in the C language. Hence

#define var(a,b) ”(‘~a’+‘~b’+‘c’)”

is allowed. The parameters should be referred to inside a pair of ‘’ as with all preprocessor variables. A special feature is the socalled delayed substitution. With macro’s like the above the question is always when a preprocessor variable will be substituted. Take for instance

```
#define c "3"
#define var1(a,b) "(‘~a’+‘~b’+‘c’)"
#define var2(a,b) "(‘~a’+‘~b’+‘~c’)"
#redefine c "4"
Local F1 = ‘var1(1,2)’;
Local F2 = ‘var2(1,2)’;
Print;
.end

F1 =
   6;

F2 =
   7;
```

The parameter c will be substituted inmediately when var1 is defined. In var2 it will be only substituted when var2 is used. It should be clear that a and b should also be used in the delayed fashion because they do not exist yet at the moment of the definition of var1 and var2. Notice also that the whole macro, with its arguments should be placed between the backquote and the quote. Another example can be found with the call instruction. See section 1.6

## 1.13 #do

Syntax:
#do lvar = i1,i2
#do lvar = i1,i2,i3
#do lvar = {string1|...|stringn}
#do lvar = {string1,...,stringn}
#do lvar = nameofexpression
See also enddo (1.16)
The #do instruction needs a matching #enddo instruction. All code inbetween these two instructions will be read as many times as indicated in the parameter field of the #do instruction. The parameter lvar is a preprocessor variable of which the value is determined by the other parameters. Inside the loop it should be referred to by enclosing its name between a backquote/quote pair as is usual for preprocessor variables. The various possible parameter fields have the following meaning:

**#do lvar = i1,i2** The parameters i1 and i2 should be integers. The first time in the loop lvar will get the value of i1 (as a string) and each next time its value will be one greater (translated into a string again). The last time in the loop the value of lvar will be the greatest integer that is less or equal to i2. If i2 is less than i1, the loop is skipped completely.

**#do lvar = i1,i2,i3** The parameters i1,i2 and i3 should be integers. The first time in the loop lvar will get the value of i1 (as a string) and each next time its value will be incremented by adding i3 (translated into a string again). If i3 is positive, the last value of lvar will be the one for which lvar+i3 is greater than i2. If i2 is less than i1, the loop is skipped completely.

If i3 is negative the last value of lvar will be the one for which lvar+i3 is less than i2. If i3 is zero there will be an error.

**#do lvar = {string1|...|stringn}** The first time in the loop the value of lvar is the string indicated by string1, the next time will be string2 etc till the last time when it will be stringn. This is called a listed loop. The notation with the | is an old notation which is still accepted. The new notation uses a comma instead.

**#do lvar = {string1,...,stringn}** The first time in the loop the value of lvar is the string indicated by string1, the next time will be string2 etc till the last time when it will be stringn. This is called a listed loop.

**#do lvar = expression** The loop variable will take one by one for its value all the terms of the given expression. This is protected against changing the expression inside the loop by making a copy of the expression inside the memory. Hence one should be careful with very big expressions. An expression that is zero gives a loop over zero terms, hence the loop is never executed.

The first two types of #do instructions are called numerical loops. In the parameters of numerical loops the preprocessor calculator is invoked automatically. One should make sure not to use a leading { for the first numerical parameter in such a loop.

After a loop has been finished, the corresponding preprocessor variable will be undefined. This means that if there is a previous preprocessor variable by the same name, the value of the #do instruction will be used inside the loop, and afterwards the old value will be active again.

It is allowed to overwrite the value of a preprocessor #do instruction variable. This can be very useful to create the equivalent of a repeat loop that contains .sort instructions as in

```
#do i = 1,1
    id,once,x = y+2;
    if ( count(x,1) > 0 ) redefine i "0";
    .sort
#enddo
```

A few remarks are necessary here. The redefine should be before the last .sort inside the loop, because the #do instruction is part of the preprocessor. Hence the value of i is considered before the module is executed. This means that if the redefine would be after the .sort, two things would go wrong: First the loop would be terminated before the redefine would ever make a chance of being executed. Second the statement would be compiled in the expectation that there is a variable i, but then the loop would be terminated. Afterwards, when the statement is being executed it would refer to a variable that does not exist any longer.

If one wants to make a loop over the externals of the brackets of an expression only one needs to do some work. Assume we have the expression F and we want to loop over the brackets in x and y:

```
L   FF = F;
Bracket x,y;
.sort
CF acc,acc2;
Skip F;
Collect acc,acc2;
id  acc(x?) = 1;
```

```
id  acc2(x?)= 1;
B   x,y;
.sort
Skip F;
Collect acc;
id  acc(x?) = 1;
.sort
#do i = FF
L   G = F[`i'];
    .
    .
#enddo
```

Notice that we have to do the collect trick twice because the first time the bracket could be too long for one term. The second time that restriction doesn't exist because besides the x and the y there are only integer coefficients.

## 1.14 #else

Syntax:
#else
See also if (1.21), endif (1.17), elseif (1.15), ifdef (1.22), ifndef (1.23)
This instruction is used inside a #if/#endif construction. The code that follows it until the #endif instruction will be read if the condition of the #if instruction (and of none of the corresponding #elseif instructions) is not true. If any of these conditions is true, this code is skipped. The reading is stopped after the matching #endif is encountered and continued after this matching #endif instruction.

## 1.15 #elseif

Syntax:
#elseif ( condition )
See also if (1.21), endif (1.17), else (1.14)
The syntax of the condition is identical to the syntax for the condition in the #if instruction. The #elseif instruction can occur between an #if and an #endif instruction, before a possible matching #else instruction. The code after this condition till the next #elseif instruction, or till a #else instruction or till a #endif instruction, whatever comes first, will be read if the condition in the #elseif instruction is true and none of the conditions in matching previous #if or #elseif instructions were true. The reading is stopped after the matching #elseif/#else/#endif is encountered and continued after the matching #endif instruction.

Example

```
#if ( `i' == 2 )
    some code
#elseif ( `i' == 3 )
    more code
#elseif ( `j' >= "x2y" )
    more code
```

```
#else
    more code
#endif
```

## 1.16 #enddo

Syntax:
  #enddo
See also do (1.13)
Used to terminate a preprocessor do loop. See the #do instruction.

## 1.17 #endif

Syntax:
  #endif
See also if (1.21), else (1.14), elseif (1.15), ifdef (1.22), ifndef (1.23)
Used to terminate a #if, #ifdef or #ifndef construction. Reading will continue after it.

## 1.18 #endprocedure

Syntax:
  #endprocedure
See also procedure (1.28), call (1.6)
Each procedure must be terminated by an #endprocedure instruction. If the procedure resides in its own file, the #endprocedure will cause the closing of the file. Hence any text that is in the file after the #endprocedure instruction will be ignored.

When control reaches the #endprocedure instruction, all (local) preprocessor variables that were defined inside the procedure and all parameters of the call of the procedure will become undefined.

## 1.19 #endswitch

Syntax:
  #endswitch
See also switch (1.32), case (1.7), break (1.5), default (1.11)
This instruction marks the end of a #switch construction. After none or one of the cases of the #switch construction has been included in the control flow, reading will continue after the matching #endswitch instruction. Each #switch needs a #endswitch, unless a .end instruction is encountered first.

## 1.20 #exchange

Syntax:
  #exchange expr1,expr2
  #exchange $var1,$var2

Exchanges the names of two expressions. This means that the contents of the expressions remain where they are. Hence the order in which the expressions are processed remains the same, but the name under which one has to refer to them has been changed.

In the variety with the dollar variables the contents of the variables are exchanged. This is not much work, because dollar variables reside in memory and hence only two pointers to the contents have to be exchanged (and some extra information about the contents).

This instruction can be very useful when sorting expressions or dollar variables by their contents.

## 1.21 #if

Syntax:
#if ( condition )
See also endif (1.17), else (1.14), elseif (1.15), ifdef (1.22), ifndef (1.23)
The #if instruction should be accompanied by a matching #endif instruction. In addition there can be between the #if and the #endif some #elseif instructions and/or a single #else instruction. The condition is a logical variable that is true if its value is not equal to zero, and false if its value is zero. Hence it is allowed to use

```
#if ‘i’
    statements
#endif
```

provided that i has a value which can be interpreted as a number. If there is just a string that cannot be seen as a logical condition or a number it will be interpreted as false. The regular syntax of the simple condition is

```
#if ‘i’ == st2x
    statements
#endif
```

or

```
#if ( ‘i’ == st2x )
    statements
#endif
```

in which the compare is a numerical compare if both strings can be seen as numbers, while it will be a string compare if at least one of the two cannot be seen as a numerical object. One can also use more complicated conditions as in

```
#if ( ( ‘i’ > 5 ) && ( ‘j’ > ‘i’ ) )
```

These are referred to as composite conditions. The possible operators are

> Greater than, either in numerical or in lexicographical sense.

< Less than, either in numerical or in lexicographical sense.

>= Greater than or equal to, either in numerical or in lexicographical sense.

<= Less than or equal to, either in numerical or in lexicographical sense.

== **or** = Equal to.

! = Not equal to.

&& Logical and operator to combine conditions.

|| Logical or operator to combine conditions.

If the condition evaluates to true, the lines after the #if instruction will be read until the first matching #elseif instruction, or a #else instruction or a #endif instruction, whatever comes first. After such an instruction is encountered input reading stops and continues after the matching #endif instruction.

Like with the regular if-statement (see 5.44), there are some special functions that allow the asking of questions about objects. These are

termsin() The argument of termsin is the name of an expression or a $-expression. This function then evaluates into the number of terms in that expression.

maxpowerof() The argument of maxpowerof is the name of a symbol. This function then evaluates into the maximum power of that symbol. If no maximum power has been set in the declaration of the symbol, the general maximum power for symbols is returned (see 5.98).

minpowerof() The argument of minpowerof is the name of a symbol. This function then evaluates into the minimum power of that symbol. If no minimum power has been set in the declaration of the symbol, the general minimum power for symbols is returned (see 5.98).

## 1.22 #ifdef

Syntax:

#ifdef ‘prevar’

See also if (1.21), endif (1.17), else (1.14), ifndef (1.23)

If the named preprocessor variable has been defined the condition is true, else it is false. For the rest the instruction behaves like the #if instruction.

## 1.23 #ifndef

Syntax:

#ifndef ‘prevar’

See also if (1.21), endif (1.17), else (1.14), ifdef (1.22)

If the named preprocessor variable has been defined the condition is false, else it is true. For the rest the instruction behaves like the #if instruction.

## 1.24 #include

Syntax:

#include[-+] filename

#include[-+] filename # foldname

The named file is searched for and opened. Reading continues from this file until its end. Then the file will be closed and reading continues after the #include instruction. If a foldname is specified, FORM will only read the contents of the first fold it encounters in the given file that has the specified name.

The file is searched for in the current directory, then in the path specified in the -p option when FORM is started (see the chapter on running FORM). If this option has not been used, FORM will look for the environment variable FORMPATH. If this variable exists it will be interpreted as a path and FORM will search the indicated directories for the given file. If none is found there will be an error message and execution will be halted.

The optional + or - sign after the name has influence on the listing of the contents of the file. A - sign will have the effect of a #- instruction during the reading of the file. A plus sign will have the effect of a #+ instruction at the beginning of the file.

A fold is defined by a starting line of the format:

```
    *--#[ name :
```

and a closing line of the format

```
    *--#] name :
```

in which the first character is actually the current commentary character (see the #commentchar instruction). All lines between two such lines are considered to be the contents of the fold. If FORM decides that it needs this fold, it will read these contents and put them in its input stream. More about folds is explained in the manual of the STedi editor which is also provided in the FORM distribution.

## 1.25 #message

Syntax:

#message themessagestring

This instruction places a message in the output that is clearly marked as such. It is printed with an initial three characters in front as in

```
    Symbols a,b,c;
    #message Simple example;
~~~Simple example;
    Local F = (a+b+c)^10;
    .end

Time =       0.00 sec    Generated terms =         66
                F        Terms in output =         66
                         Bytes used      =       1138
```

Note that the semicolon is not needed and if present is printed as well. If one needs messages without this clear markation, one should use the #write instruction.

## 1.26 #pipe

Syntax:

#pipe systemcommand

See also system (1.33)

This forces a system command to be executed by the operating system. The complete string (excluding initial blanks or tabs) is passed to the operating system. Next FORM will intercept the output of whatever is produced and read that as input. Hence, whenever output is produced

FORM will take action, and it will wait when no output is ready. After the command has been finished, FORM will continue with the next line. This instruction has only been implemented on systems that support pipes. This is mainly UNIX and derived systems. Note that this instruction also introduces operating system dependent code. Hence it should be used with great care.

## 1.27 #preout

Syntax:
    #preout ON
    #preout OFF
Turns listing of the output of the preprocessor to the compiler on or off. Example:

```
    #PreOut ON
    S   a1,...,a4;
 S,a1,a2,a3,a4
    L   F = (a1+...+a4)^2;
 L,F=(a1+a2+a3+a4)^2
    id  a4 = -a1;
 id,a4=-a1
    .end

Time =       0.00 sec    Generated terms =         10
                F        Terms in output =          3
                         Bytes used      =         52
```

## 1.28 #procedure

Syntax:
    #procedure name(var1,...,varn)
See also endprocedure (1.18), call (1.6)
Name is the name of the procedure. It will be referred to by this name. If the procedure resides in a separate file the name of the file should be name.prc and the #procedure instruction should form the first line of the file. The # should be the first character of the file. The parameter field is optional. If there are no parameters, the procedure should also be called without parameters (see the #call instruction). The parameters (here called var1 to varn) are preprocessor variables and hence they should be referred to between a backquote/quote pair as in 'var1' to 'varn'. If there exist already variables with such names when the procedure is called, the new definition comes on top of the old one. Hence in the procedure (and procedures called from it, unless the same problems occurs there too, as would be the case with recursions) the new definition is used, and it is released again when control returns from the procedure. After that the old definition will be in effect again.

If the procedure is included in the regular input stream, FORM will read the text of the procedure until the #endprocedure instruction and store it in a special buffer. When the procedure is called, FORM will read the procedure from this buffer, rather than from a file. In systems where file transfer is slow (very busy server with a slow network) this may be faster, especially when many small procedures are called.

## 1.29 #redefine

Syntax:
#redefine name "string"
See also define (1.12), undefine (1.35)
in which name refers to the name of the preprocessor variable to be redefined. The contents of the string will be its new value. If no variable of the given name exists yet, the instruction will be equivalent to the #define instruction.

## 1.30 #remove

Syntax:
#remove filename
or
#remove <filename>
See also write (1.36), append (1.4, create (1.10, close (1.8)
Deletes the named file from the system. Under UNIX this would be equivalent to the instruction

```
#system rm filename
```

and under MS-DOS oriented systems like Windows it would be equivalent to

```
#system del filename
```

The difference with the #system instruction is that the #remove instruction does not depend on the particular syntax of the operating system. Hence the #remove instruction can always be used.

## 1.31 #show

Syntax:
#show [preprocessorvariablename[s]]
If no names are present, the contents of all preprocessor variables will be printed to the regular output. If one or more preprocessor variables are specified (separated by comma's), only their contents will be printed. The preprocessor variables should be represented by their name only. No enclosing backquote/quote should be used, because that would force a substitution of the preprocessor variable before the instruction gets to see the name. Example:

```
    #define MAX "3"
    Symbols a1,...,a`MAX';
    L F = (a1+...+a`MAX')^2;
    #show
#The preprocessor variables:
0: VERSION_ = "3"
1: SUBVERSION_ = "1"
2: NAMEVERSION_ = ""
3: DATE_ = "Thu Jan 27 05:43:20 2005"
4: NAME_ = "testpre.frm"
5: CMODULE_ = "1"
6: MAX = "3"
```

```
.end

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

We see that the variable MAX has indeed the value 3. There are six additional variables which have been defined by FORM itself. Hence the trailing underscore which cannot be used in user defined names. The current version of FORM is shown in the variable VERSION_ and the name of the current program is given in the variable NAME_. For more about the system defined preprocessor variables see 1.1.

There is another preprocessor variable that does not show in the listings. Its name is SHOW-INPUT_. This variable has the value one if the listing of the input is on and the value zero if the listing of the input is off.

## 1.32 #switch

Syntax:
#switch string
See also endswitch (1.19), case (1.7), break (1.5), default (1.11)
the string could for instance be a preprocessor variable as in

```
#switch ‘i’
```

The #switch instruction, together with #case, #break, #default and #endswitch, allows the user to conveniently make code for a number of cases that are distinguished by the value of a preprocessor variable. In the past this was only possible with the use of folds in the #include instruction and the corresponding include file (see 1.24). Because few people have an editor that can handle the folds in a proper way, it was judged that the more common switch mechanism might be friendlier. The proper syntax of a complete construction would be

```
#switch ‘par’
#case 1
   some statements
#break
#case ax2
   other statements
#break
#default
   more statements
#break
#endswitch
```

The number of cases is not limited. The compare between the strings in the #switch instruction and in the #case instructions is as a text string. Hence numerical strings have no special meaning. If a #break instruction is omitted, control may go into another case. This is called fall-through. This is a way in which one can have the same statements for several cases. The #default instruction is not mandatory.

FORM will look for the first case of which the string matches the string in the #switch instruction. Input reading (control flow) starts after this #case instruction, and continues till either a #break instruction is encountered, or the #endswitch is met. After that input reading continues after the #endswitch instruction. If no case has a matching string, input reading starts after the #default instruction. If no #default instruction is found, input reading continues after the matching #endswitch instruction.

#switch constructions can be nested. They can be combined with #if constructions, #do instructions, etc. but they should obey normal nesting rules (as with nesting of brackets of different types).

## 1.33 #system

Syntax:

  #system systemcommand

See also pipe (1.26)

This forces a system command to be executed by the operating system. The complete string (excluding initial blanks or tabs) is passed to the operating system. FORM will then wait until control is returned. Note that this instruction introduces operating system dependent code. Hence it should be used with great care.

## 1.34 #terminate

Syntax:

  #terminate <exit code>

This forces FORM to terminate execution immediately. If an exit code is given (an integer number), this will be the return value that FORM gives to the shell program from which it was run. If no return value is specified, the value -1 will be returned.

## 1.35 #undefine

Syntax:

  #undefine name

See also define (1.12), redefine (1.29)

in which name refers to the name of the preprocessor variable to be undefined. This statement causes the given preprocessor variable to be removed from the stack of preprocessor variables. If an earlier instance of this variable existed (other variable with the same name), it will become active again. There are various other ways by which preprocessor variables can become undefined. All variables belonging to a procedure are undefined at the end of a procedure, and so are all other preprocessor variables that were defined inside this procedure. The same holds for the preprocessor variable that is used as a loop parameter in the #do instruction.

## 1.36 #write

Syntax:

  #write [<filename>] "formatstring" [,variables]

See also append (1.4), create (1.10), remove (1.30), close (1.8)

If there is no file specified, the output will be to the regular output channel. If a file is specified, FORM will look whether this file is open already. If it is open already, the specified output will be added to the file. If it is not open yet it will be opened. Any previous contents will be lost. This would be equivalent to using the #create instruction first. If output has to be added to an existing file, the #append instruction should be used first.

The format string is like a format string in the language C. This means that it is placed between double quotes. It will contain text that will be printed, and it will contain special character sequences for special actions. These sequences and the corresponding actions are:

**\ n** A newline character.

**\ t** A tab character.

**\ ”** A double quote character.

**\ b** A backslash character.

**%%** The character %.

**%** If the last character in the string, it causes the omission of a linefeed at the end of the printing. Note that if this happens in the regular output (as opposed to a file) there may be interference with the listing of the input.

**%$** A dollar expression. The expression should be indicated in the list of variables. Each occurrence of %$ will look for the next variable.

**%e** An active expression. The expression should be indicated in the list of variables. Each occurrence of %e will look for the next variable. Unlike the output caused by the print statement the expression will be printed without its name and there will also be no = sign unless there is one in the format string of course. If the current output format is fortran output there is an extra option. After the name of the expression one should put between parentheses the name to be used when there are too many continuation cards.

**%E** Like %e, but whereas the %e terminates the expression with a ;, the %E does not give this trailing semicolon.

**%s** A string. The string should be given in the list of variables and be enclosed between double quotes. Each occurrence of %s will look for the next variable in the list.

If no special variables are asked for (by means of %$, %e, %E or %s) the list of variables will be ignored (if present). Example:

```
    Symbols a,b;
    L   F = a+b;
    #$a1 = a+b;
    #$a2 = (a+b)^2;
    #$a3 = $a1^3;
    #write " One power: %$\n Two powers: %$\n Three powers: %$\n%s"\
           ,$a1,$a2,$a3," The end"
 One power: a+b
 Two powers: 2*a*b+a^2+b^2
 Three powers: 3*a*b^2+3*a^2*b+a^3+b^3
 The end
    .end
```

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

We see that the writing occurs immediately after the #write instruction, because it is done by the preprocessor. Hence the output comes before the execution of the expression F.

```
    S   x1,...,x10;
    L   MyExpression = (x1+...+x10)^4;
    .sort
    Format Fortran;
    #write <fun.f> "      FUNCTION fun(x1,x2,x3,x4,x5,x6,x7,x8,x9,x10)"
    #write <fun.f> "      REAL x1,x2,x3,x4,x5,x6,x7,x8,x9,x10"
    #write <fun.f> "      fun = %e",MyExpression(fun)
    #write <fun.f> "      RETURN"
    #write <fun.f> "      END"
    .end
```

Some remarks are necessary here. Because the #write is a preprocessor instruction, the .sort is essential. Without it, the expression has not been worked out at the moment we want to write. The name of the expression is too long for fortran, and hence the output file will use a different name (in this case the name 'fun' was selected). The output file looks like

```
      FUNCTION fun(x1,x2,x3,x4,x5,x6,x7,x8,x9,x10)
      REAL x1,x2,x3,x4,x5,x6,x7,x8,x9,x10
      fun = 24*x1*x2*x3*x4 + 24*x1*x2*x3*x5 + 24*x1*x2*x3*x6 + 24*x1*x2
     & *x3*x7 + 24*x1*x2*x3*x8 + 24*x1*x2*x3*x9 + 24*x1*x2*x3*x10 + 12*
          .....
     & x8 + 4*x6**3*x9 + 4*x6**3*x10 + x6**4 + 24*x7*x8*x9*x10 + 12*x7*
     & x8*x9**2
      fun = fun + 12*x7*x8*x10**2 + 12*x7*x8**2*x9 + 12*x7*x8**2*x10 +
     & 4*x7*x8**3 + 12*x7*x9*x10**2 + 12*x7*x9**2*x10 + 4*x7*x9**3 + 4*
     & x7*x10**3 + 12*x7**2*x8*x9 + 12*x7**2*x8*x10 + 6*x7**2*x8**2 +
     & 12*x7**2*x9*x10 + 6*x7**2*x9**2 + 6*x7**2*x10**2 + 4*x7**3*x8 +
     & 4*x7**3*x9 + 4*x7**3*x10 + x7**4 + 12*x8*x9*x10**2 + 12*x8*x9**2
     & *x10 + 4*x8*x9**3 + 4*x8*x10**3 + 12*x8**2*x9*x10 + 6*x8**2*
     & x9**2 + 6*x8**2*x10**2 + 4*x8**3*x9 + 4*x8**3*x10 + x8**4 + 4*x9
     & *x10**3 + 6*x9**2*x10**2 + 4*x9**3*x10 + x9**4 + x10**4

      RETURN
      END
```

and each time after 19 continuation lines we have to break the expression and use the `fun = fun +` trick to continue.

## 1.37 Some remarks

It should be noted that the various constructions like #do/#enddo, #procedure/#endprocedure, #switch/#endswitch and #if/#endif all create a certain environment. These environments cannot be mixed. This means that one cannot make code of the type

```
#do i = 1,5
 #if ( ‘MAX’ > ‘i’ )
    id f(‘i’) = g‘i’(x);
#enddo
 some statements
#do i = 1,5
 #endif
#enddo
```

whether this could be considered useful or not. Similarly one cannot make a construction that might be very useful:

```
#do i = 1,5
  #do j‘i’ = 1,3
#enddo
  some statements
#do i = 1,5
  #enddo
#enddo
```

Currently the syntax does not allow this. This may change in the future.

# Chapter 2

# Modules

Modules are the basic execution blocks. Statements are always part of a module, and they will be executed only when the module is executed. This is directly opposite to preprocessor instructions which are executed when they are encountered in the input stream.

Modules are terminated by a line that starts with a period. Such a line is called the module instruction. Once the module instruction has been recognized, the compilation of the module is terminated and the module will be executed. All active expressions will be processed one by one, term by term. When each term of an expression has been through all statements of the module, the combined results of all operations on all the terms of the expression will be sorted and the resulting expression will be sent to the output. This can be an intermediate file, or it can be some memory, depending on the size of the output. If the combined output of all active expressions is less than the parameter "ScratchSize", the results stay in memory. ScratchSize is one of the setup parameters (see chapter 11).

A module consists in general of several types of statements:

**Declarations** These are the declarations of variables.

**Specifications** These tell what to do with existing expressions as a whole.

**Definitions** These define new expressions.

**Executable statements** The operations on all active expressions.

**OutputSpecifications** These specify the output representation.

**End-of-module specifications** Extra settings that are for this module only.

**Mixed statements** They can occur in various classes. Most notably the print statement.

Statements must occur in such an order that no statement follows a statement of a later category. The only exception is formed by the mixed statements, which can occur anywhere. This is different from earlier versions of FORM in which the order of the statements was not fixed. This did cause a certain amount of confusion about the workings of FORM.

There are several types of modules.

**.sort** The general end-of-module. Causes execution of all active expressions, and prepares them for the next module.

**.end** Executes all active expressions and terminates the program.

**.store** Executes all active expressions. Then it writes all active global expressions to an intermediate storage file and removes all other non-global expressions. Removes all memory of declarations except for those that were made before a .global instruction.

**.global** No execution of expressions. It just saves declarations made thus far from being erased by a .store instruction.

**.clear** Executes all active expressions. Then it clears all buffers with the exception of the main input stream. Continues execution in the main input stream as if the program had started at this point. The only parameters that cannot be changed at this point are the setup parameters. They remain.

Each program must be terminated by a .end instruction. If such an instruction is absent and FORM encounters an end-of-input it will issue a warning and generate a .end instruction.

