Schoonschip

a program for symbol handling

Martinus J. G. Veltman
edited by
David N. Williams

January, 1991

First version: January, 1964
Current version: January, 1991
Last corrected: November 29, 2001

Contents

Changes from Earlier Versions

This version of Schoonschip (January 1, 1991) differs considerably from versions prior to January, 1989. Every effort has been made to avoid changing code that has been shown to work correctly, and to maintain upward compatibility as far as possible. To be able to run known problems and compare the results is invaluable. The present version builds on twenty-seven years of experience.

Modern users tackle ever larger problems, and the improvement of Schoonschip's handling of large problems was a high priority. There were several considerations, including first of all new ways to present and formulate problems. That led to the introduction of a new concept, that of character summation, which makes possible the very efficient generation of diagrams. As yet the main applications seem to be in high energy physics, but the concept appears to have greater generality. More specific to high energy physics is gamma matrix algebra. Efficient methods for dealing with gamma matrices in other than four dimensions have been incorporated. That allows the application of dimensional regularization to problems with fermions.

character summation

gamma matrices

Another important tool for handling large problems is the `Freeze` command. It existed in CDC versions of Schoonschip, and has now been implemented in this version.

`Freeze`

Last but not least, the output sorting methods of Schoonschip have been vastly improved. Previous versions tended to generate very large disk files, and consequently were quite time consuming. That seriously limited the magnitude of the problems that could be treated, especially on relatively small systems (PC's). It is to be emphasized that there is no reason in principle why Schoonschip should not run just as well on PC's as on mainframes. Of course the CPU speeds are different, but that is often more than compensated by the fact that mainframes usually deal with a multitude of users. And disk file handling is not necessarily slower on a PC. The main difference is that mainframes have essentially infinite disk space, but that advantage is rapidly disappearing as larger and larger disk systems become available to PC users. Right now

output sorting

the standard is about 40 megabytes, but there are already very good and inexpensive systems with capacities in the 100-200 megabyte range.

The new sorting methods give an improvement of one or two orders of magnitude in time and disk space for large problems (those producing disk files of more than 1 megabyte). Of course, "reasonable" default parameters are built in; but for truly large problems the Schoonschip parameters must be tailored to the problem. It is suggested that Chapter XIX, *Handling Large Problems*, be read at least once, just to get a feel for the situation; if the need arises one can go back for more details.

Several new statements and commands have been implemented. Some of them replace older, more cumbersome forms. All statements are summarized in Chapter XVIII, *Statements*, except for conditional statements and certain passive, type defining statements which have been categorized as "declarations." Substitution commands are discussed in Chapter XII, *Substitution Commands*.

New statements:

```
Anti            Fortran         Progress
Beep            Freeze          Rationalize
Digits          Large           Round
Directory       Maximum         Showl
Dostop          Nshow           Silence
Dryrun          Overflow        Synonym
External        Precision       Traditional
```

New conditional input statements:

```
Goto            Gotoifn
Gotoif          Set
```

New commands:

```
Absol       Gammas (old Trick,Trace)
Anti        Iall
All         Ndotpr
Beep        Order
Dostop      Print
Expand      Stats
Extern
```

Slight modifications:

- `Integ` on CDC versions of Schoonschip truncated numbers. Later versions rounded to the nearest integer. Truncation has been reinstated as the default, but the statement `Round on` may be used to obtain rounding to the nearest integer.

- `B` lists may now include vector names. All dot products or vector components relating to a vector in the list are factored out. There is a new, `Exclusive` option for `B` lists. If the first argument in the `B` list is the keyword `Excl`, then the names that follow will be the only quantities factored out; and in that case function names may also appear in the `B` list. The `Exclusive` option is especially helpful with the `Freeze` command. See Chapter IV, *Types*, and Chapter XIX, *Handling Large Problems*, for more details. `Excl`
- A `D` function with or without dummy arguments may now appear on the left-hand side of a substitution, although not combined with other factors.
- The `Count` command has been slightly enhanced, allowing multiple counts with one command.
- The `Symme` and `Asymm` commands have been slightly enhanced, allowing the handling of groups of arguments.
- The `Spin` command now allows vectors as arguments to identify spinors.
- The substitution types

```
Id,Funct,a= ...
Id,Funct,p(mu~)= ...
```

may now have function names between the keyword and the last argument. Then only those functions are affected. Example: *function names in substitutions*

```
Id,Funct,Epf,p(mu~) = k(mu) + q(mu)
```

New notation: Long names containing special symbols such as `+`, `-`, blanks, etc., may now be used for algebraic symbols. They must be enclosed in square brackets. Example: `[a+2/3*b]` is recognized as a symbol with that name. *long names* *square brackets*

Entirely rewritten: Chapter on *Conditional Substitutions*, (Chapter XI).

New chapters:

Conditional Input
Character Summation
Gamma Algebra
Statements
Handling Large Problems
File Handling Statements
R Input

It is suggested that one read (or try to read) the new chapters at least once, to get an idea of what