Module instructions can contain a special commentary that will be printed in all statistics that are generated during the execution of the module. This special commentary is restricted to 20 characters (the statistics have a fixed format and hence there is only a limited amount of space available). This commentary is initiated by a colon and terminated by a semicolon. The characters between this colon and the semicolon are the special message, also called advertisement. Example

```
.sort:Eliminate x;
```

would give in the statistics something like

```
Time =        0.46 sec    Generated terms =        360
                 F        Terms in output =        360
           Eliminate x  Bytes used      =       4506
```

If the statistics are switched off, there will be no printing of this advertisement either.

For backwards compatibility there is still a mechanism to pass module options via the module instructions. This is a feature which will probably disappear in future versions of FORM. We do give the syntax to allow the user to identify the option properly and enable proper translation into the ModuleOption statement.

```
    .sort(PolyFun=functionname);
    .sort(PolyFun=functionname):advertisement;
```

causes the given function to be treated as a polynomial function. This means that its (single) argument would be treated as the coefficient of the terms. The action of FORM on individual terms is

1. Ignore polynomial functions with more than one argument.

2. If there is no polynomial function with a single argument, generate one with the argument 1.

3. If there is more than one polynomial function with a single argument, multiply the arguments and replace these functions with a single polynomial function with the product of the arguments for a single argument.

4. Multiply the argument of the polynomial function with the coefficient of the term. Replace the coefficient itself by one.

If, after this, two terms differ only in the argument of their polynomial function FORM will add the arguments and replace the two terms by a single term which is identical to the two previous terms except for that the argument of its polynomial function is the sum of their two arguments.

It should be noted that the proper placement of .sort instructions in a FORM program is an art by itself. Too many .sort instructions cause too much sorting, which can slow execution down considerably. It can also cause the writing of intermediate expressions which are much larger than necessary, if the next statements would cause great simplifications. Not enough .sort instructions can make that cancellations are postponed unnecessarily and hence much work will be done double. This can slow down execution by a big factor. First an example of a superfluous .sort:

```
    S a1,...,a7;
    L F = (a1+...+a7)^16;
    .sort

Time =      31.98 sec    Generated terms =      74613
                F        Terms in output =      74613
                         Bytes used      =    1904316
    id a7 = a1+a2+a3;
    .end

Time =     290.34 sec
                F        Terms active    =      87027
                         Bytes used      =    2253572

Time =     295.20 sec    Generated terms =     735471
                F        Terms in output =      20349
                         Bytes used      =     538884
```

Without the sort the same program gives:

```
    S a1,...,a7;
    L F = (a1+...+a7)^16;
    id a7 = a1+a2+a3;
    .end

Time =     262.79 sec
                F        Terms active    =      94372
                         Bytes used      =    2643640

Time =     267.81 sec    Generated terms =     735471
                F        Terms in output =      20349
                         Bytes used      =     538884
```

and we see that the sorting in the beginning is nearly completely wasted. Now a clear example of not enough .sort instructions. A common problem is the substitution of one power series into another. If one does this in one step one could have:

```
    #define MAX "36"
    S  j,x(:‘MAX’),y(:‘MAX’);
    *
```

```
    * Power series expansion of ln_(1+x)
    *
    L F = -sum_(j,1,‘MAX’,sign_(j)*x^j/j);
    *
    * Substitute the expansion of x = exp_(y)-1
    *
    id x = x*y;
    #do j = 2,‘MAX’+1
    id x = 1+x*y/‘j’;
    #enddo
    Print;
    .end

Time =      76.84 sec    Generated terms =      99132
                F        Terms in output =          1
                         Bytes used      =         18

   F =
      y;
```

With an extra .sort inside the loop one obtains for the same program (after suppressing some of the statistics:

```
    #define MAX "36"
    S  j,x(:‘MAX’),y(:‘MAX’);
    *
    * Power series expansion of ln_(1+x)
    *
    L F = -sum_(j,1,‘MAX’,sign_(j)*x^j/j);
    *
    * Substitute the expansion of x = exp_(y)-1
    *
    id x = x*y;
    #do j = 2,‘MAX’+1
    id x = 1+x*y/‘j’;
    .sort: step ‘j’;

Time =       0.46 sec    Generated terms =        360
                F        Terms in output =        360
                 step 2  Bytes used      =       4506
    #enddo
           .
           .
           .
Time =       3.07 sec    Generated terms =          3
                F        Terms in output =          1
                step 37  Bytes used      =         18
    Print;
    .end
```

```
Time =       3.07 sec    Generated terms =          1
                F        Terms in output =          1
                         Bytes used      =         18

   F =
      y;
```

It is very hard to give general rules that are more specific than what has been said above. The user should experiment with the placements of the .sort before making a very large run.

# Chapter 3

# Pattern matching

Substitutions are made in FORM by specifying a generic object that should be replaced by an expression. This generic object is called a pattern. Patterns that the user may already be familiar with are the regular expressions in many UNIX based systems or just a statement like `ls *.frm` to list only files of which the name ends in `.frm`. In this case the `*` is called a wildcard that can take any string value. In symbolic manipulation there will be wildcards also, but their nature will be different. They are also indicated in a different way.

In FORM wildcard variables are indicated by attaching a question mark (?) to the name of a variable. The type of the variable indicates what type of object we are looking for. Assume the following id-statements:

```
Functions f,g;
Symbol x;

id f(g?,x) = g(x,x);
```

In this statement g will match any function and hence all occurrences of f, in which the first argument is a function and the second argument is the symbol x, will match. In the right hand side the function g will be substituted by whatever identity g had to assume in the left hand side to make the match. Hence `f(f,x)` will be replaced by `f(x,x)`.

In general function wildcards can only match functions. Even though tensors are special functions, regular function wildcards cannot match tensors, and tensor wildcards cannot match functions. However commuting function wildcards can match noncommuting functions *et vise versa*.

Index wildcards can only match indices. The dimension of the indices is not relevant. Hence:

```
id f(mu?,mu?) = 4;
```

would match both `f(ka,ka)` and `f(2,2)`. We will see later how to be more selective about such matches.

When the same wildcard occurs more than once in a pattern, it should be matched by the same object in all its occurrences. Hence the above pattern would not match `f(mu,nu)`.

There is one complication concerning the above rule of index wildcards only matching indices. FORM writes contractions with vectors in a special shorthand notation. Hence `f(mu)*p(mu)` becomes `f(p)`. This means that the substitution

```
id f(mu?)*g(nu?) = fg(mu,nu);
```

should also replace the term `f(p)*g(q)` by `fg(p,q)`. In this case it looks like the wildcard indices matched the vectors. This is however not the case, because if we take the previous pattern (with the `f(mu?,mu?)`), it is not going to match the term `f(p,p)`, because this term should be read as something of the type `f(mu, nu)*p(mu)*p(nu)` and that term does not fit the pattern `f(mu?,mu?)`.

Vector wildcards can match vectors, but they can also match vector-like expressions in function arguments. A vector-like expression is an expression in which all terms contain one single vector without indices, possibly multiplied by other objects like coefficients, functions or symbols. Hence

```
id f(p?) = p.p;
```

would match `f(q)`, `f(2*q-r)` and `f(a*q+f(x)*r)`, if p, q and r are vectors, and a and x are symbols, and f is a function. It would not match `f(x)` and neither would it match `f(q*r)`, nor `f(a*q+x)`.

Wildcard symbols are the most flexible objects. They can match symbols, numbers and expressions that do not contain loose indices or vectors without indices. These last objects are called scalar objects. Hence wildcard symbols can match all scalar objects. In

```
id x^n? = x^(n+1)/(n+1);
```

the wildcard symbol n would normally match a numerical integer power. In

```
id f(x?) = x^2;
```

there would be a match with `f(y)`, with `f(1+2*y)` and with `f(p.p)`, but there would not be a match with `f(p)` if p is a vector.

There is one extra type of wildcards. This type is rather special. It refers to groups of function arguments. The number of arguments is not specified. These variables are indicated by a question mark followed by a name (just the opposite of the other wildcard variables), and in the right hand side they are also written with the leading question mark:

```
id f(?name) = g(1,?name);
```

In this statement all occurrences of f with any number of arguments (including no arguments) will match. Hence `f(mu,nu)` will be replaced by `g(1,mu,nu)`. In the case that f is a regular function and g is a tensor, it is conceivable that the arguments in `?name` will not fit inside a tensor. For instance `f(x)`, with x a symbol, would match and FORM would try to put the symbol inside the tensor g. This would result in a runtime error. In general FORM will only accept arguments that are indices or single vectors for a substitution into a tensor. The object `?name` is called an argument field wildcard.

One should realize that the use of multiple argument field wildcards can make the pattern matching slow.

```
id f(?a,p1?,?b,p2?,?c,p3?,?d)*g(?e,p3?,?f,p1?,?g,p2?,?h) = ....
```

may involve considerable numbers of tries, especially when there are many occurrences of f and g in a term. One should be very careful with this.

A complication is the pattern matching in functions with symmetry properties. In principle FORM has to try all possible permutations before it can conclude that a match does not occur. This can become rather time consuming when many wildcards are involved. FORM has a number of tricks built in, in an attempt to speed this up, but it is clear that for many cases these tricks are not enough. This type of pattern matching is one of the weakest aspects of 'artificial intelligence' in general. It is hoped that in future versions it can be improved. For the moment the practical consequence is that argument field wildcards cannot be used in symmetric and antisymmetric functions. If one needs to make a generic replacement in a symmetric function one cannot use

```
CFunction f(symmetric),g(symmetric);
id  f(?a) = ....;
```

but one could try something like

```
CFunction f(symmetric),ff,g(symmetric);
id  f(x1?,...,x5?) = ff(x1,...,x5);
id  ff(?a) = ...;
id  ff(?a) = f(?a);
```

if f has for instance 5 arguments. If different numbers of arguments are involved, one may need more than one statement here. It just shows that one should at times be a bit careful with overuse of (anti)symmetric functions. Cyclic functions do not have this restriction.

When there are various possibilities for a match, FORM will just take the first one it encounters. Because it is not fixed how FORM searches for matches (in future versions this may be changed without notice) one should try to avoid ambiguities as in

```
id f(?a,?b) = g(?a)*h(?b);
```

Of course the current search method is fully consistent (and starts with all arguments in `?a` and none in `?b` etc, but a future pattern matcher may do it in a different order.

When two argument field wildcards in the left hand side have the same name, a match will only occur, when they match the same objects. Hence

```
id f(?a,?a) = g(?a);
```

will match `f(a,b,a,b)` or just `f` (in which case `?a` will have zero arguments), but it will not match `f(b,b,b)`.

Sometimes it is useful when a search can be restricted to a limited set of objects. For this FORM knows the concept of sets. If the name of a set is attached after the question mark, this is an indication for FORM to look only for matches in which the wildcard becomes one of the members of the set:

```
Symbols a,a1,a2,a3,b,c;
Set aa:a1,a2,a3;

id f(a?aa) = ...
```

would match `f(a1)` but not `f(b)`. Sets can also be defined dynamically by enclosing the elements between curly brackets as in:

```
Symbols a,a1,a2,a3,b,c;

id f(a?{a1,a2,a3}) = ...
```

Sets of symbols can contain (small integer) numbers as well. Similarly sets of indices can contain fixed indices (positive numbers less than the value of fixindex (see the chapter on the setup 11). This means that some sets can be ambiguous in their nature.

Sometimes sets can be used as some type of array. In the case of

```
Symbols a,a1,a2,a3,b,c,n;
Set aa:a1,a2,a3;

id f(a?aa[n]) = ...
```

not only does a have to be an element of the set aa, but if it is an element of that set, n will become the number of the element that has been matched. Hence for `f(a2)` the wildcard a would become `a2` and the wildcard n would become 2. These objects can be used in the righthandside. One can also use sets in the righthandside with an index like the n of the previous example:

```
Symbols a,a1,a2,a3,b1,b2,b3,c,n;
Functions f,g1,g2,g3;
Set aa:a1,a2,a3;
Set bb:b1,b2,b3;
Set gg:g1,g2,g3;

id f(a?aa[n]) = gg[n](bb[n]);
```

which would replace `f(a2)` by `g2(b2)`.

There is one more mechanism by which the array nature of sets can be used. In the statement (declarations as before)

```
id f(a?aa?bb) = a*f(a);
```

a will have to be an element of the set aa, but after the matching it takes the identity of the corresponding element of the set bb. Hence `f(a2)` becomes after this statement `b2*f(b2)`.

Wildcards can also give their value directly to \$-variables (see the chapter of the \$-variables 4). If a \$-variable is attached to a wildcard (if there is a set restriction, it should be after the set) the \$-variable will obtain the same contents as the wildcard, provided a match occurs. If there is more than one match, the last match will be in the \$-variable.

```
id f(a?$w) = f(a);
```

will put the match of a in `$w`. Hence in the case of `f(a2)` the \$-variable will have the value `a2`. In the case of `f(a2)*f(a3)` the eventual value of `$w` depends on the order in which FORM does the matching. This is not specified and it would not be good strategy to make programs that will depend on it. A future pattern matcher might do it differently! But one could do things like

```
while ( match(f(a?$w)) );
    id f($w) = ....
    id g($w) = ....
endwhile;
```

just to make sure with which match one is working.

# Chapter 4

# The dollar variables

In the older versions of FORM there were two types of variables: the preprocessor variables and the algebraic variables. The preprocessor variables are string variables that are used by the edit features of the preprocessor to prepare the input for the compiler part of FORM. The algebraic objects are the expressions and the various algebraic variables like the symbols, functions, vectors etc. There existed however very few possibilities to communicate from the algebraic level to the decision taking at the preprocessor level. This has changed dramatically with version 3 and the introduction of the dollar variables.

Dollar variables are basically (little) expressions that can be used to store various types of information. They can be used both as preprocessor objects as well as algebraic objects. They can also be defined and given contents both by the preprocessor and during execution on a term by term basis. Dollar variables are kept in memory. Hence it is important not to make them too big, because in that case performance might suffer.

What is a legal name for a dollar variable? Dollar variables have a name that consists of a dollar sign (`$`) followed by an alphabetic character and then potentially more alphanumeric characters. Hence `$a` and `$var` and `$r4t78y0` are legal names and `$1a` is not a legal name. The variables do not have to be declared. However FORM will complain if a dollar variable is being used, before it has encountered a statement or an instruction in which the variable has been given a value. Hence giving a variable a value counts at the same time as a declaration.

What can be stored in a dollar variable?

- Algebraic expressions like in `$var = (a+b)^2;`
- Individual objects like indices, numbers, symbols.
- Zero.
- Parts of a term.
- Argument fields that consist of zero, one or more arguments.

Actually the parts of a term are treated as a complete term and hence as a special case of an algebraic expression. Internally they are stored slightly differently for speed, but at the user level this should not be noticeable. Actually, with the exception of the argument fields, FORM can convert one type into the other and will try so, depending on the use that is made of the specific dollar variable. In the case that a variable is used in a way that should not be possible (like the content of a variable is a symbol, but it is used in a position where an index is expected) there will be a runtime error.

How is a variable used?

- As a preprocessor variable. This is done by putting the variable between a pair of ‘’ as in `‘$var’`. In this case the regular print routines of FORM make a textual representation of the variable as it exists at the moment that the preprocessor excounters this object, and this string is then substituted by the preprocessor as if it were the contents of a preprocessor variable.
- Like an expression during execution time. This would be the case in the statement

  ```
  id x = y + $var;
  ```

  in which `$var` is substituted in a way that is similar to the substitution of a local expression `F` in the statement

  ```
  id x = y + F;
  ```

  except for that the dollar variable is always stored in the CPU memory.
- As an algebraic object during execution time. This could be the case with any value of the variable that is not an expression. An example would be

  ```
  id f(?a) = f(?a,$var);
  ```

  in which the dollar variable contains an argument field.
- As an algebraic object in a delayed substitution of a pattern or a special statement. This may need some clarification. If we have the statement

  ```
  id f($var) = anything;
  ```

  the compiler does not substitute the current value of `$var`. The reason is that `$var` could have a different value for each term that runs into this statement, while the compiler compiles the statement only once. Hence FORM will substitute the value of `$var` only at the moment that it will attempt the pattern matching. This is called delayed substitution. If one likes the compiler to substitute a value, one can basically let the preprocessor take care of this by typing

  ```
  id f(‘$var’) = anything;
  ```

  A similar delayed substitution takes place in statements of the type `Trace,$var;`.

How does one give a value to a dollar variable?

- In the preprocessor. This is done with an instruction of the type `#$var = 0;`. This is an instruction that can run over more than one line. The r.h.s. can be any algebraic expression. Specifically it can contain dollar variables or local/global expressions. Such expressions are worked out during the preprocessing. Hence this variable acquires a value immediately.
- During execution when control reaches a statement of the form `$var = expression;`. Again the r.h.s. can contain any normal algebraic expression including dollar variables and local/global expressions. The r.h.s. will be evaluated and the value will be assigned to `$var`. In the case that `$var` had already a value, the old value will be deleted and the new value will be ‘installed’.

- During execution when the dollar variable is assigned the value of a wildcard as in

  ```
  id f(x?$var) = whatever;
  ```

  If the function `f` occurs more than once in a term, `$var` will have the value of the last match. In the case that the value of the first match is needed one can use the option 'once' in the id-statement as in

  ```
  id,once,f(x?$var) = whatever;
  ```

  In general one can paste the dollar variable to the end of any wildcard description. Hence one can use `id f(x?{1,2,3}$var) = ...;` and

  ```
  id f(x?set[n?$var1]$var2) = ...;
  ```

Note the difference between `#$a = 0;` and `$a = 0;`. One CANNOT make a wildcard construction for dollar variables themselves as in `id f($var?) = ...;`

Dollar variables CANNOT be indexed as in `$var(2)` or something equivalent. There is however a solution at the preprocessor level for this by defining individual variables `$var1` to `$varn` and then using `` $var`i' `` or `` `$var`i'' `` for some preprocessor variable `i`.

Printing dollar variables:

- In the preprocessor one can use the `#write` instruction (see 1.36).
- During execution one can use the Print statement (see 5.76).

In both cases one should use the format string. The syntax is described in the chapters on these statements. The format descriptor of a dollar variable is `%$` and this looks after the format string for the next dollar variable. Of course one can also use the dollar variable as a preprocessor variable when printing/writing in the preprocessor.

Examples.

Counting terms:

```
S a,b;
Off statistics;
L F = (a+b)^6;
#$a = 0;
$a = $a+1;
Print "     >> After %t we have %$ term(s)",$a;
#write "     ># $a = `$a'"
 ># $a = 0
.sort
  >> After  + a^6 we have 1 term(s)
  >> After  + 6*a^5*b we have 2 term(s)
  >> After  + 15*a^4*b^2 we have 3 term(s)
  >> After  + 20*a^3*b^3 we have 4 term(s)
  >> After  + 15*a^2*b^4 we have 5 term(s)
  >> After  + 6*a*b^5 we have 6 term(s)
  >> After  + b^6 we have 7 term(s)
#write "     ># $a = `$a'"
 ># $a = 7
.end
```

Maximum power of x in an expression:

```
S x,a,b;
Off statistics;
L F = (a+b)^4+a*(a+x)^3;
.sort
#$a = 0;
if ( count(x,1) > $a ) $a = count_(x,1);
Print "      >> After %t the maximum power of x is %$",$a;
#write "     ># $a = `$a'"
 ># $a = 0
.sort
  >> After  + 3*x*a^3 the maximum power of x is 1
  >> After  + 3*x^2*a^2 the maximum power of x is 2
  >> After  + x^3*a the maximum power of x is 3
  >> After  + 4*a*b^3 the maximum power of x is 3
  >> After  + 6*a^2*b^2 the maximum power of x is 3
  >> After  + 4*a^3*b the maximum power of x is 3
  >> After  + 2*a^4 the maximum power of x is 3
  >> After  + b^4 the maximum power of x is 3
#write "     ># $a = `$a'"
 ># $a = 3
.end
```

# Chapter 5

# Statements

## 5.1 abrackets, antibrackets

| | |
|---|---|
| Type | Output control statement |
| Syntax | ab[rackets][+][-] <list of names>; |
| | antib[rackets][+][-] <list of names>; |
| See also | bracket (5.7) and the chapter on brackets (7) |

This statement does the opposite of the bracket statement (see 5.7). In the bracket statement the variables that are mentioned are placed outside brackets and inside the brackets are all other objects. In the antibracket statement the variables in the list are the only objects that are not placed outside the brackets. For the rest of the syntax, see the bracket statement (section 5.7).

## 5.2 also

| | |
|---|---|
| Type | Executable Statement |
| Syntax | a[lso] [options] <pattern> = <expression>; |
| See also | identify (5.41), idold (5.43) |

The also statement should follow either an id statement or another also statement. The action is that the pattern matching in the also statement takes place immediately after the pattern matching of the previous id statement (or also statement) and after possible matching patterns have been removed, but before the r.h.s. expressions are inserted. It is identical to the idold statement (see 5.43). Example:

```
id    x = cosphi*x-sinphi*y;
also  y = sinphi*x+cosphi*y;
```

The options are explained in the section on the id statement (see 5.41).

## 5.3 antisymmetrize

| | |
|---|---|
| Type | Executable statement |
| Syntax | an[tisymmetrize] {<name of function/tensor> [<argument specifications>];} |
| See also | symmetrize (5.99), cyclesymmetrize (5.16), rcyclesymmetrize (5.82) |

The argument specifications are explained in the section on the symmetrize statements (see 5.99).

The action of this stament is to anti-symmetrize the (specified) arguments of the functions that are mentioned. This means that the arguments are brought to 'natural order' in the notation of FORM and each exchange of arguments or groups of arguments results in a minus sign in the coefficient of the term. The 'natural order' may depend on the order of declaration of the variables. If two arguments or groups of arguments that are part in the anti-symmetrization are identical, the function is replaced by zero.

## 5.4 apply

| | |
|---|---|
| Type | Executable statement |
| Syntax | apply ["<tablename(s)>"]; |
| See also | tablebases (8), apply (8.2) |

This statement is explained in the chapter on tablebases.

## 5.5 argument

| | |
|---|---|
| Type | Executable statement |
| Syntax | argument [<argument specifications>]<br>{<name of function/set> [<argument specifications>]}; |
| See also | endargument (5.24) |

This statement starts an argument environment. Such an environment is terminated by an endargument statement (see 5.24). The statements between the argument and the endargument statements will be applied only to the function arguments as specified by the remaining information in the argument statement. This information is given by:

- No further information: the statements are applied to all arguments of all functions.
- A series of numbers: the statements are applied to the given arguments of all functions.
- A function name (or a set of functions), possibly followed by a series of numbers: the statements are applied to the numbered arguments of the function specified. If a set of functions was specified, all the functions in the set will be taken. If no numbers are specified, all arguments of the function (or elements of the set) are taken.

The combination of a function (or set) possibly followed by numbers of arguments, can occur as many times as needed. The generic numbers of arguments that refer to all functions work in addition to the numbers specified for individual functions. Example

```
Argument 2,f,1,{f,f1},3,4;
```

This specifies the second argument of all functions. In addition the first argument of `f` will be taken and then also the third and fourth arguments of `f` and `f1` will be taken.

Argument/endargument constructions can be nested.

## 5.6 auto, autodeclare

| | |
|---|---|
| Type | Declaration statement |
| Syntax | autodeclare <variable type> <list of variables to be declared>; |
| | auto <variable type> <list of variables to be declared>; |

The variable types are

| | |
|---|---|
| s[ymbol] | Declaration of symbols. For options see 5.98. |
| v[ector] | Declaration of vectors. For options see 5.112. |
| i[ndex] | Declaration of indices. For options see 5.46. |
| i[ndices] | Declaration of indices. For options see 5.46. |
| f[unctions] | Declaration of noncommuting functions. For options see 5.61. |
| nf[unctions] | Declaration of noncommuting functions. For options see 5.61. |
| cf[unctions] | Declaration of commuting functions. For options see 5.8. |
| co[mmuting] | Declaration of commuting functions. For options see 5.8. |
| t[ensors] | Declaration of commuting tensors. For options see 5.102. |
| nt[ensors] | Declaration of noncommuting tensors. For options see 5.67. |
| ct[ensors] | Declaration of commuting tensors. For options see 5.15. |

The action of the autodeclare statement is to set a default for variable types. In a statement of the type

```
AutoDeclare Symbol a,bc,def;
```

all undeclared variables of which the name starts with the character a, the string bc or the string def will be interpreted as symbols and entered in the name tables as such. In the case there are two statements as in

```
AutoDeclare CFunction b,d;
AutoDeclare Symbol a,bc,def;
```

all previously undeclared variables of which the name starts with a, bc or def will be declared as symbols. All other previously undeclared variables of which the name starts with a b or a d will be declared as commuting functions. This is independent of the order of the autodeclare statements. FORM starts looking for the most detailed matches first. Hence the variable defi will match with the string def first.

It is also allowed to use the properties of the various variables in the autodeclare statement:

```
AutoDeclare Index i=4,i3=3,i5=5;
```

This declares all previously undeclared variables of which the name starts with an i to be 4-dimensional indices, unless their names start with i3 in which case they will be three dimensional indices, or their names start with i5 in which case they will be five dimensional indices.

## 5.7 bracket

| | |
|---|---|
| Type | Output control statement |
| Syntax | b[rackets][+][-] <list of names>; |
| See also | antibracket (5.1), keep (5.49), collect(5.10) and the chapter on brackets (7) |

This statement causes the output to be reorganized in such a way that all objects in the 'list of names' are placed outside brackets and all remaining objects inside brackets. This grouping will remain till the next time that the expression is active and is being manipulated. Hence the brackets can survive skip (see 5.91), hide (see 5.40) and even save (see 5.87) and load (see 5.51) statements. The bracket information can be used by the collect (see 5.10) and keep (see 5.49) statements, as well in r.h.s. expressions when the contents of individual brackets of an expression can be picked up (see 7).

The list of names can contain names of symbols, vectors, functions and tensors. In addition it can contain dotproducts. There should be only one bracket or antibracket (see 5.1) statement in each module. If there is more than one, only the last one has an effect.

The presense of a + or − after the bracket (or anti bracket) refers to potential indexing of the brackets. Usually FORM has the information inside the terms in an expression. If it needs to search for a particular bracket it does so by starting at the beginning of that expression. This can be slow. If one likes to access individual brackets, it may be faster to tell FORM to make an index by putting the + after the bracket or antibracket keyword. For more information, see the chapter on brackets (see 7). A − indicates that no index should be made. Currently this is the default and hence there is no need to use this option. It is present just in case the default might be changed in a future version of FORM (in which FORM might for instance try to determine by itself what seems best. This option exists for case that the user would like to overrule such a mechanism).

See also the antibracket statement in 5.1.

## 5.8 cfunctions

| | |
|---|---|
| Type | Declaration statement |
| Syntax | c[functions] <list of functions to be declared>; |
| See also | functions (5.36), nfunctions (5.61) |

This statement declares commuting functions. The name of a function can be followed by some information that specifies additional properties of the preceeding function. These can be (name indicates the name of the function to be declared):

| | |
|---|---|
| name#r | The function is considered to be a real function (default). |
| name#c | The function is considered to be a complex function. This means that internally two spaces are reserved. One for the variable name and one for its complex conjugate name#. |
| name#i | The function is considered to be imaginary. |
| name(s[ymmetric]) | The function is totally symmetric. This means that during normalization FORM will order the arguments according to its internal notion of order by trying permutations. The result will depend on the order of declaration of variables. |
| name(a[ntisymmetric]) | The function is totally antisymmetric. This means that during normalization FORM will order the arguments according to its internal notion of order and if the resulting permutation of arguments is odd the coefficient of the term will change sign. The order will depend on the order of declaration of variables. |
| name(c[yclesymmetric]) | The function is cycle symmetric in all its arguments. This means that during normalization FORM will order the arguments according to its internal notion of order by trying cyclic permutations. The result will depend on the order of declaration of variables. |
| name(r[cyclesymmetric)<br>name(r[cyclic])<br>name(r[eversecyclic]) | The function is reverse cycle symmetric in all its arguments. This means that during normalization FORM will order the arguments according to its internal notion of order by trying cyclic permutations and/or a complete reverse order of all arguments. The result will depend on the order of declaration of variables. |

The complexity properties and the symmetric properties can be combined. In that case the complexity properties should come first as in

```
CFunction f1#i(antisymmetric);
```

## 5.9 chisholm

| | |
|---|---|
| Type | Executable statement |
| Syntax | chisholm [options] <spinline indices>; |
| See also | trace4 (5.107) and the chapter on gamma algebra (9) |

This statement applies the identity

$$\gamma_a\gamma_\mu\gamma_b\mathrm{Tr}[\gamma_\mu S] \quad = \quad 2\gamma_a(S+S^R)\gamma_b$$

in order to contract traces. $S$ is here a string of gamma matrices and $S^R$ is the reverse string. This identity is particularly useful when the matrices $\gamma_6 = 1+\gamma_5$ and/or $\gamma_7 = 1-\gamma_5$ are involved. The spinline index refers to which trace should be eliminated this way. The options are

| | |
|---|---|
| symmetrize | If there is more than one contraction with other gamma matrices, the answer will be the sum of the various contractions, divided by the number of different contractions. This will often result in a minimization of the number of $\gamma_5$ matrices left in the final results. |
| nosymmetrize | The first contraction encountered will be taken. No attempt is made to optimize with respect to the number of $\gamma_5$ matrices left. |

IMPORTANT: the above identity is only valid in 4 dimensions. For more details, see the chapter on gamma algebra 9.

## 5.10 collect

| | |
|---|---|
| Type | Specification statement |
| Syntax | collect <name of function>; |
| | collect <name of function> <name of other function>; |
| | collect <name of function> <name of other function> <percentage>; |
| See also | bracket (5.7), antibracket (5.1) and the chapter on brackets (7) |

Upon processing the expressions (hence expressions in hide as well as skipped expressions do not take part in this) the contents of the brackets (if there was a bracket or antibracket statement in the preceeding module) are collected and put inside the argument of the named function. Hence if the expression `F` is given by

```
F =
   a*(b^2+c)
 + a^2*(b+6)
 + b^3 + c*b + 12;
```

the statement

```
Collect cfun;
```

will change `F` into

```
F = a*cfun(b^2+c)+a^2*cfun(b+6)+cfun(b^3+c*b+12);
```

The major complication occurs if the content of a bracket is so long that it will not fit inside a single term. The maximum size of a term is limited by the setup parameter maxtermsize (see 11). If this size is exceed FORM will split the bracket contents over more than one term, in each of which it will be inside the named function. It will issue a warning that it has done so.

If a second function is specified (the alternative collect function) and if a bracket takes more space than can be put inside a single term, the bracket contents will be split over more than one term, in each of which it will be inside the alternative collect function. In this case there is no need for a warning as the user can easily check whether this has occurred by checking whether the alternative function is present in the expression.

If additionally a pertage is specified (an integer in the range of 1 to 99) this determines how big the argument must be as compared to MaxTermSize (see chapter 11 on the setup) before use is made of the alternate collect function.

## 5.11 commuting

| | |
|---|---|
| Type | Declaration statement |
| Syntax | co[mmuting] <list of functions to be declared>; |
| See also | cfunctions (5.8), functions (5.36) |

This statement is completely identical to the cfunction statement (see 5.8).

## 5.12 compress

| | |
|---|---|
| Type | Declaration statement |
| Syntax | comp[ress] <on/off>; |
| See also | on (5.71), off (5.70) |

This statement is obsolete. The user should try to use the compress option of the on (see 5.71) or the off (see 5.70) statements.

## 5.13 contract

| | |
|---|---|
| Type | Executable statement |
| Syntax | contract [<argument specifications>]; |

Statement causes the contraction of pairs of Levi-Civita tensors `e_` (see also 6) into combinations of Kronecker delta's. If there are contracted indices, and if their dimension is identical to the number of indices of the Levi-Civita tensors, the regular shortcuts are taken. If there are contracted indices with a different dimension, the contraction treats these indices temporarily as different and lets the contraction be ruled by the contraction mechanism of the Kronecker delta's. In practise this means that the dimension will enter via $\delta_\mu^\mu \rightarrow \dim(\mu)$.

In FORM there are no upper and lower indices. Of course the user can emulate those. The contract statement always assumes that there is a proper distribution of upper and lower indices if the user decided to work in a metric in which this makes a difference. Note however that due to the fact that the Levi-Civita tensor is considered to be imaginary, there is usually no need to do anything special. This is explained in the chapter on functions (see 6).

There are several options to control which contractions will be taken. They are

| | |
|---|---|
| Contract; | Here only a single pair of Levi-Civita tensors will be contracted. The pair that is selected by FORM is the pair that will give the smallest number of terms in their contraction. |
| Contract <number>; | This tells FORM to keep contracting pairs of Levi-Civita tensors until there are <number> or <number>+1 Levi-Civita tensors left. A common example is<br>Contract 0;<br>which will contract as many pairs as possible. |
| Contract:<number>; | Here the number indicates the number of indices in the Levi-Civita tensors to be contracted. Only a single pair will be contracted and it will be the pair that gives the smallest number of terms. |
| Contract:<number> <number>; | The First number refers to the number of indices in the Levi-Civita tensors to be contracted. The second number refers to the number of Levi-Civita tensors that should be left (if possible) after contraction. |

Note that the order in which FORM selects the contractions is by looking at which pair will give the smallest number of terms. This means that usually the largest buildup of terms is at the end. This is not always the case, because there can be a complicated network of contracted indices.

## 5.14 ctable

| | |
|---|---|
| Type | Declaration statement |
| Syntax | ctable <options> <table to be declared>; |
| See also | functions (5.36), table (5.100), ntable (5.66) |

This statement declares a commuting table and is identical to the table command (see 5.100) which has the commuting property as its default.

## 5.15 ctensors

| | |
|---|---|
| Type | Declaration statement |
| Syntax | ct[ensors] <list of tensors to be declared>; |
| See also | functions (5.36), tensors (5.102), ntensors (5.67) |

This statement declares commuting tensors. It is equal to the tensor statement (see 5.102) which has the commuting property as its default.

## 5.16 cyclesymmetrize

| | |
|---|---|
| Type | Executable statement |
| Syntax | cy[clesymmetrize] {<name of function/tensor> [<argument specifications>];} |
| See also | symmetrize (5.99), antisymmetrize (5.3), rcyclesymmetrize (5.82) |

The argument specifications are explained in the section on the symmetrize statements (see 5.99).

The action of this stament is to cycle-symmetrize the (specified) arguments of the functions that are mentioned. This means that the arguments are brought to 'natural order' in the notation of FORM by trying cyclic permutations of the arguments or groups of arguments. The 'natural order' may depend on the order of declaration of the variables.

## 5.17 delete

| | |
|---|---|
| Type | Specification statement |
| Syntax | delete storage; |
| See also | save (5.87), load (5.51) |

This statement clears the complete storage file and reduces it to zero size. The effect is that all stored expressions are removed from the system. Because it is impossible to remove individual expressions from the store file (there is no mechanism to fill the resulting holes) it is the only way to clean up the storage file. If some expressions should be excluded from this elimination process, one should copy them first into active global expressions, then delete the storage file, after which the expressions can be written to storage again with a .store instruction.

## 5.18 dimension

| | |
|---|---|
| Type | Declaration statement |
| Syntax | d[imension] <number or symbol>; |
| See also | index (5.46) |

Sets the default dimension. This default dimension determines the dimension of the indices that are being declared without dimension specification as well as the dimension of all dummy indices. At the moment an index is declared and there is no dimension specification, FORM looks for the default dimension and uses that. This index will then have this dimension, even when the default dimension is changed at a later moment. The dummy indices always have the dimension of the current default dimension. If the default dimension is changed the dimension of all dummy indices changes with it. Varieties:

| | |
|---|---|
| Dimension <number>; | Declares the number to be the default dimension. |
| Dimension <symbol>; | Symbol must be the name of a symbol, either previously declared or declarable because of an auto-declaration (see 5.6). Declares the symbol to be the default dimension. |

| | |
|---|---|
| Dimension <symbol>:<symbol>; | The symbols must be the names of symbols, either previously declared or declarable because of an auto-declaration (see 5.6). The first symbol will be the default dimension. The second symbol will be the the first symbol minus 4. It will be used as such in the trace contractions. See also 5.108 and 5.46. |

Examples:

```
Dimension 3;
Dimension n;
Dimension n:[n-4];
```

The default dimension in FORM is 4.

## 5.19 discard

| | |
|---|---|
| Type | Executable statement |
| Syntax | dis[card]; |

This statement discards the current term. It can be very useful in statements of the type

```
if ( count(x,1) > 5 ) Discard;
```

which eliminates all terms that have more than five powers of x.

## 5.20 disorder

| | |
|---|---|
| Type | Executable statement |
| Syntax | disorder <pattern> = <expression>; |
| See also | identify (5.41) |

This statement is identical to the disorder option of the id statement (see 5.41). It is just a shorthand notation for 'id disorder'.

## 5.21 drop

| | |
|---|---|
| Type | Specification statement |
| Syntax | drop; |
| | drop <list of expressions>; |
| See also | ndrop (5.60) |

In the first variety this statement eliminates all expressions from the system. In the second variety it eliminates only the expressions that are mentioned from the system. All expressions that are to be dropped can still be used in the r.h.s. of other expressions inside the current module. Basically the expressions to be dropped are not treated for execution and after the module has finished completely they are removed. See also the ndrop statement 5.60.

## 5.22 else

| | |
|---|---|
| Type | Executable statement |
| Syntax | else; |
| See also | if (5.44), elseif (5.23), endif (5.25) |

To be used in combination with an if statement (see 5.44). The statements following the else statement until the matching endif statement (see 5.25) will be executed for the current term if the conditions of the matching proceeding if statement and/or all corresponding elseif statements (see 5.23) are false. If any of the conditions of the matching proceeding if or elseif statements are true the statements following the else statement will be skipped.

## 5.23 elseif

| | |
|---|---|
| Type | Executable statement |
| Syntax | elseif ( <condition> ); |
| See also | if (5.44), else (5.22), endif (5.25) |

Should be proceeded by an if statement (see 5.44) and followed at least by a matching endif statement (see 5.25). If the conditions of the proceeding matching if statement and all proceeding matching elseif statements are false the condition of this elseif statement will be evaluated. If it is true, the statements following it until the next matching elseif, else or endif statement will be executed. If not, control is passed to this next elseif, else or endif statement. The syntax for the condition is exactly the same as for the condition in the if statement.

## 5.24 endargument

| | |
|---|---|
| Type | Executable statement |
| Syntax | endargument; |
| See also | argument (5.5) |

Terminates an argument environment (see 5.5). The argument statement and its corresponding endargument statement must belong to the same module. Argument environments can be nested with all other environments.

## 5.25 endif

| | |
|---|---|
| Type | Executable statement |
| Syntax | endif; |
| See also | if (5.44), elseif (5.23), else (5.22) |

Terminates an if construction (see 5.44, 5.23 and 5.22). If should be noted that if constructions can be nested.

## 5.26 endinside

| | |
|---|---|
| Type | Executable statement |
| Syntax | endinside; |
| See also | inside (5.48) and the chapter on $-variables (4) |

Terminates an 'inside' environment (see 5.48) which is used to operate on the contents of $-variables (see 4).

## 5.27 endrepeat

| | |
|---|---|
| Type | Executable statement |
| Syntax | endrepeat; |
| See also | repeat (5.85), while (5.114) |

Ends the repeat endvironment. The repeat environment is started with a repeat statement (see 5.85). The repeat and its matching endrepeat should be inside the same module. Repeat environments can be nested with all other environments (and other repeat environments).

## 5.28 endterm

| | |
|---|---|
| Type | Executable statement |
| Syntax | endterm; |
| See also | term (5.103), sort (5.93) |

Terminates a term environment (see 5.103). Term environments can be nested with other term environments and with other environments in general. The whole environment should be part of one single module. See also 5.93.

## 5.29 endwhile

| | |
|---|---|
| Type | Executable statement |
| Syntax | endwhile; |
| See also | while (5.114), repeat (5.85) |

Terminates a while environment (see 5.114). The while statement and its corresponding endwhile statement must be part of the same module.

## 5.30 exit

| | |
|---|---|
| Type | Executable statement |
| Syntax | exit ["<string>"]; |
| See also | setexitflag (5.90) |

Causes execution to be aborted immediately. The string will be printed in the output. This can be used to indicate where FORM ran into the exit statement.

## 5.31 factarg

| | |
|---|---|
| Type | Executable statement |
| Syntax | factarg options {<name of function/set> [<argument specifications>]}; |
| See also | splitarg (5.94) |

Splits the indicated function arguments into individual factors. The argument specifications are as in the splitarg statement (see 5.94). There are a few extra options:

(0) Eliminates the coefficient of the term in the argument. Similar to Normalize,(0),....

(1) The coefficient of the term and its sign are pulled out separately.

(-1) The coefficient is pulled out with its sign.

In the case of the above options only the coefficient is treated. When these options are not used the whole term is treated as in:

```
Symbols a,b,c;
CFunctions f,f1,f2,f3;
Local F = f(-3*a*b)+f(3*a*b)
         +f1(-3*a*b)+f1(3*a*b)
         +f2(-3*a*b)+f2(3*a*b)
         +f3(-3*a*b)+f3(3*a*b);
FactArg,f;
Factarg,(0),f1;
Factarg,(1),f2;
Factarg,(-1),f3;
Print;
.end
```

```
F =
   f(a,b,-1,3) + f(a,b,3) + 2*f1(a*b) + f2(a*b,-1,3) + f2(a*b,3)
   + f3(a*b,-3) + f3(a*b,3);
```

## 5.32 fill

| | |
|---|---|
| Type | Declaration statement |
| Syntax | fill <tableelement> = <expression> [,<moreexpressions>]; |
| See also | table (5.100), fillexpression (5.33), printtable (5.78) |

The standard way to define elements of a table. In the left hand side one specifies the table element without the extra function arguments that could potentially occur (see 5.100). In the right hand side one specifies what the table element should be substituted by. Example:

```
Table tab(1:2,1:2,x?);
Fill tab(1,1) = x+y;
Fill tab(2,1) = (x+y)^2;
Fill tab(1,2) = tab(1,1)+y;
Fill tab(2,2) = tab(2,1)+y^2;
```

The first fill statement is a bit like a continuous attempt to try the substitution

```
id  tab(1,1,x?) = x+y;
```

The last two fill statements show that one could use the table recursively. If a real loop occurs the program may terminate due to stack overflow.
It is possible to define several table elements in one statement. In that case the various elements are separated by commas. The last index is the first one to be raised. This means that in the above example one could have written:

```
Table tab(1:2,1:2,x?);
Fill tab(1,1) = x+y,tab(1,1)+y,(x+y)^2,tab(2,1)+y^2;
```

One warning is called for. One should avoid using expressions in the right hand side of fill statements:

```
Table B(1:1);
Local dummy = 1;
.sort
Fill B(1) = dummy;
Drop dummy;
.sort
Local F = B(1);
Print;
.end
```

In the example a crash will result, because when we use the table element the expression dummy doesn't exist anymore. In a fill statement the r.h.s. is not expanded. Hence it keeps the reference to the expression dummy. When the table element is used the reference to the expression dummy is inserted and expanded. Hence one obtains the contents of dummy that exist at the moment of use. This is illustrated in the following example:

```
Table B(1:1);
Local dummy = 1;
.sort
Fill B(1) = dummy;
.sort
Local F = B(1);
Print;
.sort
Drop;
.sort
Local dummy = 2;
.sort
Local F = B(1);
Print;
.end
```

The final value of F will be 2, not 1.
A way to get around this problem is to force the evaluation of the table definition by using dollar variables:

```
Table B(1:1);
Local dummy = 1;
.sort
#$value = dummy;
Fill B(1) = '$value';
Drop dummy;
.sort
Local F = B(1);
Print;
.end
```

Here we use the character representation of the contents of the dollar variable to obtain an expression that doesn't need any further evaluation. If we would put

```
fill B(1) = $value;
```

a reference to the dollar variable would be inserted and it would only be evaluated at use again. In principle this could cause similar problems.
Not dropping the expression dummy can sometimes give the correct result, but is potentially still unsafe.

```
Table B(1:1);
Local u = 2;
Local dummy = 1;
.sort
```

```
Fill B(1) = dummy;
Drop dummy;
.sort
Local v = 5;
Local F = B(1);
Print;
.end
```

Here the answer will be 5, because after u has been dropped the expressions will be renumbered. Hence now dummy becomes the first expression, and eventually v becomes the second expression. The references in the table elements are not renumbered. Hence the r.h.s. of B(1) keeps pointing at the second expression, which at the moment of application has the value 5. One can see now also why the original example crashes. First dummy was the first expression and at the moment of application F is the first (existing) expression. Hence the substitution of B(1) causes a self reference and hence an infinite loop. Eventually some buffer will overflow.

## 5.33 fillexpression

| | |
|---|---|
| Type | Declaration statement |
| Syntax | fillexpression <table> = <expression>(<x1>,...,<xn>); |
| | fillexpression <table> = <expression>(<funname>); |
| See also | table (5.100), fill (5.32) and the table_ function (6.53) |

Used to dynamically load a table during runtime. When there are n symbols (here called x1 to xn) it is assumed that the table is n-dimensional. The expression must previously have been bracketted in these symbols and each of the brackets has the effect of a fill statement in which the powers of the x1 to xn refer to the table elements. Brackets that do not have a corresponding table element are skipped.
In the case that only a function name is specified the arguments of the function refer to the table elements.

## 5.34 fixindex

| | |
|---|---|
| Type | Declaration statement |
| Syntax | fi[xindex] {<number> <value>}; |
| See also | index (5.46) and chapter 10. |

Defines `d_(number,number) = value` in which number is the number of a fixed index (hence a positive short integer with a value less than ConstIndex (see 11). The value should be a short integer, i.e. its absolute value should be less than $2^{15}$ on 32 bit computers and less than $2^{31}$ on 64 bit computers. One can define more than one fixed index in one statement. Before one would like to solve problems involving the choice of a metric with this statement, one should consult the chapter on the use of a metric (chapter 10).

## 5.35 format

| | |
|---|---|
| Type | Output control statement |
| Syntax | fo[rmat] <option>; |
| See also | print (5.76) |

Controls the format for the printing of expressions. There is a variety of options.

<number> Output will be printed using the indicated number of characters per line. The default is 72. Numbers outside the range 1-255 are corrected to 72. Positive numbers less than 39 are corrected to 39.

float [<number>] Numbers are printed in floating point notation, even though internally they remain fractions. This is purely cosmetic. If no number is specified the precision of the output will be 10 digits. If a number is specified it indicates the number of digits to be used for the precision.

rational Output format is switched back to rational numbers (in contrast to floating point output). This is the default.

nospaces The output is printed without the spaces that make the output slightly more readable. This gives a more compact output.

spaces The output is printed with extra spaces between the terms and around certain operators to make it slightly more readable. This is the default.

fortran The output is printed in a way that is readable by a fortran compiler. This includes continuation characters and the splitting of the output into blocks of no more than 15 continuation lines. This number can be changed with the setup parameter ContinuationLines (see 11). In addition dotproducts are printed with the 'dotchar' in the place of the period between the vectors. This dotchar can be set in the setup file (see 11). Its default is the underscore character.

doublefortran Same as the fortran mode, but fractions are printed with double floating point numbers, because some compilers convert numbers like 1. into 1.E0. With this format FORM will force double precision by using 1.D0.

c Output will be C compatible. The exponent operator ($\wedge$) is represeted by the function pow. It is the responsibility of the user that this function will be properly defined. Dotproducts are printed with the 'dotchar' in the place of the period between the vectors. This dotchar can be set in the setup file (see 11). Its default is the underscore character.

maple Output will be as much as possible compatible with Maple format. It is not guaranteed that this is perfect.

mathematica Output will be as much as possible compatible with Mathematica format. It is not guaranteed that this is perfect.

reduce Output will be as much as possible compatible with Reduce format. It is not guaranteed that this is perfect.

The last few formats have not been tried out extensively. The author is open for suggestions.

## 5.36 functions

| | |
|---|---|
| Type | Declaration statement |
| Syntax | f[unctions] <list of functions to be declared>; |
| See also | cfunctions (5.8), tensors (5.102), ntensors (5.67),<br>table (5.100), ntable (5.66), ctable (5.14) |

Used to declare one or more functions. The functions declared with this statement will be noncommuting. For commuting functions one should use the cf[unctions] statement (see 5.8). Functions can have a number of properties that can be set in the declaration. This is done by appending the options to the name of the function. These options are:

| | |
|---|---|
| name#r | The function is considered to be a real function (default). |
| name#c | The function is considered to be a complex function. This means that internally two spaces are reserved. One for the variable name and one for its complex conjugate name#. |
| name#i | The function is considered to be imaginary. |
| name(s[ymmetric]) | The function is totally symmetric. This means that during normalization FORM will order the arguments according to its internal notion of order by trying permutations. The result will depend on the order of declaration of variables. |
| name(a[ntisymmetric]) | The function is totally antisymmetric. This means that during normalization FORM will order the arguments according to its internal notion of order and if the resulting permutation of arguments is odd the coefficient of the term will change sign. The order will depend on the order of declaration of variables. |
| name(c[yclesymmetric]) | The function is cycle symmetric in all its arguments. This means that during normalization FORM will order the arguments according to its internal notion of order by trying cyclic permutations. The result will depend on the order of declaration of variables. |
| name(r[cyclesymmetric)<br>name(r[cyclic])<br>name(r[eversecyclic]) | The function is reverse cycle symmetric in all its arguments. This means that during normalization FORM will order the arguments according to its internal notion of order by trying cyclic permutations and/or a complete reverse order of all arguments. The result will depend on the order of declaration of variables. |

The complexity properties and the symmetric properties can be combined. In that case the complexity properties should come first as in

```
Function f1#i(symmetric);
```

## 5.37 funpowers

| | |
|---|---|
| Type | Declaration statement |
| Syntax | funpowers <on/off>; |
| See also | on (5.71), off (5.70) |

This statement is obsolete. The user should try to use the funpowers option of the on (see 5.71) or the off (see 5.70) statements.

## 5.38 global

| | |
|---|---|
| Type | Definition statement |
| Syntax | g[lobal] <name> = <expression>; |
| | g[lobal] <names of expressions>; |
| See also | local (5.52) |

Used to define a global expression. A global expression is an expression that remains active until the first .store instruction. At that moment it is stored into the 'storage file' and stops being manipulated. After this it can still be used in the right hand side of expressions and id statements (see 5.42). Global expressions that have been put in the storage file can be saved to a disk file with the save statement (see 5.87) for use in later programs.

There are two versions of the global statement. In the first the expression is defined and filled with a right hand side expression. The left hand side and the right hand side are separated by an = sign. In this case the expression can have arguments which will serve as dummy arguments after the global expression has been stored with a .store instruction. Note that this use of arguments can often be circumvented with the replace_ function (see 6.45) as in

```
Global F(a,b) = (a+b)^2;
.store
Local FF = F(x,y);
Local GG = F*replace_(a,x,b,y);
```

because both definitions give the same result.

The second version of the global statement has no = sign and no right hand side. It can be used to change a local expression into a global expression.

## 5.39 goto

| | |
|---|---|
| Type | Executable statement |
| Syntax | go[to] <label>; |
| | go to <label>; |
| See also | label (5.50) |

Causes processing to proceed at the indicated label statement (see 5.50). This label statement must be in the same module.

## 5.40 hide

| | |
|---|---|
| Type | Specification statement |
| Syntax | hide; |
| | hide <list of expressions>; |
| See also | nhide (5.62), unhide (5.111), nunhide (5.68), pushhide (5.80), pophide (5.75) |

In the first variety this statement marks all currently active expressions for being put in hidden storage. In the second variety it marks only the specified active expressions as such.

If an expression is marked for being hidden, it will be copied to the 'hide file', a storage which is either in memory or on file depending on the combined size of all expressions being hidden. If this size exceeds the size of the setup parameter scratchsize (see 11) the storage will be on file. If it is less, the storage will be in memory. An expression that has been hidden is not affected by the statements in the modules as long as it remains hidden, but it can be used inside other expressions in the same way skipped expressions (see 5.91) or active expressions can be used. In particular all its bracket information (see 5.7) is retained and can be accessed, including possible bracket indexing.

The hide mechanism is particularly useful if an expression is not needed for a large number of modules. It has also advantages over the storing of global expressions after a .store instruction (see 2), because the substitution of global expressions is slower (name definitions may have changed and have to be checked) and also a possible bracket index is not maintained by the .store instruction.

Expressions can be returned from a hidden status into active expressions with the unhide statement (see 5.111). One might want to consult the nhide statement (5.40) as well.

When an expression is marked to be hidden it will remain such until execution starts in the current expression. When it is the turn of the expression to be executed, it is copied to the hide file instead.

Note that a .store instruction will simultaneously remove all expressions from the hide system.

## 5.41 identify

| | |
|---|---|
| Type | Executable statement |
| Syntax | id[entify] [<options>] <pattern> = <expression>; |
| See also | also (5.2), idnew (5.42), idold (5.43) |

The statement tries to match the pattern. If the pattern matches one or more times, it will be replaced by the expression in the r.h.s. taking the possible wildcard substitutions into account. For the description of the patterns, see chapter 3.
The options are

| | |
|---|---|
| multi | This option is for combinations of symbols and dotproducts only and it does not use wildcard powers. FORM determines how many times the pattern fits in one pattern matching action. Then the r.h.s. is substituted to that power. It is the default for these kinds of patterns. |

many — This is the default for patterns that contain other objects than symbols and dotproducts. The pattern is matched and taken out. Then FORM tries again to match the pattern in the remainder of the term. This is repeated until there is no further match. Then for each match the r.h.s. is substituted (with its own wildcard substitutions).

select — This option should be followed by one or more sets. After the sets the pattern can be specified. The pattern will only be substituted if none of the objects mentioned in the sets will be left after the pattern has been taken out. This holds only for objects 'at ground level'; i.e. the pattern matcher will not look inside function arguments for this. Note that this is a special case of the option 'only'.

once — The pattern is matched only once, even if it occurs more than once in the term. The first match that FORM encounters is taken. When wildcards are involved, this may depend on the order of declaration of variables. It could also be installation dependent. Also the setting of properorder (see 5.71 and 5.70) could be relevant. Try to write programs in such a way that the outcome does not depend on which match is taken.

only — The pattern will match only if there is an exact match in the powers of the symbols and dotproducts present.

ifmatch−> — This option should be followed by the name (or number) of a label. If the pattern matches, the replacement will be made after which the execution continues at the label.

disorder — This option is used for products of noncommuting functions or tensors. The match will only take place if the order of the functions in the match is different from what FORM would have made of it if the functions would be commuting. Hence if the functions in the term are in the order that FORM would give them if they would be commuting (which depends on the order of declaration) there will be no match. This can be rather handy when using wildcards as in `F(a?)*F(b?)`.

## 5.42 idnew

| | |
|---|---|
| Type | Executable statement |
| Syntax | idn[ew] [<options>] <pattern> = <expression>; |
| See also | identify (5.41), also (5.2), idold (5.43) |

This statement and its options are completely identical to the regular id or identify statement (see 5.41).

## 5.43 idold

| | |
|---|---|
| Type | Executable statement |
| Syntax | ido[ld] [<options>] <pattern> = <expression>; |
| See also | identify (5.41), also (5.2), idnew (5.42) |

This statement and its options are completely identical to the regular also statement (see 5.2). The options are described with the id or identify statement (see 5.41).

## 5.44 if

| | |
|---|---|
| Type | Executable statement |
| Syntax | if ( <condition> ); |
| | if ( <condition> ) <executable statement> |
| See also | elseif (5.23), else (5.22), endif (5.25) |

Used for executing parts of code only when certain conditions are met. Works together with the else statement (see 5.22), the elseif statement (see 5.23) and the endif statement (see 5.25). There are two versions. In the first the if statement must be accompanied by at least an endif statement. In that case the statements between the if statement and the endif statement will be executed if the condition is met. It is also possible to use elseif and else statements to be more flexible. This is done in the same way as in almost all computer languages.
In the second form the if statement does not terminate with a semicolon. It is followed by a single regular statement. No endif statement should be used. The single statement will be executed if the condition is met.
The condition in the if statement should be enclosed by parentheses. Its primary components are:

count() Returns an integer power counting value for the current term. Should have arguments that come in pairs. The first element of the pair is a variable. The second is its integer weight. The types of variables that are allowed are symbols, dotproducts, functions, tensors, tables and vectors. The weights can be positive as well as negative. They have to be short integers (Absolute value $< 2^{15}$ on 32 bit computers and $< 2^{31}$ on 64 bit computers). The vectors can have several options appended to their name. This is done by putting a + after the name of the vector and have this followed by one or more of the following letters:

| | |
|---|---|
| v | Loose vectors with an index are taken into account. |
| d | Vectors inside dotproducts are taken into account. |
| f | Vectors inside tensors are taken into account. |
| ?set | The set should be a set of functions. Vectors inside the functions that are members of the set are taken into account. It is assumed that those functions are linear in the given vector |

When no options are specified the result is identical to +vfd.

match() The argument of the match condition can be any left hand side of an id statement, including options as once, only, multi, many and select (see 5.42). The id of the id statement should not be included. FORM will invoke the pattern matcher and see how many times the pattern matches. This number is returned. In the case of once or only this is of course at most one.

expression() — The argument(s) of this condition is/are a list of expressions. In the case that the current term belongs to any of the given expressions the return value is 1. If it does not belong to any of the given expressions the return value is 0.

findloop() — The arguments are as in the replaceloop statement (see 5.86) with the exception of the outfun which should be omitted. If FORM detects an index loop in the current term that fulfils the specified conditions the return value is 1. It is 0 otherwise.

multipleof() — The argument should be a positive integer. This object is to be compared with a number (could be obtained from a condition) and if this number is an integer multiple of the argument there will be a match. If should be obvious that such a compare only makes sense for the == and != operators.

<integer> — To be compared either with another number, the result of a condition or a multipleof object.

coefficient — Represents the coefficient of the current term.

$-variable — Will be evaluated at runtime when the if statement is encountered. Should evaluate into a numerical value. If it does not, an error will result.

All the above primary components result in numerical objects. Such objects can be compared to each other in structures of the type <obj1> <operator> <obj2>. The result of such a compare is either true (or 1) or false (or 0). The operators are:

| | |
|---|---|
| > | Results in true if object 1 is greater than object 2. |
| < | Results in true if object 1 is less than object 2. |
| = | Same as ==. |
| == | Results in true if both objects have the same value. |
| >= | Results in true if object 1 is greater than or equal to object 2. |
| <= | Results in true if object 1 is less than or equal to object 2. |
| ! = | Results in true if object 1 does not have the same value as object 2. |

If the condition for true is not met, false is returned. Several of the above compares can be combined with logical operators. For this it is necessary to enclose the above compares within parentheses. This forces FORM to interprete the hierarchy of the operators properly. The extra logical operators are

| | |
|---|---|
| \|\| | The or operation. True if at least one of the objects 1 and 2 is true (or nonzero). False or zero if both are false or zero. |
| && | The and operation. True if both the objects 1 and 2 are true (or nonzero). False or zero if at least one is false or zero. |

Example:

```
if ( ( match(f(1,x)*g(?a)) && ( count(x,1,v+d,1) == 3 ) )
     || ( expression(F1,F2) == 0 ) );
    some statements
endif;
if ( ( ( match(f(1,x)*g(?a)) == 0 ) && ( count(x,1,v+d,1) == 3 ) )
     || expression(F1,F2) );
    some statements
endif;
```

We see that `match()` is equivalent to `( match() != 0 )` and something similar for `expression()`. This shorthand notation can make a program slightly more readable.

**Warning!** The if-statement knows only logical values as the result of operations. Hence the answer to anything that contains parenthesis (which counts as the evaluation of an expression) is either true (1) or false (0). Hence the object (5) evaluates to true.

## 5.45 ifmatch

| | |
|---|---|
| Type | Executable statement |
| Syntax | ifmatch− > <label> <pattern> = <expression>; |
| See also | identify (5.41) |

This statement is identical to the ifmatch option of the id statement (see 5.41). Hence

```
ifmatch-> ....
```

is just a shorthand notation for

```
id ifmatch-> ....
```

## 5.46 index, indices

| | |
|---|---|
| Type | Declaration statement |
| Syntax | i[ndex] <list of indices to be declared>; |
| | i[ndices] <list of indices to be declared>; |
| See also | dimension (5.18), fixindex (5.34) |

Declares one or more indices. In the declaration of an index one can specify its dimension. This is done by appending one or two options to the name of the index to be declared:

name=dim — The dimension is either a nonnegative integer or a previously declared symbol. If the dimension is zero this means that no dimension is attached to the index. The consequence is that the index cannot be summed over and index contractions are not performed for this index. If no dimension is specified the default dimension will be assumed (see the dimension statement 5.18).

name=dim:ext — The dimension is a symbol as above. Ext is an extra symbol which indicates the value of dim-4. This option is useful when traces over gamma functions are considered (see 5.107 and 5.108).

## 5.47 insidefirst

| | |
|---|---|
| Type | Declaration statement |
| Syntax | insidefirst <on/off>; |
| See also | on (5.71), off (5.70) |

This statement is obsolete. The user should try to use the insidefirst option of the on (see 5.71) or the off (see 5.70) statements.

## 5.48 inside

| | |
|---|---|
| Type | Executable statement |
| Syntax | inside <list of $-variables>; |
| See also | endinside (5.26) and the chapter on $-variables (4) |

works a bit like the argument statement (see 5.5) but with $-variables instead of with functions. An inside statement should be paired with an endinside statement (see 5.26) inside the same module. The statements inbetween will then be executed on the contents of the $-variables that are mentioned. One should pay some attention to the order of the action. The $-variables are treated sequentially. Hence, after the first one has been treated its contents are substituted by the new value. Then the second one is treated. If it uses the contents of the first variable, it will use the new value. If the first variable uses the contents of the second variable it will use its old value. Redefining any of the listed $-variables in the range of the 'inside-environment' is very dangerous. It is not specified what FORM will do. Most likely it will be unpleasant.

## 5.49 keep

| | |
|---|---|
| Type | Specification statement |
| Syntax | keep brackets; |
| | |
| See also | bracket (5.7), antibracket (5.1) and the chapter on brackets (7) |

The effect of this statements is that during execution of the current module the contents of the brackets are not considered. The statements only act on the 'outside' of the brackets. Only when the terms are considered finished and are ready for the sorting are they multiplied by the contents of the brackets. At times this can save much computer time as complicated pattern matching and multiplications of function arguments with large fractions have to be done only once, rather than for each complete term separately (assuming that each bracket contains a large number of terms). There can be some nasty side effects. Assume an expression like:

```
F = f(i1,x)*(g(i1,y)+g(i1,z));
B  f;
.sort
Keep Brackets;
sum i1;
```

the result will be

```
F = f(N1_?,x)*g(i1,y)+f(N1_?,x)*g(i1,z));
```

because at the moment of summing over i1 FORM is not looking inside the brackets and hence it never sees the second occurrence of i1. There are some beneficial applications of this statement in the 'mincer' package that comes with the FORM distribution. In this package the most costly step was made faster by a significant factor (depending on the problem) due to the keep brackets statement.

## 5.50 label

| | |
|---|---|
| Type | Executable statement |
| Syntax | la[bel] <name of label>; |
| See also | goto (5.39) |

Places a label at the current location. The name of the label can be any name or number. Control can be transfered to the position of the label by a goto statement (see 5.39) or the ifmatch option of an id statement (see 5.41). The only condition is that the goto statement and the label must be inside the same module. Once the module is terminated all existing labels are forgotten. This means that in a later module a label with the same name can be used again (this may not improve readability though but it is a good thing when third party libraries are used).

## 5.51 load

| | |
|---|---|
| Type | Declaration statement |
| Syntax | loa[d] <filename> [<list of expressions>]; |
| See also | save (5.87), delete (5.17) |

Loads a previously saved file (see 5.87). If no expressions are specified all expressions in the file are put in the storage file and obtain the status of stored global expressions. If a list of expressions is specified all those expressions are loaded and possible other expressions are ignored. If a specified expression is not present, an error will result. If one does not know exactly what expressions are present in a file one could load the file without a list of expressions, because FORM will list all expressions that it encountered.

## 5.52 local

| | |
|---|---|
| Type | Definition statement |
| Syntax | l[ocal] <name> = <expression>; |
| | l[ocal] <names of expressions>; |
| See also | global (5.38) |

Used to define a local expression. A local expression is an expression that will be dropped when a .store instruction is encountered. If this is not what is intended one should use global expressions (see 5.38). The statement can also be used to change the status of a global expression into that of a local expression. In that case there is no = sign and no right hand side.

## 5.53 makeinteger

| | |
|---|---|
| Type | Executable statement |
| Syntax | makeinteger [<argument specifications>] {<name of function/set> [<argument specifications>]}; |
| See also | normalize (5.63) |

Normalizes the indicated argument of the indicated functions(s) in such a way that all terms in this argument have integer coefficients with a their greatest common divider being one. This still leaves the possibility that the first term of this argument may be negative. If this is not desired one can first normalize the argument and then make its coefficients integer. The overal factor that is needed to make the coefficients like described is taken from the overal factor of the complete term. Example

```
S   a,b,c;
CF  f;
L   F = f(22/3*a+14/5*b+18/7*c);
MakeInteger,f;
Print +f;
.end
```

```
F =
   2/105*f(135*c + 147*b + 385*a);
```

Note that this feature can be used to make outputs look much more friendly. It can be used in combination with the AntiBracket statement (5.1) and the function `dum_` (6.12) to imitate a smart extra level of brackets and make outputs shorter.

## 5.54 many

| | |
|---|---|
| Type | Executable statement |
| Syntax | many <pattern> = <expression>; |
| See also | identify (5.41) |

This statement is identical to the many option of the id statement (see 5.41). Hence

```
many ....
```

is just a shorthand notation for

```
id many ....
```

## 5.55 metric

| | |
|---|---|
| Type | Declaration statement |
| Syntax | metric <option>; |

Remark: statement is inactive. Should have no effect.

## 5.56 moduleoption

| | |
|---|---|
| Type | Module control statement |
| Syntax | moduleoption <option>[,<value>]; |
| See also | polyfun (5.74), slavepatchsize (5.92) |

Used to set a mode for just the current module. It overrides the normal setting and will revert to this setting after this module. The settings are:

| | |
|---|---|
| parallel | Allows parallel execution of the current module if all other conditions are right. This is the default. |
| noparallel | Vetoes parallel execution of the current module. |
| polyfun | Possibly followed by the name of a 'polyfun'. Is similar to the polyfun statement (see 5.74) but only valid for the current module. |
| slavepatchsize | Followed by a number. Similar to the slavepatchsize statement (see 5.92) but only valid for the current module. |
| nokeep | Should be followed by a list of $-variables. Indicates that the contents of the indicated $-variables are not relevant once the module has been finished and neither is the term by term order in which the $-variables obtain their value. |
| maximum | Should be followed by a list of $-variables. Indicates that of the contents of the indicated $-variables the maximum is the only thing that is relevant once the module has been finished. The term by term order in which the $-variables obtain their value is not relevant. |
| minimum | Should be followed by a list of $-variables. Indicates that of the contents of the indicated $-variables the minimum is the only thing that is relevant once the module has been finished. The term by term order in which the $-variables obtain their value is not relevant. |
| sum | Should be followed by a list of $-variables. Indicates that the indicated $-variables are representing a sum. The term by term order in which the $-variables obtain their value is not relevant. |

The options 'nokeep', 'maximum', 'minimum' and 'sum' are for parallel versions of FORM. The presence of $-variables can be a problem when the order of processing of the terms is not well defined. These options tell FORM what these $-variables are used for. In the above cases FORM can take the appropriate action when gathering information from the various processors. This will allow parallel execution of the current module. If $-variables are used in a module and they are defined on a term by term basis, the normal action of FORM will be to veto parallel execution unless it is clear that no confusion can occur. See also the chapter on the parallel version(12).

## 5.57 modulus

| | |
|---|---|
| Type | Declaration statement |
| Syntax | m[odulus] <value>; |

Defines all calculus to be modules the given number, provided this number is positive. If this number is less than the (installation dependend but at least 10000) maximum power for symbols and dotproducts the powers of symbols and dotproducts are reduced with the relation $x^{value} = x$. The modulus calculus extends itself to fractions. This means that if the value is not a prime number division by zero could result. It is the responsibility of the user to avoid such problems.
When the value in the modulus statement is either 0 or 1 the statement would be meaningless. It is used as a signal to FORM that modulus calculus should be switched off again.
The reduction of powers with the relation $x^{value} = x$ can be switched off by specifying a negative number for value. In that case only coefficients are treated. Of course the value used for the modulus will be the absolute value of the what was specified. Hence

```
Modulus,-17;
```

will set all arithmetic to modulus 17 and powers will not be affected.
When the value for the modulus arithmetic is a prime number there exists another option. In the statement

```
Modulus p:x;
```

in which p is a prime number and x a generator (a number such that all numbers mod p that are not zero can be written as powers of x) FORM will construct a table to express all nonzero numbers less than p as powers of x. This will fail if either x is not a generator or when the numbers are too large for the construction of such a table. It then uses this table when printing results and express all numbers as powers of the generator.

## 5.58 multi

| | |
|---|---|
| Type | Executable statement |
| Syntax | multi <pattern> = <expression>; |
| See also | identify (5.41) |

This statement is identical to the multi option of the id statement (see 5.41). Hence

```
multi ....
```

is just a shorthand notation for

```
id multi ....
```

## 5.59 multiply

| | |
|---|---|
| Type | Executable statement |
| Syntax | mu[ltiply] [<option>] <expression>; |

Statement multiplies all terms by the given expression. It is advisable to use the options when noncommuting variables are involved. They are:

left Multiplication is from the left.

right Multiplication is from the right.

There is no guarantee as to what the default is with respect to multiplication from the left or from the right. It is up to FORM to decide what it considers to be most efficient when neither option is present.

Note that one should not abbreviate this command to 'multi', because there is a separate multi command (see 5.58).

## 5.60 ndrop

| | |
|---|---|
| Type | Specification statement |
| Syntax | ndrop; |
| | ndrop <list of expressions>; |
| See also | drop (5.21) |

In the first variety this statement cancels all drop plans. This means that all expressions scheduled for being dropped will be restored to their previous status of local or global expressions. In the second variety this happens only to the expressions that are specified. Example:

```
Drop;
Ndrop F1,F2;
```

This drops all expressions, except for the expressions `F1` and `F2`.

## 5.61 nfunctions

| | |
|---|---|
| Type | Declaration statement |
| Syntax | n[functions] <list of functions to be declared>; |
| See also | functions (5.36), cfunctions (5.8) |

This statement declares noncommuting tensors. It is equal to the function statement (see 5.36) which has the noncommuting property as its default.

## 5.62 nhide

| | |
|---|---|
| Type | Specification statement |
| Syntax | nhide; |
| | nhide <list of expressions>; |
| See also | hide (5.40), unhide (5.111), nunhide (5.68), pushhide (5.80), pophide (5.75) |

In its first variety this statement undoes all hide plans that exist thus far in the current module. In the second variety it does this only for the specified active expressions. See the hide statement in 5.40. Example:

```
Hide;
Nhide F1,F2;
```

Here all active expressions will be transferred to the hide file except for the expressions `F1` and `F2`.

## 5.63 normalize

| | |
|---|---|
| Type | Executable statement |
| Syntax | normalize options {<name of function/set> [<argument specifications>]}; |
| See also | argument (5.5), splitarg (5.94), makeinteger (5.53) |

Normalizes the indicated arguments of the indicated functions. Normalization means that the argument will be multiplied by the inverse of its coefficient (provided it is not zero). This holds for single term arguments. For multiple term arguments the inverse of the coefficient of the first term of the argument is used. The options and the argument specifications are as in the SplitArg statement (see 5.94). Under normal circumstances the coefficient that is removed from the argument(s) is multiplied into the coefficient of the term. This can be avoid with the extra option `(0)`. Hence

| | |
|---|---|
| Normalize,f; | changes `f(2*x+3*y)` into `2*f(x+3/2*y)` but |
| Normalize,(0),f; | changes `f(2*x+3*y)` into `f(x+3/2*y)`. |

## 5.64 nprint

| | |
|---|---|
| Type | Output control statement |
| Syntax | np[rint] <list of names of expressions>; |
| See also | print (5.76) |

Statement is used to take expressions from the list of expressions to be printed. When a print statement is used (see 5.76) without specification of expressions, all active expressions are marked for printing. With this statement one can remove a number of them from the list.

## 5.65 nskip

| | |
|---|---|
| Type | Specification statement |
| Syntax | nskip; |
| | nskip <list of expressions>; |
| See also | skip (5.91) |

In the first variety it causes the cancellation of all skip plans (see 5.91) for expressions. The status of these expressions is restored to their previous status (active local or global expressions). In the second variety this is done for the specified expressions only. Example:

```
Skip;
Nskip F1,F2;
```

This causes all active expressions to be skipped except for the expressions `F1` and `F2`.

## 5.66 ntable

| | |
|---|---|
| Type | Declaration statement |
| Syntax | ntable <options> <table to be declared>; |
| See also | functions (5.36), table (5.100), ctable (5.14) |

This statement declares a noncommuting table. For the rest it is identical to the table command (see 5.100) which has the commuting property as its default.

## 5.67 ntensors

| | |
|---|---|
| Type | Declaration statement |
| Syntax | nt[ensors] <list of tensors to be declared>; |
| See also | functions (5.36), tensors (5.102), ctensors (5.15) |

This statement declares noncommuting tensors. For the rest it is equal to the tensor statement (see 5.102) which has the commuting property as its default.
The options that exist for properties of tensors are the same as those for functions (see 5.36).

## 5.68 nunhide

| | |
|---|---|
| Type | Specification statement |
| Syntax | nunhide; |
| | nunhide <list of expressions>; |
| See also | hide (5.40), nhide (5.62), unhide (5.111), pushhide (5.80), pophide (5.75) |

In its first variety this statement undoes all unhide (see 5.111 and 5.40) plans that the system has in the current module. In its second variety this happens only with the specified expressions. Example:

```
Unhide;
Nunhide F1,F2;
```

All expressions are taken from the hide system, except for the expressions `F1` and `F2`.

## 5.69 nwrite

| | |
|---|---|
| Type | Declaration statement |
| Syntax | nw[rite] <keyword>; |
| See also | on (5.71), off (5.70) |

This statement is considered obsolete. All its varieties have been taken over by the off statement (see 5.70) and the on statement (see 5.71). The current version of FORM will still recognize it, but the user is advised to avoid its usage. In future versions of FORM it is scheduled to be used for a different kind of writing and hence its syntax may change considerably. The conversion program conv2to3 should help in the conversion of existing programs. For completeness we still give the syntax and how it should be converted. The keywords are:

| | |
|---|---|
| stats | Same as: Off stats; |
| statistics | Same as: Off statistics; |
| shortstats | Same as: Off shortstats; |
| shortstatistics | Same as: Off shortstatistics; |
| warnings | Same as: Off warnings; |
| allwarnings | Same as: Off allwarnings; |
| setup | Same as: Off setup; |
| names | Same as: Off names; |
| allnames | Same as: Off allnames; |
| shortstats | Same as: Off shortstats; |
| highfirst | Same as: Off highfirst; |
| lowfirst | Same as: Off lowfirst; |
| powerfirst | Same as: Off powerfirst; |

## 5.70 off

| | |
|---|---|
| Type | Declaration statement |
| Syntax | off <keyword>; |
| | off <keyword> <option>; |
| See also | on (5.71) |

New statement to control settings during execution. Many of these settings replace older statements. The settings and their keywords are:

| | |
|---|---|
| compress | Turns compression mode off. |
| insidefirst | Not active at the moment. |
| propercount | Turns the propercounting mode off. This means that for the generated terms in the statistics not only the 'ground level' terms are counted but also terms that were generated inside function arguments. |
| stats | Same as 'Off statistics'. |
| statistics | Turns off the printing of statistics. |
| shortstats | Same as 'Off shortstatistics'. |
| shortstatistics | Takes the writing of the statistics back from shorthand mode to the regular statistics mode in which each statistics messages takes three lines of text and one blank line. |
| names | Turns the names mode off. This is the default. |
| allnames | Turns the allnames mode off. The default. |
| warnings | Turns off the printing of warnings. |
| allwarnings | Turns off the printing of all warnings. |
| highfirst | Puts the sorting in a low first mode. |
| lowfirst | Leaves the default low first mode and puts the sorting in a high first mode. |
| powerfirst | Puts the sorting back into 'highfirst' mode. |
| setup | Switches off the mode in which the setup parameters are printed. This is the default. |
| properorder | Turns the properorder mode off. This is the default. |
| parallel | Disallows the running of the program in parallel mode (only relevant for parallel versions of FORM). |

If a description is too short, one should also consult the description in the on statement (see 5.71).

## 5.71 on

| | |
|---|---|
| Type | Declaration statement |
| Syntax | on <keyword>; |
| | on <keyword> <option>; |
| See also | off (5.70) |

New statement to control settings during execution. Many of these settings replace older statements. The settings and their keywords are:

| | |
|---|---|
| compress | Turns compression mode on. This compression is a relatively simple compression that hardly costs extra computer time but saves roughly a factor two in diskstorage. The old statement was ‘compress on’ but this should be avoided in the future. This setting is the default. |
| insidefirst | Not active at the moment. |
| propercount | Sets the counting of the terms during generation into ‘propercount’ mode. This means that only terms at the ‘ground level’ are counted and terms inside functions arguments are not counted in the statistics. This setting is the default. |
| stats | Same as ‘On statistics’. |
| statistics | Turns the writing of runtime statistics on. This is the default. It is possible to change this default with one of the setup parameters in the setup file (see 11). |
| shortstats | Same as ‘On shortstatistics’. |
| shortstatistics | Puts the writing of the statistics in a shorthand mode in which the complete statistics are written on a single line only. |
| names | Turns on the mode in which at the end of each module the names of all variables that have been defined by the user are printed. This is an inspection mode for debugging by the user. Default is off. |
| allnames | Same as ‘On names’ but additionally all system variables are printed as well. Default is off. |
| warnings | Turns on the printing of warnings in regular mode. This is the default. |
| allwarnings | Puts the printing of warnings in a mode in which all warnings, even the very unimportant warnings are printed. |
| highfirst | In this mode polynomials are sorted in a way that high powers come before low powers. |
| lowfirst | In this mode polynomials are sorted in a way that low powers come before high powers. This is the default. |
| powerfirst | In this mode polynomials are sorted in a way that high powers come before low powers. The most relevant is however the combined power of all symbols. |
| setup | Causes the printing of the current setup parameters for inspection. Default is off. |
| properorder | Turns the properorder mode on. The default is off. In the properorder mode FORM pays particular attention to function arguments when bringing terms and expressions to normal form. This may cost a considerable amount of extra time. In normal mode FORM is a bit sloppy (and much faster) about this, resulting sometimes in an ordering that appears without logic. This concerns only function arguments! This mode is mainly intended for the few moments in which the proper ordering is important. |
| parallel | Allows the running of the program in parallel mode unless other problems prevent this. This is of course only relevant for parallel versions of FORM. The default is on. |

## 5.72 once

| | |
|---|---|
| Type | Executable statement |
| Syntax | once <pattern> = <expression>; |
| See also | identify (5.41) |

This statement is identical to the once option of the id statement (see 5.41). Hence

```
once ....
```

is just a shorthand notation for

```
id once ....
```

## 5.73 only

| | |
|---|---|
| Type | Executable statement |
| Syntax | only <pattern> = <expression>; |
| See also | identify (5.41) |

This statement is identical to the only option of the id statement (see 5.41). Hence

```
only ....
```

is just a shorthand notation for

```
id only ....
```

## 5.74 polyfun

| | |
|---|---|
| Type | Declaration statement |
| Syntax | polyfun <name of function>; |
| | polyfun; |
| See also | moduleoption (5.56) |

Declares the specified function to be the 'polyfun'. The polyfun is a function of which the single argument is considered to be the coefficient of the term. If two terms are otherwise identical the arguments of their polyfun will be added during the sorting, even if these arguments are little expressions. Hence

```
PolyFun acc;
Local F = 3*x^2*acc(1+y+y^2)+2*x^2*acc(1-y+y^2);
```

will result in

```
F = x^2*acc(5+y+5*y^2);
```

Note that the external numerical coefficient is also pulled inside the polyfun.
If the polyfun statement has no argument, FORM reverts to its default mode in which no polyfun exists. This does not change any terms. If one would like to remove the polyfun from the terms one has to do that 'manually' as in

```
PolyFun;
id  acc(x?) = x;
```

## 5.75 pophide

| | |
|---|---|
| Type | Specification statement |
| Syntax | pophide; |
| See also | hide (5.40), nhide (5.62), unhide (5.111), nunhide (5.68), pushhide (5.80) |

Undoes the action of the most recent pushhide statement (see 5.80). If there is no matching pushhide statement an error will result.

## 5.76 print

| | |
|---|---|
| Type | Print statement |
| Syntax | Print [<options>]; |
| | Print {[<options>] <expression>}; |
| | Print [<options>] "<format string>" [<objects>]; |
| See also | print[] (5.77), nprint (5.64), printtable (5.78) |

General purpose print statement. Has three modes. In the first two modes flags are set for the printing of expressions after the current module has been finished. The third mode concerns printing during execution. This allows the printing of indivudual terms or $-variables on a term by term basis. It should be considered as a useful debugging device.
In the first mode all active expressions are scheduled for printing. The options are

+f   Printing will be only to the log file.

-f   Printing will be both to the screen and to the log file. This is the default.

+s   Each term will start a new line. This is called the single term mode.

-s   Regular term mode. There can be more terms in a line. Linebreaks are placed when the line is full. The line size is set in the format statement (see 5.35). This is the default.

In the second mode one can specify individual expressions to be printed. The options hold for all the expressions that follow them until new options are specified. The options are the same as for the first mode.
In the third mode there is a format string as for the printf command in the C programming language. Of course the control characters are not exactly the same as for the C language because the objects are different. The special characters are:

%t The current term will be printed at this position including its sign, even if this is a plus sign.

%T The current term will be printed at this position. If its coeficient is positive no leading plus sign is printed.

%$ A dollar expression will be printed at this position. The name(s) of the dollar expression(s) should follow the format string in the order in which they are used in the format string.

%% The character %.

% If this is the last character of the string no linefeed will be printed at the end of the print command.

\n A linefeed.

Each call is terminated with a linefeed. Example:

```
    Symbols a,b,c;
    Local F = 3*a+2*b;
    Print "> %T";
    id  a = b+c;
    Print ">> %t";
    Print;
    .end
> 3*a
>>  + 3*b
>>  + 3*c
> 2*b
>>  + 2*b

   F =
      5*b + 3*c;
```

The options in the third mode are identical to those of the first mode. Because of the mixed nature of this statement it can occur in more than one location in the module.

## 5.77 print[]

| | |
|---|---|
| Type | Output control statement |
| Syntax | print[] {[<options>] <name>}; |
| See also | print (5.76) |

Print statement to cause the printing of expressions at the end of the current module. Is like the first two modes of the regular print statement (see 5.76), but when printing FORM does not print the contents of each bracket, only the number of terms inside the bracket. Is to be used in combination with a bracket or an antibracket statement (see 5.7 and 5.1). Apart from this the options are identical to those of the first two modes of the print statement.

## 5.78 printtable

| | |
|---|---|
| Type | Print statement |
| Syntax | printtable [<options>] <tablename>; |
| | printtable [<options>] <tablename> > <filename>; |
| | printtable [<options>] <tablename> >> <filename>; |
| See also | print (5.76), table (5.100), fill (5.32), fillexpression (5.33) and the table_ function (6.53) |

Almost the opposite of a FillExpression statement (see 5.33). Prints the contents of a table according to the current format (see 5.35). The output can go to standard output, the logfile or a specified file. The elements of the table that have been defined and filled are written in the form of fill statements (see 5.32) in such a way that they can be read in a future program to fill the table with the current contents. This is especially useful when the fillexpression statement has been used to dynamically extend tables based on what FORM has encountered during running. This way those elements will not have to be computed again in future programs.

The options are

| | |
|---|---|
| +f | Output is to the logfile and not to the screen. |
| -f | Output is both to the logfile and to the screen. This is the default. |
| +s | Output will be in a mode in which each new term starts a new line. |
| -s | Output will be in the regular mode in which new terms continue to be written on the same line within the limits of the number of characters per line as set in the format statement. Default is 72 characters per line. This is the default. |

If redirection to a file is specified output will be only to this file. The +f option will be ignored. There are two possibilities:

| | |
|---|---|
| > filename | The old contents of the file with name 'filename' will be overwritten. |
| >> filename | The table will be appended to the file with the name 'filename'. This allows the writing of more than one table to a file. |

## 5.79 propercount

| | |
|---|---|
| Type | Declaration statement |
| Syntax | propercount <on/off>; |
| See also | on (5.71), off (5.70) |

This statement is obsolete. The user should try to use the propercount option of the on (see 5.71) or the off (see 5.70) statements.

## 5.80 pushhide

| | |
|---|---|
| Type | Specification statement |
| Syntax | pushhide; |
| See also | hide (5.40), nhide (5.62), unhide (5.111), nunhide (5.68), pophide (5.75) |

Hides all currently active expressions (see 5.40). The pophide statement (see 5.75) can bring them back to active status again.

## 5.81 ratio

| | |
|---|---|
| Type | Executable statement |
| Syntax | ratio <symbol1> <symbol2> <symbol3>; |

This statement can be used for limited but fast partial fractioning. In the statement

```
ratio a,b,c;
```

in which `a`, `b` and `c` should be three symbols FORM will assume that $c = b - a$ and then make the substitutions

$$\begin{aligned}
\frac{1}{a^m}\frac{1}{b^n} &= \sum_{i=0}^{m-1}(-1)^i\binom{n-1+i}{n-1}\frac{1}{a^{m-i}}\frac{1}{c^{n+i}} + \sum_{i=0}^{n-1}(-1)^m\binom{m-1+i}{m-1}\frac{1}{b^{n-i}}\frac{1}{c^{m+i}} \\
\frac{b^n}{a^m} &= \sum_{i=0}^{n}\binom{n}{i}\frac{c^i}{a^{m-n+i}} \qquad m \geq n \\
\frac{b^n}{a^m} &= \sum_{i=0}^{m-1}\binom{n}{i}\frac{c^{n-i}}{a^{m-i}} + \sum_{i=0}^{n-m}\binom{m-1+i}{m-1}c^i b^{n-m-i} \qquad m < n
\end{aligned}$$

Of course, such substitutions can be made also by the user in a more flexible way. This statement has however the advantage of the best speed.

Actually the ratio statement is a leftover from the Schoonschip inheritance. For most simple partial fractioning one could use

```
repeat id 1/[x+a]/[x+b] = (1/[x+a]-1/[x+b])/[b-a];
repeat id [x+a]/[x+b] = 1-[b-a]/[x+b];
repeat id [x+b]/[x+a] = 1+[b-a]/[x+a];
```

or similar constructions. This does not give the speed of the binomials, but it does make the program more readable and it is much more flexible.

## 5.82 rcyclesymmetrize

| | |
|---|---|
| Type | Executable statement |
| Syntax | rc[yclesymmetrize] {<name of function/tensor> [<argument specifications>];} |
| See also | symmetrize (5.99), cyclesymmetrize (5.16), antisymmetrize (5.3) |

The argument specifications are explained in the section on the symmetrize statement (see 5.99).

The action of this stament is to reverse-cycle-symmetrize the (specified) arguments of the functions that are mentioned. This means that the arguments are brought to 'natural order' in the notation of FORM by trying cyclic and reverse cyclic permutations of the arguments or groups of arguments. The 'natural order' may depend on the order of declaration of the variables.

## 5.83 redefine

| | |
|---|---|
| Type | Executable statement |
| Syntax | r[edefine] <preprocessor variable> "<string>"; |
| See also | preprocessor variables in the chapter on the preprocessor (1) |

This statement can be used to change the contents of preprocessor variables. The new contents can be used after the current module has finished execution and the preprocessor becomes active again for further translation and compilation. This termwise adaptation of the value of a preprocessor variable can be very useful in setting up multi module loops until a certain condition is not met any longer. Example:

```
#do i = 1,1
   statements;
   if ( condition ) redefine i "0";
   .sort
#enddo
```

As long as there is a term that fulfils the condition the loop will continue. This defines effectively a while loop (see 5.114) over various modules. Note that the .sort instruction is essential. Note also that a construction like

```
if ( count(x,1) > 3 ) redefine i "`i'+1";
```

is probably not going to do what the user intends. It is not going to count terms with more than three powers of x. The preprocessor will insert the compile time value of the preprocessor variable i. If this is 0, then each time a term has more than three powers of x, i will get the string value `0+1`. If one would like to do such counting, one should use a dollar variable (see 4).

## 5.84 renumber

| | |
|---|---|
| Type | Executable statement |
| Syntax | renumber <number>; |
| See also | sum (5.97) |

Renumbers the dummy indices. Dummy indices are indices of the type `N1_?`. Normally FORM tries to renumber these indices to make the internal representation of a term 'minimal'. It does not try exhaustively though. Especially interference with symmetric or antisymmetric functions is far from perfect. This is due to considerations of economy. With the renumber statement the user can force FORM to do better. The allowable options are:

0 All exchanges of one pair of dummy indices are tried until all pair exchanges yield no improvements. This is the default if no option is specified.

1 If there are N sets of dummy indices all N! permutations are tried. This can be very costly when a large number of indices is involved. Use with care!

## 5.85 repeat

| | |
|---|---|
| Type | Executable statement |
| Syntax | repeat; |
| | repeat <executable statement> |
| See also | endrepeat (5.27), while (5.114) |

The repeat statement starts a repeat environment. It is terminated with an endrepeat statement (see 5.27). The repeat statement and its matching endrepeat statement should be inside the same module.

The staments inside the repeat environment should all be executable statements (or print statements) and if any of the executable statements inside the environment has changed the current term, the action of the endrepeat statement will be to bring control back to the beginning of the environment. In that sense the repeat/endrepeat combination acts as

```
do
   executable statements
while any action due to any of the statements
```

The second form of the statement is a shorthand notation:

```
repeat;
   single statement;
endrepeat;
```

is equivalent to

```
repeat single statement;
```

Particular attention should be given to avoid infinite loops as in

```
repeat id a = a+1;
```

A more complicated infinite loop is

```
repeat;
   id  S(x1?)*R(x2?) = T(x1,x2,x2-x1);
   id  T(x1?,x2?,x3?pos_) = T(x1,x2-2,x3-1)*X(x2);
   id  T(x1?,x2?,x3?) = S(x1)*R(x2);
endrepeat;
```

If the current term is S(2)*R(2), the statements in the loop do not change it in the end. Yet the program goes into an infinite loop, because the first id statement will change the term (action) and the third statement will change it back. FORM does not check that the term is the same again. Hence there is action inside the repeat environment and hence the statements will be executed again. This kind of hidden action is a major source of premature terminations of FORM programs.

Repeat environments can be nested with all other environments (and of course also with other repeat/endrepeat combinations).

## 5.86 replaceloop

| | |
|---|---|
| Type | Executable statement |
| Syntax | replaceloop <parameters>; |
| See also | the findloop option of the if statement (5.44) |

This statement causes the substitution of index loops. An index loop is a sequence of contracted indices in which the indices are arguments of various instances of the same function and each contracted index occurs once in one instance of the function and once in another instance of the function. Such a contraction defines a connection and if a number of such connections between occurrences of the function form a loop this structure is a candidate for replacement. Examples of such loops are:

```
f(i1,i2,j1)*f(i2,i1,j2)
f(i1,i2,j1)*f(i2,i3,j2)*f(i1,i3,j3)
f(i1,k1,i2,j1)*f(k2,i2,i3,j2)*f(i1,k3,i3,j3)
```

The first term has a loop of two functions or vertices and the other two terms each define a loop of three vertices. The parameters are:

| | |
|---|---|
| <name> | The name of the function that defines the 'vertices'. This must always be the first parameter. |
| arguments=number | Only occurrences of the vertex function with the specified number of arguments will be considered. The specification of this parameter is mandatory. |
| loopsize=number | Only a loop with this number of vertices will be considered. |
| loopsize=all | All loop sizes will be considered and the smallest loop is substituted. |
| loopsize<number | Only loops with fewer vertices than 'number' will be considered and the smallest looop will be substituted. |
| outfun=<name> | Name of an output function in which the remaining arguments of all the vertex functions will be given. This parameter is mandatory. |
| include-<name> | Name of a summable index that must be one of the links in the loop. This parameter is optional. |

The loopsize parameter is mandatory. Hence one of its options must be specified. The order of the parameters is not important. The only important thing is that the name of the vertex function must be first. The names of the keywords may be abbreviated as in

```
ReplaceLoop f,a=3,l=all,o=ff,i=i2;
```

although this does not improve the readability of the program. Hence a more readable abbreviated version might be

```
ReplaceLoop f,arg=3,loop=all,out=ff,inc=i2;
```

The action of the statement is to remove the vertex functions that constitute the loop and replace them by the output function. This outfun will have the arguments of all the vertex functions minus the contracted indices that define the loop. The order of the arguments is the order in which they are encountered when following the loop. The order of the arguments in the outfun depends however on the order in which FORM encounters the vertices. Hence the outfun will often be cyclesymmetric (see 5.36 and 5.16). If FORM has to exchange indices to make a 'proper loop' (i.e. giving relevance to first index as if it is something incoming and the second index as if it is something outgoing) and if the vertex function is antisymmetric, each exchange will result in a minus sign. Examples:

```
Functions f(antisymmetric),ff(cyclesymmetric);
Indices i1,...,i8;
Local F = f(i1,i4,i2)*f(i5,i2,i3)*f(i3,i1,i6)*f(i4,i7,i8);
ReplaceLoop f,arg=3,loop=3,out=ff;
```

would result in

```
-f(i4,i7,i8)*ff(i4,i5,i6)
```

and

```
Functions f(antisymmetric),ff(cyclesymmetric);
Indices i1,...,i9;
Local F = f(i1,i4,i2)*f(i5,i2,i3)*f(i3,i1,i6)*f(i4,i7,i8)
          *f(i6,i7,i8);
ReplaceLoop f,arg=3,loop=all,out=ff;
```

would give

```
-f(i1,i4,i2)*f(i5,i2,i3)*f(i3,i1,i6)*ff(i4,i6)
```

because the smallest loop will be taken. A number of examples can be found in the package 'color' for group theory invariants that is part of the FORM distribution.
A related object is the findloop option of the if statement (see 5.44). This option just probes whether a loop is present but makes no replacements.

## 5.87 save

| | |
|---|---|
| Type | Declaration statement |
| Syntax | sa[ve] <filename> [<names of global expressions>]; |
| See also | load (5.51) |

Saves the contents of the store file (all global expressions that were stored in .store instructions) to a file with the indicated name. If a list of expressions is provided only those expressions are saved

and the others are ignored. Currently the contents of the save file are still architecture dependent. This will be changed in the future.
Together with the load statement (see 5.51) the save statement provides a mechanism to transfer data in internal notation from one program to another. It is the prefered method to keep results from a lengthy job for further analysis without needed the long initial running time.
In order to avoid confusion .sav is the preferred extension of saved files.

## 5.88 select

| | |
|---|---|
| Type | Executable statement |
| Syntax | select <list of sets> <pattern> = <expression>; |
| See also | identify (5.41) |

This statement is identical to the select option of the id statement (see 5.41). Hence

```
select ....
```

is just a shorthand notation for

```
id select ....
```

## 5.89 set

| | |
|---|---|
| Type | Declaration statement |
| Syntax | set <set to be declared>:<element> [<more elements>]; |

Declares a single set and specifies its elements. Sets have a type of variables connected to them. There can be sets of symbols, sets of functions, sets of vectors, sets of indices and sets of numbers. For the purpose of sets tensors and tables counts as functions.
There can also be mixed sets of indices and numbers. When a number could be either a fixed index or just a number FORM will keep the type of the set unfixed. This can change either when the next element is a symbolic index or a number that cannot be a fixed index (like a negative number). If the status does not get resolved the set can be used in the wildcarding of both symbols and indices. Normally sets of numbers can be used only in the wildcarding of symbols.

## 5.90 setexitflag

| | |
|---|---|
| Type | Executable statement |
| Syntax | setexitflag; |
| See also | exit (5.30) |

Causes termination of the program after execution of the current module has finished.

## 5.91 skip

| | |
|---|---|
| Type | Specification statement |
| Syntax | skip; |
| | skip <list of expressions>; |
| See also | nskip (5.65) |

In the first variety this statement marks all active expressions that are in existence at the moment this statement is compiled, to be skipped. In the second variety this is done only to the active expressions that are specified. If an expression is skipped in a given module, the statements in the module have no effect on it. Also it will not be sorted again at the end of the module. This means that any bracket information (see 5.7) in the expression remains the way it was. Consult also the nskip statement in 5.65.

Skipped expressions can be used in the expressions in the r.h.s. of id statements (see 5.41), in multiply expressions (see 5.59), etc.

## 5.92 slavepatchsize

| | |
|---|---|
| Type | Declaration statement |
| Syntax | slavepatchsize <value>; |
| See also | moduleoption (5.56) |

Sets the size of the buffer for sending terms to the secondary processors in the parallel version of FORM. In the sequential version this statement is ignored. If the value of slavepatchsize is less than the maximum term size (see 11) it will be readjusted to the value of maxtermsize.

## 5.93 sort

| | |
|---|---|
| Type | Executable statement |
| Syntax | sort; |
| See also | term (5.103), endterm (5.28) |

Statement to be used inside the term environment (see 5.103 and 5.28). It forces a sort in the same way as a .sort instruction forces a sort for entire expressions.

## 5.94 splitarg

| | |
|---|---|
| Type | Executable statement |
| Syntax | splitarg options {<name of function/set> [<argument specifications>]}; |
| See also | splitfirstarg (5.95), splitlastarg (5.96), factarg (5.31) |

Takes the indicated argument of a function and if such an argument is a subexpression that consists on more than one term, all terms become single arguments of the function as in

```
f(a+b-5*c*d) --> f(a,b,-5*c*d)
```

The way arguments are indicated is rather similar to the way this is done in the argument statement (see 5.5). One can however indicate only a single group of functions in one statement. Additionally there are other options. All options are in the order that they should be specified:

| | |
|---|---|
| (term) | Only terms that are a numerical multiple of the given term are split off. The terms that are split off will trail the remainder. |
| ((term)) | Only terms that contain the given term will be split off. The terms that are split off will trail the remainder. |

The statement is terminated with a sequence of functions or sets of functions. The splitting action will apply only to the specified functions or to members of the set(s). If no functions or sets of functions are specified all functions will be treated, including the built in functions.
The argument spacifications consist of a list of numbers, indicating the arguments that should be treated. If no arguments are specified, all arguments will be treated.

## 5.95 splitfirstarg

| | |
|---|---|
| Type | Executable statement |
| Syntax | splitfirstarg {<name of function/set> [<argument specifications>]}; |
| See also | splitarg (5.94), splitlastarg (5.96) |

A little bit like the SplitArg statement (see 5.94). Splits the given argument(s) into its first term and a remainder. Then replaces the argument by the remainder, followed by the first term.
The statement is terminated with a sequence of functions or sets of functions. The splitting action will apply only to the specified functions or to members of the set(s). If no functions or sets of functions are specified all functions will be treated, including the built in functions.
The argument spacifications consist of a list of numbers, indicating the arguments that should be treated. If no arguments are specified all arguments will be treated.

## 5.96 splitlastarg

| | |
|---|---|
| Type | Executable statement |
| Syntax | splitlastarg {<name of function/set> [<argument specifications>]}; |
| See also | splitarg (5.94), splitfirstarg (5.95) |

A little bit like the SplitArg statement (see 5.94). Splits the given argument(s) into its last term and a remainder. Then replaces the argument by the remainder, followed by the last term.
The statement is terminated with a sequence of functions or sets of functions. The splitting action will apply only to the specified functions or to members of the set(s). If no functions or sets of functions are specified all functions will be treated, including the built in functions.

The argument spacifications consist of a list of numbers, indicating the arguments that should be treated. If no arguments are specified all arguments will be treated.

## 5.97 sum

| | |
|---|---|
| Type | Executable statement |
| Syntax | sum <list of indices>; |
| See also | renumber (5.84) |

The given indices will be summed over. There are two varieties. In the first the index is followed by a sequence of nonnegative short integers. In that case the summation means that for each of the integers a new instance of the term is created in which the index is replaced by that integer. In the second variety the index is either the last object in the statement or followed by another index. In that case the index is replaced by an internal dummy index of the type `N1_?` (or with another number instead of the 1). Such indices have the current default dimension and can be renamed at will by FORM to bring terms into standard notation. For example:

```
f(N2_?,N1_?)*g(N2_?,N1_?)
```

will be changed into

```
f(N1_?,N2_?)*g(N1_?,N2_?).
```

The user can use these dummy indices in the left hand side of id-statements.

## 5.98 symbols

| | |
|---|---|
| Type | Declaration statement |
| Syntax | s[ymbols] <list of symbols to be declared>; |

Declares one or more symbols. Each symbol can be followed by a number of options. These are (assuming that x is the symbol to be declared):

| | |
|---|---|
| x#r | The symbol is real. This is the default. |
| x#c | The symbol is complex. This means that two spaces are reserved for this symbol, one for x and one for x#(the complex conjugate). |
| x#i | The symbol is imaginary. |
| x(:5) | The symbol has the maximum power 5. This means that $x^6$ and higher powers are automatically eliminated during the normalization of a term. Of course any other number, positive or negative, is allowed. |
| x(-3:) | The symbol has the minimum power -3. This means that $x^{-4}$ and lower powers are automatically eliminated during the normalization of a term. Of course any other number, positive or negative, is allowed. Note that when the minimum power is positive, terms that have no power of x should technically be eliminated, but FORM will not do so. Such an action can be achieved at any moment with a combination of the count option of an if statement (see 5.44) and a discard statement (see 5.19). |
| x(-3:5) | The combination of a maximum and a minimum power restriction (see above). |

Complexity properties and power restrictions can be combined. In that case the complexity properties come first and then the power restrictions.

## 5.99 symmetrize

| | |
|---|---|
| Type | Executable statement |
| Syntax | symm[etrize] {<name of function/tensor> [<argument specifications>];} |
| See also | antisymmetrize (5.3), cyclesymmetrize (5.16), rcyclesymmetrize (5.82) |

The arguments consist of the name of a function (or a tensor), possibly foloowed by some specifications. Hence we have the following varieties:

| | |
|---|---|
| <name> | The function is symmetrized in all its arguments. |
| <name><numbers> | The function is symmetrized in the arguments that are mentioned. If there are fewer arguments than the highest number mentioned in the list or arguments, no symmetrization will take place. |
| <name>:<number> | Only functions with the specified number of arguments will be considered. Note: the number should follow the colon directly without intermediate space or comma. |
| <name>:<number><numbers> | If there is a number immediately following the colon, only functions with exactly that number of arguments will be considered. If the list of arguments contains numbers greater than this number, they will be ignored. If no number follows the colon directly, this indicates that symmetrization will take place, no matter the number of arguments of the function. If the list of arguments has numbers greater than the number of arguments of the function, these numbers will be ignored. |
| <name><br><(groups of numbers)> | The groups are specified as lists of numbers of arguments between parenthesis. All groups must have the same number of arguments or there will be a compile error. The groups are symmetrized as groups. The arguments do not have to be adjacent. Neither do they have to be ordered. The symmetrizeation takes place in a way that the first elements of the groups are most significant, etc. If any argument number is greater than the number of arguments of the function, no symmetrization will take place. |

| | |
|---|---|
| <name>:<number> <(groups of numbers)> | The groups are specified as lists of numbers of arguments between parenthesis. All groups must have the same number of arguments or there will be a compile error. The groups are symmetrized as groups. The arguments do not have to be adjacent. Neither do they have to be ordered. The symmetrizeation takes place in a way that the first elements of the groups are most significant, etc. If no number follows the colon directly symmetrization takes place no matter the number of arguments of the function. Groups that contain a number that is greater than the number of arguments of the function will be ignored. If a number follows the colon directly, only functions with that number of arguments will be symmetrized. Again, groups that contain a number that is greater than the number of arguments of the function will be ignored. |

The action of this stament is to symmetrize the (specified) arguments of the functions that are mentioned. This means that the arguments are brought to 'natural order' in the notation of FORM by trying permutations of the arguments or groups of arguments. The 'natural order' may depend on the order of declaration of the variables.

Examples:

```
Symmetrize Fun;
Symmetrize Fun 1,2,4;
Symmetrize Fun:5;
Symmetrize Fun: 1,2,4;
Symmetrize Fun:5 1,2,4;
Symmetrize Fun (1,6),(7,3),(5,2);
Symmetrize Fun:8 (1,6),(7,3),(5,2);
Symmetrize Fun: (1,6),(7,3),(5,2);
```

## 5.100 table

| | |
|---|---|
| Type | Declaration statement |
| Syntax | table <options> <table to be declared>; |
| See also | functions (5.36), ctable (5.14), ntable (5.66), fill (5.32) |

The statement declares a single table. A table is a very special instance of a function. Hence it can be either commuting or noncommuting. The table statement declares its function to be commuting. A noncommuting table is declared with the ntable statement (se 5.66). A table has a number of table indices (at least one!) and after that it can have a number of regular function arguments with or without wildcarding. The table indices can come in two varieties: matrix like or sparse. In the case of a matrix like table, for each of the indices a range has to be specified. FORM then reserves a location for each of the potential elements. For a sparse table one only specifies the number of indices. Sparse tables take less space, but they require more time searching whether an element has been defined. For a matrix like table FORM can look directly whether an element has been defined. Hence one has a tradeoff between space and speed.

Table elements are defined with the fill statement (see 5.32). Fill statements for table elements cannot be used before the table has been declared with a table or ntable statement.

When FORM encounters an unsubstituted table it will look for its indices. Then it can check whether the table element has been defined. If not, it can either complain (when the 'strict' option is used) or continue without substitution. Note that an unsubstituted table element is a rather expensive object as FORM will frequently check whether it can be substituted (new elements can be defined in a variety of ways....). If the indices match a defined table element, FORM will check whether the remaining arguments of the table will match the function-type arguments given in the table declaration in the same way regular function arguments are matched. Hence these arguments can contain wildcards and even argument field wildcards. If a match occurs, the table is replaced immediately.
The options are

| | |
|---|---|
| check | A check is executed on table boundaries. An element that is outside the table boundaries (regular matrix type tables only) will cause an error message and execution will be halted. |
| relax | Normally all elements of a table should be defined during execution and an undefined element will give an error message. The relax option switches this off and undefined elements will remain as if they are regular functions. |
| sparse | The table is considered to be sparse. In the case of a sparse table only the number of indices should be specified as ranges are not relevant. Each table element is stored separately. Searching for table elements is done via a balanced tree. This takes of course more time than the matrix type seach with is just by indexing. A matrix type table is the default. |
| strict | If this option is specified all table elements that are encountered during execition should be defined. An undefined table element will result in an error and execution is halted. Additionally all table elements should be properly defined at the end of the module in which the table has been defined. |
| zerofill | Any undefined table element is considered to be zero. |

The defaults are that the table is matrix-like and table elements that cannot be substituted will result in an error.

Ranges for indices in matrix-like tables are indicated with a semicolon as in

```
Symbol x;
Table t1(1:3,-2:4);
Table t2(0:3,0:3,x?);
Table sparse,t3(4);
```

The table `t1` is two dimensional and has 21 elements. The table `t2` is also two dimensional and has 16 elements. In addition there is an extra argument which can be anything that a wildcard symbol will match. The table `t3` is a sparse table with 4 indices.

If the computer on which FORM runs is a 32 bit computer no table can have more than $2^{15} = 32768$ elements. On a 64 bit computer the limit is $2^{31}$, but one should take into account that each element declared causes some overhead.

If the wildcarding in the declaration of a table involves the definition of a dollar variable (this is allowed! See 4) parallel execution of the entire remainder of the FORM program is switched off. This is of course only relevant for parallel versions of FORM. But if at all possible one should try to find better solutions than this use of dollar variables, allowing future parallel processing of the program.

## 5.101 tablebase

This statement is explained in the chapter on tablebases (8).

## 5.102 tensors

| | |
|---|---|
| Type | Declaration statement |
| Syntax | t[ensors] <list of tensors to be declared>; |
| See also | functions (5.36), ctensors (5.15), ntensors (5.67) |

A tensor is a special function that can have only indices for its arguments. If an index a contracted with the index of a vector SCHOONSCHIP notation is used. This means that the vector is written as a pseudo argument of the tensor. It should always be realized that in that case in principle the actual argument is a dummy index. Tensors come in two varieties: commuting and noncommuting. The tensor statement declares a tensor to be commuting. In order to declare a tensor to be noncommuting one should use the ntensor statement (see 5.67).
The options that exist for properties of tensors are the same as those for functions (see 5.36).

## 5.103 term

| | |
|---|---|
| Type | Executable statement |
| Syntax | term; |
| See also | endterm (5.28), sort (5.93) |

Begins the term environment. This environment is terminated with the endterm statement (see 5.28). The action is that temparily the current term is seen as a little expression by itself. The statements inside the environment are applied to it and one can even sort the results with the sort statement (see 5.93) which should not be confused with the .sort instruction that terminates a module. Inside the term environment one can have only executable statements and possibly term-wise print statements (see 5.76). When the end of the term environment is reached, the results are sorted (as would be done with an expression at the end of a module) and execution continues with the resulting terms. This environment can be nested.

## 5.104 testuse

| | |
|---|---|
| Type | Executable statement |
| Syntax | testuse ["<tablename(s)>"]; |
| See also | tablebases (8), testuse (8.10) |

This statement is explained in the chapter on tablebases.

## 5.105 totensor

| | |
|---|---|
| Type | Executable statement |
| Syntax | totensor [nosquare] [!<vector or set>] <vector> <tensor>; |
| | totensor [nosquare] [!<vector or set>] <tensor> <vector>; |
| See also | tovector (5.106) |

Looks for multiple occurrences of the given vector, either inside dotproducts, contracted with a tensor, as argument of a function or as loose a loose vector with an index. In all occurrences in which the vector has been contracted a dummy index is introduced to make the contraction apparent. Then all these vectors witheir indices are replaced by the specified tensor with all the indices of these vectors. To make this clearer:

$$p^{\mu_1}p^{\mu_2}p^{\mu_3} \rightarrow t^{\mu_1\mu_2\mu_3}$$

and hence

```
p.p1^2*f(p,p1)*p(mu)*tt(p1,p,p2,p)
```

gives after `totensor p,t;`

```
f(N1_?,p1)*tt(p1,N2_?,p2,N3_?)*t(p1,p1,mu,N1_?,N2_?,N3_?)
```

The options are

| | |
|---|---|
| nosquare | Dotproducts with twice the specified vector (square of the vector) are not taken into account. |
| !vector | Dotproducts involving the specified vector are not treated. |
| !set | The set should be a set of vectors. All dotproducts involving a vector of the set are not treated. |

## 5.106 tovector

| | |
|---|---|
| Type | Executable statement |
| Syntax | tovector <tensor> <vector>; |
| | tovector <vector> <tensor>; |
| See also | totensor (5.105) |

The opposite of the totensor statement. The tensor is replaced by a product of the given vectors, each with one of the indices of the tensor as in:

$$t^{\mu_1\mu_2\mu_3} \rightarrow p^{\mu_1}p^{\mu_2}p^{\mu_3}$$

## 5.107 trace4

| | |
|---|---|
| Type | Executable statement |
| Syntax | trace4 [<options>] <index>; |
| See also | tracen (5.108), chisholm (5.9), unittrace (5.110) and the chapter on gamma algebra (9) |

Takes the trace of the gamma matrices with the given trace line index. It assumes that the matrices are defined in four dimensions, henec it uses some relations that are only valid in four dimensions. For details about these relations and other methods used, consult chapter 9 on gamma matrices. The options are:

| | |
|---|---|
| contract | Try to use the Chisholm identity to eliminate this trace and contract it with other gamma matrices. See also 5.9. This is the default. |
| nocontract | Do not use the Chisholm identity to eliminate this trace and contract it with other gamma matrices. See also 5.9. |
| nosymmetrize | When using the Chisholm identity to eliminate this trace and contract it with other gamma matrices, do not do it in the symmetric fashion, but use the first contraction encountered. See also 5.9. |
| notrick | The final stage of trace taking, when all indices are different and there are no contractions with identical vectors, as well as no $\gamma_5$ matrices present, is done with n-dimensional methods, rather than with 4-dimensional tricks. *there is still a BUG here* |
| symmetrize | When using the Chisholm identity to eliminate this trace and contract it with other gamma matrices, try to do it in the symmetric fashion. See also 5.9. |
| trick | The final stage of trace taking, when all indices are different and there are no contractions with identical vectors is done using the 4-dimensional relation $\gamma^a\gamma^b\gamma^c = \epsilon^{abcd}\gamma_5\gamma^d + \gamma^a\delta^{bc} - \gamma^b\delta^{ac} + \gamma^c\delta^{ab}$ This gives a shorter result for long traces. |

## 5.108 tracen

| | |
|---|---|
| Type | Executable statement |
| Syntax | tracen <index>; |
| See also | trace4 (5.107), chisholm (5.9), unittrace (5.110) and the chapter on gamma algebra (9) |

Takes the trace of the gamma matrices with the spin line indicated by the index. It is assumed that the trace is over a symbolic number of dimensions. Hence no special 4-dimensional tricks are used. The presence of $\gamma_5$, $\gamma_6$ or $\gamma_7$ is not tolerated. When indices are contracted FORM will try to use the special symbol for the dimension$-4$ if it has been defined in the declaration of the index (see 5.46. This results in relatively compact expressions. For more details on the algorithm used, see chapter 9 on gamma matrices.

## 5.109 tryreplace

| | |
|---|---|
| Type | Executable statement |
| Syntax | tryreplace {<name> <replacement>}; |
| See also | the replace_ function (6.45) |

The list of potential replacements should be similar to the arguments of the replace_ function (see 6.45). FORM will make a copy of the current term, try the replacement and if the replacement results in a term which, by the internal ordering of FORM, comes before the current term, the current term is replaced by the new variety.

## 5.110 unittrace

| | |
|---|---|
| Type | Declaration statement |
| Syntax | u[nittrace] <value>; |
| See also | trace4 (5.107), tracen (5.108), chisholm (5.9) and the chapter on gamma algebra (9) |

Sets the value of the trace of the unit matrix in the Dirac algebra (i.e. the object `g1_(n)` for traceline `n`)). The parameter `value` can be either a short positive number or any symbol with the exception of `i_`. See also chapter 9.

## 5.111 unhide

| | |
|---|---|
| Type | Specification statement |
| Syntax | unhide; |
| | unhide <list of expressions>; |
| See also | hide (5.40), nhide (5.62), nunhide (5.68), pushhide (5.80), pophide (5.75) |

In its first variety this statement causes all statements in the hide file to become active expressions again. In its second variety only the specified expressions are taken from the hide system and become active again. An expression that is made active again can be manipulated again in the module in which the unhide statement occurs. For more information one should look at the hide statement in 5.40.

Note that if only a number of expressions is taken from the hide system, the hide file may be left with ‘holes’, i.e. space between the remaining expressions that contain no relevant information any longer. FORM contains no mechanism to use the space in these holes. Hence if space is at a premium and many holes develop one should unhide all expressions (this causes the hide system to be started from zero size again) and then send the relevant expressions back to the hide system.

## 5.112 vectors

| | |
|---|---|
| Type | Declaration statement |
| Syntax | v[ectors] <list of vectors to be declared>; |

Used for the declaration of vectors. Example:

```
Vectors p,q,q1,q2,q3;
```

## 5.113 write

| | |
|---|---|
| Type | Declaration statement |
| Syntax | w[rite] <keyword>; |
| See also | on (5.71), off (5.70) |

This statement is considered obsolete. All its varieties have been taken over by the on statement (see 5.71) and the off statement (see 5.70). The current version of FORM will still recognize it, but the user is advised to avoid its usage. In future versions of FORM it is scheduled to be used for a different kind of writing and hence its syntax may change considerably. The conversion program conv2to3 should help in the conversion of existing programs. For completeness we still give the syntax and how it should be converted. The keywords are:

| | |
|---|---|
| stats | Same as: On stats; |
| statistics | Same as: On statistics; |
| shortstats | Same as: On shortstats; |
| shortstatistics | Same as: On shortstatistics; |
| warnings | Same as: On warnings; |
| allwarnings | Same as: On allwarnings; |
| setup | Same as: On setup; |
| names | Same as: On names; |
| allnames | Same as: On allnames; |
| shortstats | Same as: On shortstats; |
| highfirst | Same as: On highfirst; |
| lowfirst | Same as: On lowfirst; |

| | |
|---|---|
| powerfirst | Same as: On powerfirst; |

## 5.114 while

| | |
|---|---|
| Type | Executable statement |
| Syntax | while ( condition ); |
| See also | endwhile (5.29), repeat (5.85), if (5.44) |

This statement starts the while environment. It should be paired with an endwhile statement (see 5.29) which terminates the while environment. The statements between the while and the endwhile statements will be executed as long as the condition is met. For the description of the condition one should consult the if statement (see 5.44). The while/endwhile combination is equivalent to the construction

```
repeat;
   if ( condition );


   endif;
endrepeat;
```

If only a single statement is inside the environment one can also use

```
while ( condition ) statement;
```

Of course one should try to avoid infinite loops. In order to maximize the speed of FORM not all internal stacks are protected and hence the result may be that FORM may crash. It is also possible that FORM may detect a shortage of buffer space and quit with an error message.

For each term for which execution reaches the endwhile statement, control is brought back to the while statement. For each term that reaches the while statement the condition is checked and if it is met, the statements inside the environment are executed again on this term. If the condition is not met, execution continues after the endwhile statement.

# Chapter 6

# Functions

Functions are objects that can have arguments. There exist several types of functions in FORM. First there is the distinction between commuting and noncommuting functions. Commuting functions commute with all other objects. This property is used by the normalization routines that bring terms into standard form. Noncommuting functions do not commute necessarily with other noncommuting functions. They do however commute with objects that are considered to be commuting, like symbols, vectors and commuting functions. Various instances of the same noncommuting function but with different arguments do not commute either.
The next subdivision of the category of functions is in regular functions, tensors and tables. Tensors are special functions that can have only indices or vectors for their arguments. If an argument is a vector, it is assumed that this vector is there as the result of an index contraction. Tables are functions with automatic substitution rules. A table must have at least one table index. Each time during normalization FORM will check whether an instance of a table can be substituted. This means that undefined table elements will slow the program down somewhat.
All the various types of functions are declared with their own declaration statements. These are described in the chapter for the statements (see chapter 5).

One of the useful properties of functions is the wildcarding of their arguments during pattern matching. The following argument wildcards are possible:

x? Here x is a symbol. This symbol can match either a symbol, any numerical argument, or a complete subexpression argument that is not vectorlike or indexlike.

i? Here i is an index. This index can match either an index, a vector (actually the dummy index of the vector that was contracted), or a complete subexpression that is vectorlike (again actually the contracted dummy index).

v? Here v is a vector. This vector can match either a vector or a complete subexpression that is vectorlike.

f? Here f is any functiontype. This function can match any function. It is the responsibility of the user to avoid problems in the righthandside if f happens to match a tensor.

?a This is an argument field wildcard. This can match a complete set of arguments. The set can be empty. Argument field wildcards have a name that starts with a question mark followed by a name. They do not have to be declared as there cannot be confusion.

In addition to the above syntax FORM knows a number of special functions with well defined properties. All these functions have a name that ends in an underscore. In addition the names of these built in objects are case insensitive. This means for instance that the factorial function can be refered to as `fac_`, `Fac_` or `FAC_` or whatever the user considers more readable. The built in functions are:

## 6.1 abs_

If one argument which is numerical it evaluates into the absolute value of the argument.

## 6.2 bernoulli_

If it has one nonzero integer argument n it evaluates into the n-th coefficient in the power series expansion of $x/(1-e^{-x})$.

## 6.3 binom_

binom_(n,i) $= n!/(i!(n-i)!)$. If the arguments are non integer or negative, no substitution is made.

## 6.4 conjg_

Currently not doing anything.

## 6.5 count_

Similar to the count object in the if statement (see 5.44). This function expects the same arguments as the count object and returns the corresponding count value for the current term.

## 6.6 d_

The kronecker delta. Should have two indices for arguments. Often indicated as $\delta^{\mu\nu}$. In automatic summation over the indices the d_ often vanishes again as in `d_(mu,nu)*p(mu)*q(nu)` $\rightarrow$ `p.q` and similar replacements.

## 6.7 dd_

This is a combinatorics function. The tensor dd_ with an even number of indices is equal to the totally symmetric tensor built up from products of kronecker delta's. Each term in this symmetric combination is normalized to one. In principle there are $n!/(2^{n/2}(n/2)!$ terms in this combination. The profit comes when some or all the indices are contracted with vectors and some of these vectors are identical. In that case FORM will use combinatorics to generate only different terms, each with the proper prefactor. This can result in great time and space savings.

## 6.8 delta_

If one argument and it is numerical the result is one if the argument is zero and zero otherwise. If two arguments, the result is one if the arguments are numerical and identical. If they are numerical and they differ the result is zero. In all other cases nothing is done.

## 6.9 deltap_

If one argument and it is numerical the result is zero if the argument is zero and one otherwise. If two arguments, the result is zero if the arguments are numerical and identical. If they are numerical and they differ the result is one. In all other cases nothing is done.

## 6.10 denom_

Internal function to describe denominators. Has a single argument. `den(a+b)` is printed as `1/(a+b)`.

## 6.11 distrib_

This is a combinatorics function. It should have at least five arguments. If we have

```
distrib_(type,n,f1,f2,x1,...,xm)
```

with type and n integers, f1 and f2 functions and then a number of arguments there can be action if $-2 \leq$ type $\leq 2$. The typical action is that the arguments `x1,...,xm` will be divided over the two functions in all possible ways. For each possibility a new term is generated. The relative order of the arguments is kept. If type is negative it is assumed that the collection of x-arguments is antisymmetric and hence the number of permutations needed to make the split will determine whether there will be a minus sign on the resulting term. When type is zero all possible divisions are generated. Hence there will be $2^m$ divisions. The second argument is then not relevant. If type is 1 or -1 the second parameter says that the first function should obtain n arguments. The remaining arguments go to the second function. If type is 2 or -2 the second function should obtain n arguments. Example:

```
Symbols x1,...,x4;
CFunctions f,f1,f2;
Local F = f(x1,...,x4);
id  f(?a) = distrib_(-1,2,f1,f2,?a);
Print +s;
.end

 F =
     + f1(x1,x2)*f2(x3,x4)
     - f1(x1,x3)*f2(x2,x4)
     + f1(x1,x4)*f2(x2,x3)
     + f1(x2,x3)*f2(x1,x4)
     - f1(x2,x4)*f2(x1,x3)
     + f1(x3,x4)*f2(x1,x2)
    ;
```

When adjacent x-arguments are identical FORM uses combinatorics to avoid generating more terms than necessary.

## 6.12 dum_

Special function for printing virtual brackets. `dum_(a+b)` is printed as `(a+b)`: the name of this function is not printed!

## 6.13 dummy_

For internal use only.

## 6.14 dummyten_

For internal use only.

## 6.15 e_

The Levi-Civita tensor. It is a totally antisymmetric tensor with well defined contraction rules (see 5.13).

## 6.16 exp_

Internal function with two arguments. Represents argument1 to the power argument2. Of course it is printed in the standard power notation.

## 6.17 fac_

The factorial function. If it has a single nonzero integer argument n it is replaced by n! but if the result is bigger than the maximum allowable number an error will result.

## 6.18 firstbracket_

In the case that there is a single argument and this single argument is the name of an expression, this function is replaced by the part that is outside brackets in the first term of the expression. If there are no brackets the function is replaced by one.

## 6.19 g5_

The $\gamma_5$ Dirac gamma matrix. We assume here that it anticommutes with the other Dirac gamma matrices. Anybody who does not like that should program private libraries (this should not be too difficult with the new cycle symmetric functions (see 5.36)). There should be a single index to indicate the spinline.

## 6.20 g6_

There should be a single index to indicate the spinline. As in Schoonschip we use $\gamma_6 = 1 + \gamma_5$.

## 6.21 g7_

There should be a single index to indicate the spinline. As in Schoonschip we use $\gamma_7 = 1 - \gamma_5$.

## 6.22 g_

The Dirac gamma matrix. Its first argument should be an index (either symbolic or numeric). Then follow zero, one or more indices to indicate a string of gamma matrices that belong together. Gamma matrices with the same first index are considered to belong together, but as long as the indices are symbolic no assumptions are made about whether they go together or not. Hence no commutation or anticommutation properties are applied for different spin lines unless the spinline indices are both numeric.

## 6.23 gcd_

`gcd_(x1,x2)` is replaced by the greatest common divisor of the two integers `x1` and `x2`. If the arguments are not integers, nothing is done.

## 6.24 gi_

The unit Dirac gamma matrix. Should have a single index to indicate its spin line. Its is identical to a regular gamma matrix with no Lorenz indices: `gi_(n) = g_(n)`

## 6.25 integer_

This is a rounding function. It should have either one or two arguments. If there is a single argument and it is numeric, it will be rounded down to become an integer. If there are two arguments of which the first is numeric and the second is either 1, 0 or -1, the result will be the rounded value of the first argument. If the second argument is 1, the rounding will be down, when it is -1, the rounding will be up and when it is zero the rounding will be towards zero. In all other cases nothing is done.

## 6.26 invfac_

One divided by the factorial function. If it has a single nonzero integer argument n it is replaced by 1/n! but if this results in a number bigger than the maximum allowable number an error will result.

## 6.27 match_

Currently not active. Replaced automatically by 1.

## 6.28 max_

If all its arguments are numeric, this function returns the maximum value of these arguments.

## 6.29 maxpowerof_

If this function has a single argument that is a symbol, it returns the maximum power restriction of this symbol. If none was given it will be the installation dependent value MAXPOWER which is 10000 on 32 bit machines.

## 6.30 min_

If all its arguments are numeric, this function returns the minimum value of these arguments.

## 6.31 minpowerof_

If this function has a single argument that is a symbol, it returns the minimum power restriction of this symbol. If none was given it will be the installation dependent value -MAXPOWER which is -10000 on 32 bit machines.

## 6.32 mod_

If there are two integer arguments and the second argument is a positive short integer (less than $2^{15}$ on 32 bit computers and less than $2^{31}$ on 64 bit computers) the return value is the first argument modulus the second. Note that if the second argument is not a prime number and the first argument contains a denominator, division by zero could occur. It is up to the user to avoid such cases.

## 6.33 nargs_

Is replaced by an integer indicating the number of arguments that the function has.

## 6.34 nterms_

If this function has only one argument it is replaced by the number of terms inside this argument.

## 6.35 pattern_

Currently not active. Replaced automatically by 1.

## 6.36 poly_

Experimental. No guarantees of anything. Not even speed. For internal use with the following polynomial functions.

## 6.37 polyadd_

Experimental. No guarantees of anything. Not even speed. The arguments should be two expressions or $-variables (one of each is also allowed). Both should contain only symbols with nonnegative integer powers. The result is the sum of the two polynomials.

## 6.38 polydiv_

Experimental. No guarantees of anything. Not even speed. The arguments should be two expressions or $-variables (one of each is also allowed). Both should contain only symbols with nonnegative integer powers. The result is the quotient in the polynomial division of the first polynomial divided by the second. Any remainder is discarded.

## 6.39 polygcd_

Experimental. No guarantees of anything. Not even speed. The arguments should be two expressions or $-variables (one of each is also allowed). Both should contain only symbols with nonnegative integer powers. The result is the Greatest Common Divisor of the two polynomials. Note that a: this function can be very slow if several variables are involved and no quick divisor is found. b: the speed is very sensitive to the order of declaration. c: the algorithm is far from optimal. The future should see great improvements here.

## 6.40 polyintfac_

Experimental. No guarantees of anything. Not even speed. The argument should be an expression or $-variable. It should contain only symbols with nonnegative integer powers. The result is a polynomial that has been multiplied by a constant such that all coefficients are integer with content 1 (meaning that the GCD of all these coefficients is one).

## 6.41 polymul_

Experimental. No guarantees of anything. Not even speed. The arguments should be two expressions or $-variables (one of each is also allowed). Both should contain only symbols with nonnegative integer powers. The result is the polynomial multiplication of both polynomials.

## 6.42 polynorm_

Experimental. No guarantees of anything. Not even speed. The argument should be one expression or $-variable. It should contain only symbols with nonnegative integer powers. Normalizes the polynomial in such a way that the first term has coefficient one.

## 6.43 polyrem_

Experimental. No guarantees of anything. Not even speed. The arguments should be two expressions or $-variables (one of each is also allowed). Both should contain only symbols with nonnegative integer powers. The result is the remainder in the polynomial division of the first polynomial divided by the second polynomial.

## 6.44 polysub_

Experimental. No guarantees of anything. Not even speed. The arguments should be two expressions or $-variables (one of each is also allowed). Both should contain only symbols with nonnegative integer powers. The result is the first polynomial minus the second polynomial.

## 6.45 replace_

This function defines a rather general purpose replacement mechanism. It should have pairs of arguments. Each pair consists of a single symbol, index, vector or function, followed by what this object should be replaced by in the entire term. Functions can only be replaced by functions, indices only by indices. A vector can be replaced by a single vector or by a vectorlike expression. A symbol can be replaced by a single symbol, a numerical expression or a complete subexpression that is not indexlike or vectorlike. This mechanism is sometimes needed to make replacements in ways that are very hard with the id statements because those do not make replacements automatically inside function arguments (see 5.42). It also allows to exchange two variables as the replacements are executed simultaneously by the wildcard substitution mechanism.

```
Multiply replace_(x,y,y,x);
```

will exchange x and y.

## 6.46 reverse_

Can only occur as an argument of a function. Is replaced by the reversed string of its own arguments.

## 6.47 root_

If we have `root_(n,x)` and `n` is a positive integer and `x` is a rational number and `y` is a rational number with $y^n = x$ (no imaginary numbers are considered and negative numbers are avoided if possible. Only one root is given) then `root_(n,x)` is replaced by `y`. This function was originally intended for internal use. Do not hold it against the author that `root_(2,1)` is replaced by `1`. In the case that it is needed the user should manipulate the sign or the complexity properties externally.

## 6.48 setfun_

Currently not active.

## 6.49 sig_

Is replaced by the sign of the (numerical) argument, i.e. by -1 if there is a single negative argument and by +1 if there is a single numerical argument that is greater or equal to zero.

## 6.50 sign_

`sign_(n)` is replaced by `(-1)^n` if n is an integer.

## 6.51 sum_

General purpose sum function. The first argument should be the summation parameter (a symbol). The second argument is the starting point of summation, the third argument the 'upper' limit and a potential fourth argument the increment. These numbers should all be integers. Summation stops when the summation parameter obtains a value that has passed the upper limit. The last argument is the summant, the object to be summed over. It can be any subexpression. If it contains the summation parameter, it will be replaced by its value for each generated term. Examples:

```
sum_(j,1,4,sign_(j)*x^j/j)
sum_(i,1,9,2,sign_((i-1)/2)*x^i*invfac_(i))
```

## 6.52 sump_

Special sum function. Its arguments are like for the sum_ function, but each new term is the product of the previously generated term with the last argument in which the current value of the summation parameter has been substituted. The first term is always one. Example:

```
    Symbol i,x;
    Local F = sump_(i,0,5,x/i);
    Print;
    .end

  F =
     1 + x + 1/2*x^2 + 1/6*x^3 + 1/24*x^4 + 1/120*x^5;
```

This function is a leftover from the SCHOONSCHIP days. The ordinary sum_ function is much more readable.

## 6.53 table_

For action the arguments should be the name of a table and then either the name of a function or one symbol for each dimension of the table. In the case of the list of symbols the return value will be a monomial in the given symbols in which the powers of the symbols correspond to the table indices of the defined table elements with the coefficients the table contents corresponding to those indices. In the case of a function name the return value will be a sum over terms in which the table elements are indicated by arguments in the given function while these functions are then multiplied by the corresponding table elements. This is one way to put a complete table inside an expression and store it (with the save statement of 5.87) in a binary way for a future run in which the table can be filled again with the fillexpression (see 5.33) statement. Note that for obvious reasons one should avoid using symbols or functions that also occur inside the table definitions.

## 6.54 tbl_

This function is the 'table stub function' as used by the tablebase construction. This is explained in chapter 8. It is mainly for internal use, but it could occur in the output.

## 6.55 term_

If the single argument is a $-variable, the term_ function will be removed and the remainder of the current term will be used as value/contents for the $-variable. This is one of the ways to put things inside a $-variable.

## 6.56 termsin_

If there is a single argument and this argument is the name of an active (or previously active during the current job) expression, the function is replaced by the number of terms in this expression. Stored expressions that were entered via a load statement (see 5.51) are excluded from this because for them FORM would have to actually count the terms.

## 6.57 theta_

If there is a single numerical argument x the function is replaced by one if $x \geq 0$ and by zero if $x < 0$. If there are two numerical arguments $x_1$ and $x_2$ the function is replaced by one if $x_1 = x_2$ or if the arguments are in natural order (if theta_ would be a symmetric function there would be no reason to exchange the arguments) and by zero if the arguments are not in natural order (they would be exchanged in a symmetric function). In all other cases nothing is done.

## 6.58 thetap_

If there is a single numerical argument x the function is replaced by one if $x > 0$ and by zero if $x \leq 0$. If there are two numerical arguments $x_1$ and $x_2$ the function is replaced by zero if $x_1 = x_2$ or if the arguments are not in natural order. If the arguments are in natural order the function is replaced by one. In all other cases nothing is done.

## 6.59 Extra reserved names

In addition there are some names that have been reserved for future use. At the moment these functions do not do very much. It is hoped that in the future some simplifications of the arguments can be implemented. These functions are:

| | |
|---|---|
| sqrt_ | The regular sqare root. |
| ln_ | The natural logarithm. |
| sin_ | The sine function. |
| cos_ | The cosine function. |
| tan_ | The tangent function. |
| asin_ | The inverse of the sine function. |
| acos_ | The inverse of the cosine function. |
| atan_ | The inverse of the tangent function. |
| atan2_ | Another inverse of the tangent function. |
| sinh_ | The hyperbolic sine function. |

| | |
|---|---|
| cosh_ | The hyperbolic cosine function. |
| tanh_ | The hyperbolic tangent function. |
| asinh_ | The inverse of the hyperbolic sine function. |
| acosh_ | The inverse of the hyperbolic cosine function. |
| atanh_ | The inverse of the hyperbolic tangent function. |
| li2_ | The dilogarithm function. |
| lin_ | The polylogarithm function. |

The user is allowed to use these functions, but it could be that in the future they will develop a nontrivial behaviour. Hence caution is required.

# Chapter 7

# Brackets

At times one would like to order the output in a specific way. In an expression which is for instance a polynomial in terms of the symbol $x$, one might want to make this behaviour in terms of $x$ more apparent by printing the output in such a way, that all powers of $x$ are outside parentheses, and the whole rest is inside parentheses. This is done with the bracket statement:

```
Bracket x;
```

or in short notation

```
B x;
```

One can specify more than one object in the bracket statement, but only a single bracket statement (the last one) is considered. Bracket statements belong to the module in which they occur. Hence they are forgotten after the next end-of-module.

If a vector is mentioned in a bracket statement, all occurrences of this vector as a loose vector, a vector with any index, inside a dotproduct, or inside a tensor are taken outside brackets. If the vector occurs inside a non-commuting tensor, all other non commuting objects that are to the left of this tensor will also be taken outside the parentheses.

When a function or tensor is mentioned in a bracket statement, it is not allowed to have any arguments in the bracket statement. All occurrences of this function will be pulled outside brackets. If the function is non-commuting, all other functions and/or tensors that are non-commuting and are to the left of the specific function(s) or tensor(s) will also be outside parentheses.

The opposite of the bracket statement is the antibracket statement:

```
AntiBracket x;
```

or

```
ABracket x;
```

or

```
AB x;
```

This statement causes also brackets in the output, but now everything is put outside brackets, except for powers of x and coefficients. This way one can make the $x$-dependence apparent differently.

Because the bracket statement causes a different ordering of the terms when storing the expression, one can use this ordering in the next module. There are various ways to do this.

One can use the contents of a given bracket in a r.h.s. expression as in

```
Symbols a,b,c,x;
L  F = a*x^2+b*x+c;
B x;
.sort
L  Discriminant = F[x]^2-4*F[x^2]*F[1];
Print;
.end
```

The outside of the bracket is placed between braces after the name of the expression. The bracket that has nothing outside is referred to with the number 1. If a bracket is empty, its contents will be represented by the value zero.

The regular algorithm by which FORM finds brackets in an expression, is to start from the beginning and inspect each term until it finds the appropriate bracket. This is fully in the spirit of the sequential treatment of expressions in FORM. This can however be rather slow in big expressions that reside on a disk. Hence there is the bracket index feature. It is invoked by putting a +-sign after the bracket (or B) statement as in

```
Bracket+ x;
```

or

```
B+ x;
```

This option causes FORM to build a tree of (disk) positions for the different brackets, with the condition that the whole storage of this tree of brackets does not exceed a given maximum space, named 'bracketindexsize' (see chapter 11 on the setup parameters). If the index would need more space FORM will start skipping brackets in the index. This means that it will have to look for the bracket in a sequential fashion, but starting from the position indicated by the previous bracket in the index. This will still be very fast, provided the index is not very small.

When the bracket index option is used, FORMwill not compress the expressions that use such an index with the zlib compression, even if the user asked for this in an earlier statement. The use if the index indicates that the brackets are going to be used intensively, and hence the continuous decompression that would result would destroy most of the profit that comes from the index. If the brackets are only for cosmetics in the output, it is better not to use the index option. It does use resources to construct the index tree. Also when brackets are only used sequentially as in the features discussed below, the presence of the index is not beneficial. It should only be applied when contents of brackets are used in the above way (like with the discriminant).

There are several statements that make use of the bracket ordering:

- Keep Brackets; This statement takes from the input one term at a time as usual, but then it takes the part outside the brackets, executes the statements of the module only on that part of the term, and then, when all statements of the module have had their effect, the resulting term(s) is/are multiplied by the full content of the bracket. The next term taken from the input will be the first term of the next bracket. This way one can hide part of the terms for the pattern matcher. Also one can avoid that the same matching will occur many times, as in an expression of the type

  ```
  + f(y)*(x+x^2+x^3+x^4+1)
  ```

  If we would want to make a replacement of the type

```
Keep Brackets;
id f?{f1,f2,f3}(u?) = f(u+1)/u;
```

the pattern matching and the substitution would have to be done only once, rather than 5 times, as would be the case if the Keep bracket statement would not be used.

- Collect FunctionName; The contents of the various brackets will be placed inside a function with the given name. Hence

  ```
  + f(y)*(x+x^2+x^3+x^4+1)
  + f(y^2)*(x+2*x^2+3*x^3+4*x^4+1)
  ```

  with

  ```
  Collect h;
  ```

  would result in:

  ```
  + f(y)*h(x+x^2+x^3+x^4+1)
  + f(y^2)*h(x+2*x^2+3*x^3+4*x^4+1)
  ```

  This can be very useful to locate $x$-dependence even further, because bracketting the new expression in terms of $h$ could make very clear whether a given polynomial in $x$ would factor the whole expression, or which factors are occurring. To bring `h(x+1)` and `h(2*x+2)` to multiples of the same objects one should consult the pages on the normalize (5.63) and makeinteger (5.53) statements.

  The Collect statement, together with the PolyFun statement, can also be very useful, if the variable $x$ (or other variables) is temporarily not playing much of a role in the pattern matching. It can make the program much faster.

  For more information on the collect statement one should consult section 5.10.

Restrictions: The bracket index can only be used with active expressions. Hence the access of specific brackets in stored expressions will always be of the slow variety. To make it faster, one can copy the expression into a local expression with indexed brackets, use it, and drop the expression when it is not needed any longer.

The brackets can also be used to save space on the disk in problems in which the expressions become rather large. Let us assume the following simple problem:

```
Symbols x1,...,x12;
Local F = (x1+...+x12)^10;
.sort
id  x1 = x4+x7;
.end
```

If the program is run like this the expression F contains 352716 terms after the sort and after the id the sorting in the .end results in a final stage sort of which the statistics are:

```
Time =      46.87 sec
                F        Terms active    =     504240
                         Bytes used      =   13462248

Time =      52.09 sec    Generated terms =     646646
                F        Terms in output =     184756
                         Bytes used      =    4883306
```

We see, that the intermediate sort file still contains more than 500000 terms and more than 13 Mbytes, while the final result contains less than 5 Mbytes. Why is this? When the terms in FORM are sorted first come the powers of `x1`, because this is the variable that was declared first. Hence the terms that do not have powers of `x1` come much later in the input and will not be compared with the terms generated by the substitution of for instance a single power of `x1` until very late in the sorting. What can we do about this? We can try to group the terms in the first sort such that after the substitution like terms will be 'very close' to each other and hence will add quickly. This is done in the program

```
Symbols x1,...,x12;
Local F = (x1+...+x12)^10;
AntiBracket x1,x4,x7;
.sort
id  x1 = x4+x7;
.end
```

Now all powers of the mentioned variables will be inside the brackets and all other variables will be outside. Because the terms inside the brackets are all following each other in the input of the second module, terms that will add will be generated closely together. The result is visible in the final statistics:

```
Time =      47.23 sec
                F        Terms active    =     184761
                         Bytes used      =    4928008

Time =      48.40 sec    Generated terms =     646646
                F        Terms in output =     184756
                         Bytes used      =    4883306
```

Now the final step of the sorting has already almost the proper number of terms. The difference is due to brackets that are half in one 'patch' on the disk and half in the next 'patch' (for the meaning of the patches, one should read the part about sorting in chapter 11 on the setup file. It should be rather clear now that this saves disk space and the corresponding amount of time. These early cancellations can also be seen in the first statistics message of the second module. In the first case it is

```
Time =      19.76 sec    Generated terms =      10431
                F   5216 Terms left      =       8065
                         Bytes used      =     239406
```

and in the second case it is

```
Time =      22.82 sec    Generated terms =      10124
                F   5835 Terms left      =       3186
                         Bytes used      =      96678
```

This also causes a more efficient use of the large buffer and again a better use of the disk. There have been cases in which this ‘trick’ was essential to keep the sort file inside the available disk space.

# Chapter 8

# The TableBase

The tablebase statement controls a database-like structure that allows FORM to control massive amounts of data in the form of tables and table elements. The contents of a tablebase are formed by one or more table declarations and a number of fill statements. These fill statements however are not immediately compiled. For each fill statement a special fill statement is generated and compiled that is of the form

```
Fill tablename(indices) = tbl_(tablename,indices,arguments);
```

The function tbl_ is a special function to make a temporary table substitution. It indicates that the corresponding element can be found in a tablebase that has been opened. At a later stage one can tell FORM to see which table elements are actually needed and then only those will be loaded from the tablebase and compiled.

Tablebases have a special internal structure and the right hand sides of the fill statements are actually stored in a compressed state. These tablebases can be created with special statements and uploaded with any previously compiled table. Hence one can prepare a tablebase in a previous job, to be used at a later stage, without the time penalty of loading the whole table at that later stage.

Assume we have a file named no11fill.h that loooks like

```
Symbols ...;
Table,sparse,no11fill(11,N?);
Fill no11fill(-3,1,1,1,1,1,1,1,0,0,0) = ....
Fill no11fill(-2,1,1,1,1,1,1,1,0,0,0) = ....
etc.
```

It should be noted that only sparse tables can be stored inside a tablebase. The right hand sides could be typically a few kilobytes of formulas and there could be a few thousand of these fill statements. To make this into a tablebase one would use the program

```
#-
#include no11fill.h
#+
TableBase "no11.tbl" create;
TableBase "no11.tbl" addto no11fill;
.end
```

The include instruction makes that FORM reads and compiles the table. Then the first tablebase statement creates a new tablebase file by the name no11.tbl. If such a file existed already, the old version will be lost. If one would like to add to an existing tablebase, one should use the open keyword. The second tablebase statement adds the table no11fill to the tablebase file no11.tbl. This takes care of declaring the table, making an index of all elements that have been filled and putting their right hand sides, in compressed form, into the tablebase. The compression is based on the zlib library, provided by Jean-loup Gailly and Mark Adler (version 1.1.3, July 9th, 1998) and it strikes a nice balance between speed and compression ratio.

The tablebase can be loaded in a different program as in

```
TableBase "no11.tbl" open;
```

This loads the main index of the file into memory.

If one would like to compile the short version of the fill statements (the normal action at this point) one needs to use the load option. Without any names of tables it will read the index of all tables. If tables are specied, only the index of those tables is taken and the proper `tbl_` fill statements are generated:

```
TableBase "no11.tbl" open;
TableBase "no11.tbl" load no11fill;
```

If one would like to compile the complete tables, rather than just the shortened versions, one can use the enter option as in:

```
TableBase "no11.tbl" open;
TableBase "no11.tbl" enter no11fill;
```

Let us assume we used the load option. Hence now an occurrence of a table element will be replaced by the stub-function `tbl_`. In order to have this replaced by the actual right hand side of the original fill statement we have to do some more work. At a given moment we have to make FORM look which elements are actually needed. This is done with the TestUse statement as in

```
TestUse no11fill;
```

This does nothing visible. It just marks internally which elements will be needed and have not been entered yet.

The actual entering of the needed elements is done with the use option:

```
TableBase "no11.tbl" use;
```

If many elements are needed, this statement may need some compilation time. Note however that this is time at a moment that it is clear that the elements are needed, which is entirely different from a fixed time at the startup of a program when the whole table is loaded as would have to be done before the tablebase statement existed. Usually however only a part of the table is needed, and in the extreme case only one or two elements. In that case the profit is obvious.

At this point the proper elements are available inside the system, but because we have two versions of the table (one the short version with `tbl_`, the other the complete elements) we have to tell FORM to apply the proper definitions with the 'apply' statement.

```
Apply;
```

Now the actual rhs will be inserted.

One may wonder why this has to be done in such a 'slow' way with this much control over the process. The point is that at the moment the table elements are recognized, one may not want the rhs yet, because it may be many lines. Yet one may want to take the elements away from the main stream of action. Similarly, having a table element recognized at a certain stage, may not mean automatically that it will be needed. The coefficient may still become zero during additional manipulations. Hence the user is left with full control over the process, even though that may lead to slightly more programming. It will allow for the fastest program.

For the name of a tablebase we advise the use of the extension .tbl to avoid confusion.

Note that the above scheme may need several applications, if table elements refer in their definition to other table elements. This can be done with a construction like:

```
#do i = 1,1
    TestUse;
    .sort
    TableBase "basename.tbl" use;
    Apply;
    if ( count(tbl_,1) ) Redefine i "0";
    .sort
#enddo
```

It will stay in the loop until there are no more `tbl_` functions to be resolved.

The complete syntax (more is planned):

## 8.1 addto

Syntax:
TableBase "file.tbl" addto tablename;
TableBase "file.tbl" addto tablename(tableelement);
See also open (8.9) and create (8.4).
Adds the contents of a (sparse) table to a tablebase. The base must be either an existing tablebase (made accessible with an open statement) or a new tablebase (made available with a create statement). In the first version what is added is the collection of all fill statements that have been used to define elements of the indicated table, in addition to a definition of the table (if that had not been done yet). In the second version only individual elements of the indicated table are added. These elements are indicated as it should be in the left hand side of a fill statement.
One is allowed to specify more than one table, or more than one element. If one likes to specify anything after an element, it should be realized that one needs to use a comma for a separator, because blank spaces after a parenthesis are seen as irrelevant.
Examples:

```
TableBase "no11.tbl" open;
TableBase "no11.tbl" load;
TableBase "no11.tbl" addto no11filb;
TableBase "no11.tbl" addto no11fill(-3,1,1,1,1,2,1,1,0,0,0),
                           no11fill(-2,1,1,2,1,1,1,1,0,0,0);
```

## 8.2 apply

Syntax:
Apply; Apply number;
Apply tablename(s); Apply number tablename(s);
See also testuse (8.10) and use (8.11).

The actual application of fill statements that were taken from the tablebases. If no tables are specified, this is done for all tables, otherwise only for the tables whose names are mentioned. The elements must have been registered as used before with the application of a testuse statement, and they must have been compiled from the tablebase with the use option of the tablebase statement. The number refers to the maximum number of table elements that can be substituted in each term. This way one can choose to replace only one element at a time. If no number is present all occurrences will be replaced. This refers also to occurences inside function arguments. If only a limited number is specified in the apply statement, the occurrences inside function arguments have priority.

## 8.3 audit

Syntax:
  TableBase "file.tbl" audit;
See also open (8.9)
Prints a list of all tables and table elements that are defined in the specified tablebase. This tablebase needs to be opened first. As of the moment there are no options for the audit. Future options might include formatting of the output.

## 8.4 create

Syntax:
  TableBase "file.tbl" create;
See also open (8.9)
This creates a new file with the indicated name. This file will be initialized as a tablebase. If there was already a file with the given name, its old contents will be lost. If one would like to add to an existing tablebase, one should use the 'open' option.

## 8.5 enter

Syntax:
  TableBase "file.tbl" enter;
  TableBase "file.tbl" enter tablename(s);
See also open (8.5) and load (8.6).
Scans the specified tablebase and (in the first variety) creates for all elements of all tables in the tablebase a fill statement with its full contents. This is at times faster than reading the fill statements from a regular input file, because the tablebase has its contents compressed. Hence this costs less file access time. When table names are specified, only the tables that are mentioned have their elements treated this way.
The tablebase must of course be open for its contents to be available.
If one would like FORM to only see what elements are available and load that information one should use the load option.

## 8.6 load

Syntax:
  TableBase "file.tbl" load;

TableBase "file.tbl" load tablename(s);
See also open (8.9) and enter (8.5).
Scans the index of the specified tablebase and (in the first variety) creates for all elements of all tables in the tablebase a fill statement of the type

```
Fill tablename(indices) = tbl_(tablename,indices,arguments);
```

This is the fill statement that will be used when elements of one of these tables are encountered. The function `tbl_` is called the (table)stub function. When table names are specified, only the tables that are mentioned have their elements treated this way.
The tablebase must of course be open for its contents to be available.
If one would like to actually load the complete fill statements, one should use the enter option.

## 8.7 off

Syntax:
TableBase "file.tbl" off subkey;
See also addto (8.1) and off (8.8).
Currently only the subkey 'compress' is recognized. It makes sure that no compression is used when elements are being stored in a tablebase with the addto option. This could be interesting when the right hand sides of the fill statements are relatively short.

## 8.8 on

Syntax:
TableBase "file.tbl" on subkey;
See also addto (8.1) and off (8.7).
Currently only the subkey 'compress' is recognized. It makes sure that compression with the gzip algorithms is used when elements are being stored in a tablebase with the addto option. This is the default.

## 8.9 open

Syntax:
TableBase "file.tbl" open;
See also create (8.4)
This opens an existing file with the indicated name. It is assumed that the file has been created with the 'create' option in a previous FORM program. It gives the user access to the contents of the tablebase. In addition it allows the user to add to its contents.
Just like with other files, FORM will look for the file in in current directory and in all other directories mentioned in the environment variable 'FORMPATH' (see for instance the #call (1.6) and the #include (1.24) instructions).

## 8.10 testuse

Syntax:
TestUse;

TestUse tablename(s);
See also use (8.11).
Tests for all elements of the specified tables (if no tables are mentioned, this is done for all tables) whether they are used in a stub function `tbl_`. If so, this indicates that these elements must be compiled from a tablebase, provided this has not been done already. The compilation will have to be done at a time, specified by the user. This can be done with the use option. All this statement does is set some flags in the internals of FORM for the table elements that are encountered in the currently active expressions.

## 8.11 use

Syntax:
TableBase "file.tbl" use;
TableBase "file.tbl" use tablename(s);
See also testuse (8.10) and apply (8.2).
Causes those elements of the specified tables to be compiled, that a previous testuse statement has encountered and that have not yet been compiled before. If no tables are mentioned this is done for all tables. The right hand sides of the definition of the table elements will not yet be substituted. That is done with an apply statement.

# Chapter 9

# Dirac algebra

For its use in high energy physics FORM is equipped with a built-in class of functions. These are the gamma matrices of the Dirac algebra which are generically denoted by g_. The gamma matrices fulfill the relations:

```
{g_(j1,mu),g_(j1,nu)} = 2 * d_(mu,nu)
[g_(j1,mu),g_(j2,nu)] = 0    j1 not equal to j2.
```

The first argument is a so-called spin line index. When gamma matrices have the same spin line, they belong to the same Dirac algebra and commute with the matrices of other Dirac algebra's. The indices mu and nu are over space-time and are therefore usually running from 1 to 4 (or from 0 to 3 in Bjorken & Drell metric). The totally antisymmetric product e_(m1,m2,...,mn)g_(j,m1) $\times$ ...g_(j,mn)/n! is defined to be gamma5 or g5_(j). The notation 5 finds its roots in 4 dimensional space-time. The unit matrix is denoted by gi_(j). In four dimensions a basis of the Dirac algebra can be given by:

```
gi_(j)
g_(j,mu)
[g_(j,mu),g_(j,nu)]/2
g5_(j)*g_(j,mu)
g5_(j)
```

In a different number of dimensions this basis is correspondingly different. We introduce the following notation for convenience:

```
g6_(j) = gi(j) + g5_(j)     (from Schoonschip)
g7_(j) = gi(j) - g5_(j)
g_(j,mu,nu) = g_(j,mu)*g_(j,nu)  (from Reduce)
g_(j,mu,nu,.....,ro,si) =
                  g_(j,mu,nu,.....,ro)*g_(j,si)
g_(j,5_) = g5_(j)
g_(j,6_) = g6_(j)
g_(j,7_) = g7_(j)
```

The common operation on gamma matrices is to obtain the trace of a string of gamma matrices. This is done with the statement:

trace4, j — Take the trace in 4 dimensions of the combination of all gamma matrices with spin line j in the current term. Any non-commuting objects that may be between some of these matrices are ignored. It is the users responsibility to issue this statement only after all functions of the relevant matrices are resolved. The four refers to special tricks that can be applied in four dimensions. This allows for relatively compact expressions. For the complete syntax, consult 5.107

tracen, j — Take the trace in an unspecified number of dimensions. This number of dimensions is considered to be even. The traces are evaluated by only using the anticommutation properties of the matrices. As the number of dimensions is not specified the occurrence of a g5_(j) is a fatal error. In general the expressions that are generated this way are longer than the four dimensional expressions. For the complete syntax, consult 5.108

It is possible to alter the value of the trace of the unit matrix. Its default value is 4, but by using the statement (see 5.110)

```
unittrace value;
```

it can be altered. Value may be any positive short number ($< 2^{15}$ on 32 bit machines and $< 2^{31}$ on 64 bit machines) or a single symbol with the exception of the symbol i_.

There are several options for the 4-dimensional traces. These options find their origin in the Chisholm relation that is valid in 4 dimensions but not in a general number of dimensions. This relation can be found in the literature. It is given by:

$$\gamma_\mu Tr[\gamma_\mu S] = 2(S + S^R) \tag{9.1}$$

in which S is a string of gamma matrices with an odd number of matrices ($\gamma_5$ counts for an even number of matrices). $S^R$ is the reversed string. This relation can be used to combine traces with common indices. The use of this relation is the default for trace4. If it needs to be switched off, one should add the extra parameter 'nocontract':

```
trace4,nocontract,j;
```

The parameter 'contract' is the default but it can be used to enhance the readability of the program. The second parameter that refers to this relation is the parameter 'symmetrize'. Often it happens that there are two or more common indices in two spin lines. Without the symmetrize parameter (or with the 'nosymmetrize' parameter) the first of these indices is taken and the relation is applied to it. With the 'symmetrize' parameter the average over all possibilities is taken. This means of course that of there are two common indices the amount of work is doubled. There is however a potential large advantage. In some traces that involve the use of $\gamma_5$ the use of automatic algorithms results often in an avalanche of terms with a single Levi-Civita tensor, while symmetry arguments can show that these terms should add up to zero. By working out the traces in a more symmetric fashion FORM is often capable of eliminating all or nearly all of these Levi-Civita tensors. Normally such an elimination is rather complicated. It involves relations that have so far defied proper implementation, even though people have been looking for such algorithms already for a long time. Hence the use of the symmetry from the beginning seems at the moment the best bet.

It is possible to only apply the Chisholm identity without taking the trace. This is done with the chisholm statement (see 5.9).

The n dimensional traces can use a special feature, when the declaration of the indices involved will allow it. When an index has been declared as n-dimensional and the dimension is followed by a second symbol as in

```
symbols n,nn;
index mu=n:nn;
```

and if the index `mu` is a contracted index in a single n-dimensional trace, then the formula for this trace can be shortened by using `nn` (one term) instead of the quantity $(n-4)$ (two terms). This can make the taking of the n-dimensional traces significantly faster.

Algorithms:

FORM has been equipped with several built in rules to keep the number of generated terms to a minimum during the evaluation of a trace. These rules are:

**rule 0** Strings with an odd number of matrices (gamma5 counts for an even number of matrices) have a trace that is zero, when using trace4 or tracen.

**rule 1** A string of gamma matrices is first scanned for adjacent matrices that have the same contractable index, or that are contracted with the same vector. If such a pair is found, the relations

```
g_(1,mu,mu) = gi_(1)*d_(mu,mu)
g_(1,p1,p1) = gi_(1)*p1.p1
```

are applied.

**rule 2** Next there is a scan for a pair of the same contractable indices that has an odd number of other matrices in between. This is done only for 4 dimensions (trace4) and the dimension of the indices must be 4. If found, the Chisholm identity is applied:

```
g_(1,mu,m1,m2,...mn,mu) = -2*g_(1,mn,...,m2,m1)
```

**rule 3** Then (again only for trace4) there is a search for a pair of matrices with the same 4 dimensional index and an even number of matrices in between. If found, one of the following variations of the Chisholm identity is applied:

```
g_(1,mu,m1,m2,mu) = 4*gi_(1)*d_(m1,m2)
g_(1,mu,m1,m2,...,mj,mn,mu) =
                 2*g_(1,mn,m1,m2,...,mj)
                +2*g_(1,mj,...,m2,m1,mn)
```

**rule 4** Then there is a scan for pairs of matrices that have the same index or that are contracted with the same vector. If found, the identity:

```
g_(1,mu,m1,m2,...,mj,mn,mu) =
          2*d_(mu,mn)*g_(1,mu,m1,m2,...,mj)
         -2*d_(mu,mj)*g_(1,mu,m1,m2,...,mn)
          ....
       -/+2*d_(mu,m2)*g_(1,mu,m1,...,mj,mn)
       +/-2*d_(mu,m1)*g_(1,mu,m2,...,mj,mn)
       -/+2*d_(mu,mu)*g_(1,m1,m2,...,mj,mn)
```

is used to 'anticommute' these identical objects till they become adjacent and can be eliminated with the application of rule 1. In the case of an n-dimensional trace and when `mu` is an index (it might also be a vector in the above formula) for which the definition of the dimension involved two symbols, there is a shorter formula. In that case the last three terms can be combined into two terms:

```
-/+(n-4)*g_(1,m1,m2,...,mj,mn)
-/+4*d_(m1,m2)*g_(1,m3,m4,...,mj,mn)
```

It should be clear now that this formula is only superior, when there is a single symbol to represent $(n-4)$. After this all gamma matrices that are left have a different index or are contracted with different vectors. These are treated using:

**rule5** Traces in 4 dimensions for which all gamma matrices have a different index, or are contracted with a different four-vector are evaluated using the reduction formula

```
g_(1,mu,nu,ro) =
              g_(1,5_,si)*e_(mu,nu,ro,si)
             +d_(mu,nu)*g_(1,ro)
             -d_(mu,ro)*g_(1,nu)
             +d_(nu,ro)*g_(1,mu)
```

For tracen the generating algorithm is based on the generation of all possible pairs of indices/vectors that occur in the gamma matrices in combination with their proper sign. When the dimension is not specified, there is no shorter expression.

Remarks:

When an index is declared to have dimension n and the command trace4 is used, the special 4 dimensional rules 2 and 3 are not applied to this index. The application of rule 1 or 4 will then give the correct results. The result will nevertheless be wrong due to rule 5, when there are at least 10 gamma matrices left after the application of the first 4 rules, as the two algorithms in rule 5 give a difference only, when there are at least 10 gamma matrices. For counting gamma matrices the $\gamma_5$ counts for 4 matrices with respect to this rule. The result is unpredictable, when both indices in four dimensions and indices in n dimensions occur in the same string of gamma matrices. Therefore one should be very careful, when using the four dimensional trace under the condition that the results need to be correct in n dimensions. This is sometimes needed, when a $\gamma_5$ is involved. The tracen-statement will not allow the presence of a $\gamma_5$. In general it is best to simulate n-dimensional traces with a $\gamma_5$ separately. The eventual trace, with all matrices with a different index, can be generated with the use of the 'distrib_' function:

```
*
*   Symmetric trace of a gamma5 and 12 regular matrices
*
I   m1,...,m12;
F   G5,g1,g2;
L   F = G5(m1,...,m12);
id  G5(?a) = distrib_(-1,4,g1,g2,?a);
id  g1(?a) = e_(?a);
id  g2(?a) = g_(1,?a);
```

```
tracen,1;
.end
```

```
Time =       1.07 sec    Generated terms =      51975
                F        Terms in output =      51975
                         Bytes used      =     919164
```

This rather symmetric result is in contrast to the 4-dimensional result which is much shorter, but it is very unsymmetric:

```
*
*   Regular trace of a gamma5 and 12 regular matrices
*
I   m1,...,m12;
L   F = g_(1,5_,m1,...,m12);
trace4,1;
.end
```

```
Time =       0.02 sec    Generated terms =       1053
                F        Terms in output =       1029
                         Bytes used      =      20284
```

The precise workings of the distrib_ function is given in 6.11.

# Chapter 10

# A few notes on the use of a metric

When FORM was designed, it was decided to make its syntax more or less independent of a choice of the metric. Hence statements and facilities that programs like SCHOONSCHIP or REDUCE provide but which depend on the choice of a metric have been left out. Instead there are facilities to implement any choice of the metric, when the need really arises. When one makes a proper study of it, it turns out that one usually has to do very little or nothing.

First one should realize that FORM does not know any specific metric by itself. Dotproducts are just objects of manipulation. It is assumed that when a common index of two vectors is contracted, this works out properly into a scalar object. This means that if one has a metric with upper and lower indices, one index is supposed to be an upper index and the other is supposed to be a lower index. If the user does not like this, it is his/her responsibility to force the system into a different action. This is reflected in the fact that FORM does not have an internal metric tensor $\eta_{\mu\nu}$. It has only a Kronecker delta $\delta_{\mu\nu}$ = `d_(mu,nu)` with `p(mu)*d_(mu,nu)*q(nu)` $\rightarrow$ `p.q` when mu and nu are summable indices.

The dependency of a metric usually enters with statements like $p^2 = \pm m^2$, which the user should provide anyway, because FORM does not have such knowledge. Connected to this is the choice of a propagator as either $\gamma_\mu p_\mu + m$ or $\gamma_\mu p_\mu + i\ m$. This is also something the user should provide. The only objects that FORM cecognizes and that could be considered as metric-dependent are the gamma matrices and the Levi-Civita tensor `e_`. Because the trace of a $\gamma_5$ involves a Levi-Civita tensor, the two are intimately connected. The anticommutator of two gamma matrices is defined with the Kronecker delta. Amazingly enough that works out well, provided that, if such Kronecker delta's survive in the output, they are interpreted as a metric tensor. This should be done with great care, because at such a point one does something that depends of the metric; one may have to select whether the indices are upper or lower indices. One should check carefully that the way the output is interpreted leads indeed to the results that are expected. This is anyway coupled to how one should interpret the input, because in such a case one would also have an input with 'open' indices and give them a proper interpretation. The rule is that generally one does not have to do anything. The upper indices in the input will be upper indices in the output and the same for lower indices.

The contraction of two Levi-Civita tensors will give products of Kronecker delta's. This means that formally there could be an error of the sign of the determinant of the metric tensor, if one would like the Kronecker delta to play the role of a metric tensor. Hence it is best to try to avoid such a situation.

In FORM the $\gamma_5$ is an object that anticommutes with the $\gamma_\mu$ and has $\gamma_5\gamma_5 = 1$. Its properties in the trace are

$$Tr[\gamma_5\gamma_{m_1}\gamma_{m_2}\gamma_{m_3}\gamma_{m_4}] \quad = \quad \epsilon_{\mu_1\mu_2\mu_3\mu_4}$$

This has a number of interesting consequences. The V-A and V+A currents are represented by $\gamma_7\gamma_\mu = (1-\gamma_5)\gamma_\mu$ and $\gamma_6\gamma_\mu = (1+\gamma_5)\gamma_\mu$ respectively. Under conjugation we have to replace $\gamma_5$ by $-\gamma_5$ as is not uncommon.

There was a time that a conjugation operation was planned in FORM. As time progressed, it was realized that this would introduce problems with some of the internal objects. Hence some objects have the property that they are considered imaginary. In practise FORM does not do anything with this. Neither does it do anything with the declarations real, complex and imaginary. If ever a way is found to implent a conjugation operator that will make everybody happy, it may still be built in.

The above should give the user enough information to convert any specific metric to what is needed to make FORM do what is expected from it. Afterwards one can convert back, provided no metric specific operations are done. Such metric specific things are for instance needed in some types of approximations in which one substitutes objects by (vector)components halfway the calculation. In that case one cannot rely on that the conversions at the beginning and the end will be compensating each other. For this case FORM allows the user to define a private metric. All the tools exist to make this a success with the exception of a loss in speed of course. Let us have a look at the contraction of two Levi-Civita tensors in an arbitrary metric:

```
Indices m1,m2,m3,n1,n2,n3,i1,i2,i3;
Cfunction eta(symmetric),e(antisymmetric);
Off Statistics;
*
*   We have our own Levi-Civita tensor e
*
Local F = e(m1,m2,m3)*e(m1,m2,m3);
*
*   We write the contraction as
*
id  e(m1?,m2?,m3?)*e(n1?,n2?,n3?) =
        e_(m1,m2,m3)*e_(i1,i2,i3)*
            eta(n1,i1)*eta(n2,i2)*eta(n3,i3);
*
*   Now we can use the internal workings of the contract:
*
Contract;
Print +s;
.sort

   F =
       + eta(i1,i1)*eta(i2,i2)*eta(i3,i3)
       - eta(i1,i1)*eta(i2,i3)^2
       - eta(i1,i2)^2*eta(i3,i3)
       + 2*eta(i1,i2)*eta(i1,i3)*eta(i2,i3)
       - eta(i1,i3)^2*eta(i2,i2)
```

```
  ;

*
*   For specifying a metric we need individual components:
*
Sum i1,1,2,3;
Sum i2,1,2,3;
Sum i3,1,2,3;
Print +s;
.sort

F =
   + 6*eta(1,1)*eta(2,2)*eta(3,3)
   - 6*eta(1,1)*eta(2,3)^2
   - 6*eta(1,2)^2*eta(3,3)
   + 12*eta(1,2)*eta(1,3)*eta(2,3)
   - 6*eta(1,3)^2*eta(2,2)
  ;

*
*   And now we can provide the metric tensor
*
id  eta(1,1) = 1;
id  eta(2,2) = 1;
id  eta(3,3) = -1;
id  eta(1,2) = 0;
id  eta(1,3) = 0;
id  eta(2,3) = 0;
Print +s;
.end

F =
   - 6
  ;
```

This is the ultimate in flexibility of course. It can also be worked out in a different way. In this case we try to change the behaviour of the Kronecker delta a bit. This is dangerous and needs, in addition to a good understanding of what is happening, good testing to make sure that what the user wants is indeed what does happen. Here we use the FixIndex (5.34) statement. This one assigns specific values to selected diagonal elements of the Kronecker delta. Of course it is the responsibility of the user to make sure that the calculation will indeed run into those elements. This is by no means automatic, because when FORM uses formal indices it never writes them out in components. Moreover, it would not be defined what would be the components connected to an index. The index could run over $0, 1, 2, 3$ or over $1, 2, 3, 4$, or maybe even over $5, 7, 9, 11$. And what does an n-dimensional index run over? In the above example it is the sum (5.97) statement that determines this. Hence this is fully under the control of the user. Therefore a proper way to deal with the above example would be

```
Indices i1,i2,i3;
```

```
FixIndex 1:1,2:1,3:-1;
Off Statistics;
*
Local F = e_(i1,i2,i3)*e_(i1,i2,i3);
Sum i1,1,2,3;
Sum i2,1,2,3;
Sum i3,1,2,3;
Print +s;
.sort

F =
   + 6*e_(1,2,3)*e_(1,2,3)
  ;

Contract;
Print +s;
.end

F =
   - 6
  ;
```

In the case that one would like to exchange the order of the summation and the contraction, while using the FixIndex mechanism, one needs to be more careful. In that case we have to prevent the indices from being summed over while they are indices of a Kronecker delta, because as long as the indices are symbolic, FORM will replace `d_(i1,i1)` by the dimension of `i1`, and that is not what we want. Hence we have to declare the indices to be non-summable by giving them dimension zero:

```
Indices i1=0,i2=0,i3=0;
FixIndex 1:1,2:1,3:-1;
Off Statistics;
*
Local F = e_(i1,i2,i3)*e_(i1,i2,i3);
Contract;
Print +s;
.sort

F =
   + d_(i1,i1)*d_(i2,i2)*d_(i3,i3)
   - d_(i1,i1)*d_(i2,i3)*d_(i2,i3)
   - d_(i1,i2)*d_(i1,i2)*d_(i3,i3)
   + 2*d_(i1,i2)*d_(i1,i3)*d_(i2,i3)
   - d_(i1,i3)*d_(i1,i3)*d_(i2,i2)
  ;

Sum i1,1,2,3;
Sum i2,1,2,3;
Sum i3,1,2,3;
Print +s;
```

```
   .end

F =
    - 6
  ;
```

As we can see, the automatic summation over the indices is not performed now and this gives us a chance to do the summation manually. After that the fixindex statement can have its effect.

It should be clear from the above examples that it is usually much easier to manipulate the input in such a way that the terms with two Levi-Civita tensors have the negative sign from the beginning. This would give programs that are less complicated and much faster.

Hence we are faced with the situation that in normal cases one does not do anything. If one wants to go beyond this and wants to interfere with the inner workings themselves by for instance inserting a factor $i$ in front of the $\gamma_5$ and emulating the upper and lower indices of a favorite metric, this leads from one problem to the next. Extreme care is needed. This is usually done by people who have first worked with other programs in which things don't work as naturally as in FORM. By the time one has really figured out how to deal with the metric and how to make use of the internal algorithms of FORM, one usually does not have to do very much again.

As in the Zen saying:

To the beginning student mountains are mountains and water is water. To the advanced student mountains stop being mountains and water stops being water. To the master mountains are mountains again and water is water again.

Of course the modern master also checks that what he expects the system to do, is indeed what the system does.

# Chapter 11

# The setup

When FORM is started, it has a number of settings built in that were determined during its installation. If the user would like to alter these settings, it is possible to either specify their desired values in a setup file or to do so at the beginning of the program file. There are two ways in which FORM can find a setup file. The first way is by having a file named 'form.set' in the current directory. If such a file is present, FORM will open it and interpret its contents as setup parameters. If this file is not present, one may specify a setup file with the -s option in the command tail. This option must precede the name of the input file. After the -s follow one or more blanks or tabs and then the full name of the setup file. FORM will try to read startup parameters from this file. If a file 'form.set' is present, FORM will ignore the -s option and its corresponding file name. This order of interpretation allows the user to define an alias with a standard setup file which can be overruled by a local setup file. If, in the beginning of the program file, before any other statements with the exception of the #- instruction and commentary statements, there are lines that start with #: the remaining contents of these lines are interpreted exactly like the lines in the setup file. The specifications in the program file take precedence over all other specifications. If neither of the above methods is used, FORM will use a built in set of parameters. Their values may depend on the installation and are given below.

The following is a list of parameters that can be set. The syntax is rather simple: The full word must be specified (case insensitive), followed by one or more blanks or tabs and the desired number, string or character. Anything after this is considered to be commentary. In the setup file lines that do not start with an alphabetic character are seen as commentary. The sizes of the buffers are given in bytes, unless mentioned otherwise. A word is 2 bytes for 32 bit machines and 4 bytes for 64 bit machines.

| | |
|---|---|
| bracketindexsize | Maximum size in bytes of any individual index of a bracketted expression. Each expression will have its own index. The index starts with a relatively small size and will grow if needed. But it will never grow beyong the specified size. If more space is needed, FORM will start skipping brackets and find those back later by linear search. See also chapter 7 and section 5.7. |
| CommentChar | This should be followed by one or more blanks and a single non-blank character. This character will be used to indicate commentary, instead of the regular $*$ in column 1. |

**CompressSize** When compressing output terms, FORM needs a compression buffer. This buffer deals recursively with compression and decompression of terms that are either written or read. Its size will be at least MaxTermSize but when there is heavy use of expressions in the right hand side of definitions or substitution it would have to be considerably longer. It is hoped that in the future this parameter can be eliminated. CompressSize should be given in bytes.

**ConstIndex** This is the number of indices that are considered to be constant indices like in fixed vector components (the so-called fixed indices). The size of this parameter is not coupled to any array space, but it should not go much beyond 1000 on a 32 bit machine. On a 64 bit machine it can go considerably further.

**ContinuationLines** The number of continuation lines that the local Fortran compiler will allow. This limits the number of continuation lines, when the output option 'Format Fortran' (see 5.35) is selected.

**DotChar** There should be a single character following this name (and the blank(s) after it). This character will be used instead of the _, when dotproducts are printed in Fortran output. This option is needed because some Fortran compilers do not recognize the underscore as a valid character. In the olden days one could use here the dollar character but nowadays many Fortran compilers do not recognize this character as belonging to a variable name.

**FunctionLevels** The maximum number of levels that may occur, when functions have functions in their arguments.

**IncDir** Directory (or path of directories) in which FORM will look for files if they are not to be found in the current directory. This involves files for the #include and #call instructions. This variable takes precedence over the Path variable.

**InsideFirst** Not having any effect at the moment.

**MaxNumberSize** Allows the setting of the maximum size of the numbers in FORM. The number should be given in words. For 32 bit systems a word is two bytes and for 64 bit systems a word is 4 bytes. The number size is always limited by the maximum terms size (see MaxTermSize). Actually it has to be less than half of MaxTermSize because a coefficient contains both a numerator and a denominator. It is not always a good idea to have the number size at its maximum value, especially when MaxTermSize is large. In that case it could be very long before a runaway algorithm runs into limitations of size (arithmetic for very long fractions is not very fast due to the continuous need for computing GCD's)

| | |
|---|---|
| MaxTermSize | This is the maximum size that an individual term may occupy in words. This size does not affect any allocations. One should realize however that the larger this size is the heavier the demand can be on the workspace, because the workspace acts as a heap during the execution and sometimes allocations have to be made in advance, before FORM knows what the size of the term will be. Consequently the evaluation tree cannot be very deep, when WorkSpace / MaxTermSize is not very big. MaxTermSize controls mainly how soon FORM starts complaining about terms that are too complicated. Its absolute maximum is 32568 on 32 bit systems and about $10^9$ on 64 bit systems (of course the workspace would have to be considerably larger than that....). |
| MaxWildCards | The maximum number of wildcards that can be active in a single matching of a pattern. |
| NwriteStatistics | When this word is mentioned, the default setting for the statistics is that no run time statistics will be shown. Ordinarily they will be shown. |
| OldOrder | A special flag (values ON/OFF) by which one can still select the old option of not checking for the order of statements inside a module. This should be used only in the case that it is nearly impossible to change a program to the new mode in which the order of the statements (declarations etc) is relevant. In the future this old mode may not exist. |
| Parentheses | The maximum number of nestings of parentheses or functions inside functions. The variable may be eliminated in a later version. |
| Path | Directory (or path of directories) in which FORM will look for files if they are not to be found in the current directory. This involves files for the #include and #call instructions. FORM will test this path after a potential path specified as IncDir. |
| ScratchSize | The size of the input and the output buffers for the regular algebra processing. Terms are read in in chunks this size and are written to the output file using buffers of this size. There are either two or three of these buffers, depending on whether the hide facility is being used (see 5.40). These buffers must have a size that is at least as large as the MaxTermSize. These buffers act as caches for the files with the extension .sc1, .sc2 and .sc3. |
| SlavePatchSize | For the parallel version. It is ignored in sequential versions. Tells FORM how big the patches of terms that are being distributed over the secondary processors should be. See also 5.92. If the number is too small, FORM will basically send individual terms. |
| SortType | Possible values are "lowfirst", "highfirst" and "powerfirst". "lowfirst" is the default. Determines the order in which the terms are placed during sorting. In the case of lowfirst, lower powers of symbols and dotproducts come before higher powers. In the case of highfirst it is the opposite. In the case of powerfirst the combined powers of all symbols together are considered and the highest combined powers come first. See also the on statement in 5.71. |

| | |
|---|---|
| TempDir | This variable should contain the name of a directory that is the directory in which FORM should make its temporary files. If the -t (or -T) option is used when FORM is started, the TempDir variable in the setup file is ignored. FORM can create a number of different temporary files. |
| WorkSpace | The size of the heap that is used by the algebra processor, when it is evaluating the substitution tree. It will contain terms, half finished terms and other information. The size of the workspace may be a limitation on the depth of a substitution tree. |
| ZipSize | This parameter defines the maximum size of a buffer for compression and decompression of parts of expressions with the ZIP algorithm. Currently this algorithm is not in use yet, but it is one of the relatively urgent future projects. ZIP compression can reduce intermediate files by a factor 4. |

Variables that take a path for their value expect a sequence of directories, separated by colon characters as in the UNIX way to define such objects.

The above parameters are conceptually relatively easy. The parameters that are still left are more complicated and are often restricted in their size by some relationships. Hence it is necessary to understand the sorting inside FORM a little bit before using them. On the other hand these parameters can influence the performance noticeably.

When terms are send to 'output' by the main algebra engine, they are put inside a buffer. This buffer is called the 'small buffer'. Its size is given by the variable *SmallSize*. When this buffer is full, or when the number of terms in this buffer exceeds a given maximum, indicated by the variable *TermsInSmall*, the contents of the buffer are sorted. The sorting is done by pointers, hence it is important that the small buffer resides inside the physical memory. During the sorting it may happen that coefficients are added. The sum of two rational numbers can take more space than any of the individual numbers, so there will be a space problem. This has been solved by the construction of an extension to the small buffer. The variable *SmallExtension* is the size of the small buffer together with this extension. The value for SmallExtension will always be at least 7/6 times the value of SmallSize.

The result of the sorting of the small buffer is written to the 'large buffer' (with the size *LargeSize*) as a single object and the filling of the small buffer can resume. Whenever there is not enough room in the large buffer for the result of sorting the small buffer, or whenever there are already a given number of these sorted 'patches' in it (controlled by the variable *LargePatches*) the buffer will be sorted by merging the patches to make room for the new results. The output is written to the sort file as a single patch. Then the results from the small buffer can be written to the large buffer. This game can continue till no more terms are generated. In the end it will be necessary to sort the results in the intermediate sort file. This can be done with up to *FilePatches* at a time. Because file operations are notoriously slow the combination of the small buffer, the small extension and the large buffer is used for caching purposes. Hence this space can be split in 'FilePatches' caches. The limitation is that each cache should be capable to contain at least two terms of maximal size. This means that the sum of SmallExtension and LargeSize must be at least FilePatches times 2*MaxTermSize*(bytes in short integer). It is possible to set the size of these caches directly with the variable *SortIOsize*. If the variable is too large, the variable FilePatches may be adjusted by FORM. If there are more than FilePatches patches in the sort file, a second sort file is needed for the output of each 'superpatch'. When the first sort file has been treated, the second sort file can be treated in exactly the same way as its predecessor. This process will finish eventually. When there are at most FilePatches patches in a sort file, the output of their merging can be written directly to the regular output. For completeness we give a list of all these variables:

| | |
|---|---|
| FilePatches | The maximum number of patches that can be merged simultaneously, when the intermediate sort file is involved. |
| LargePatches | The maximum number of patches that is allowed in the large buffer. The large buffer may reside in virtual memory, due to the nature of the sort that is applied to it. |
| TermsInSmall | The maximum number of terms that is allowed in the small buffer before it is sorted. The sorted result is either copied to the large buffer or written to the intermediate sort file (when LargeSize is too small). |
| SmallSize | The size of the small buffer in bytes. |
| SmallExtension | The size of the small buffer plus its extension. |
| LargeSize | The size of the large buffer. |
| SortIOsize | The size of the buffer that is used to write to the intermediate sorting file and to read from it. It should be noted that if this buffer is not very large, the sorting of large files may become rather slow, depending on the operating system. Hence we recommend a potential fourth stage in the sorting over having this number too small to fit more filepatches in the combined small and large buffer. Setting the small and large buffers to a decent size may avoid all problems by a: making more space for the caching, b: creating fewer file patches to start with. |

There is a second set of the above setup parameters for sorts of subexpressions as in function arguments or in the term environment (see 5.103). Because these things can happen with more than one level, whatever allocations have to be made (during runtime when needed) may have to be made several times. Hence one should be far more conservative here than with the global allocations. Anyway, those sorts should rarely involve anything very big. With the function arguments the condition is that the final result will fit inside a single term, but with the term environment no such restriction exists. The relevant variables here are subfilepatches, sublargepatches, sublargesize, subsmallextension, subsmallsize, subsortiosize and subtermsinsmall. Their meanings are the same as for the variables without the sub in front.

The default settings are

| | |
|---|---|
| bracketindexsize | 200000 |
| commentchar | * |
| compresssize | 30000 |
| constindex | 128 |
| continuationlines | 15 |
| dotchar | . |
| filepatches | 128 |
| functionlevels | 30 |
| incdir | . |
| insidefirst | ON |
| largepatches | 128 |
| largesize | 10000000 |
| maxnumbersize | 200 |
| maxtermsize | 2000 |
| maxwildcards | 100 |
| nwritestatistics | OFF |
| oldorder | OFF |
| parentheses | 100 |
| path | . |
| scratchsize | 3000000 |
| slavepatchsize | 10 |
| smallextension | 3000000 |
| smallsize | 2500000 |
| sortiosize | 100000 |
| sorttype | lowfirst |
| subfilepatches | 32 |
| sublargepatches | 64 |
| sublargesize | 1000000 |
| subsmallextension | 300000 |
| subsmallsize | 200000 |
| subsortiosize | 32768 |
| subtermsinsmall | 8000 |
| tempdir | . |
| termsinsmall | 50000 |
| workspace | 1000000 |
| zipsize | 32768 |

If one compares these numbers with the corresponding numbers for older versions one will notice that here we assume that the standard computer will have much more memory available than in the 'old time'. Basically we expect that a serious FORM user has at least 64 Mbytes available. If it is considerably less one should define a setup file with smaller settings.

More recently a new notation for large numbers has been allowed. One can use the characters K, and M to indicate kilo (three zeroes) and mega (6 zeroes) as in 10M for 10000000.

# Chapter 12

# The parallel version

This chapter is still to be written. The current version does not have all its necessary commands running yet (as specified in the moduleoption statement 5.56. Keep tuned!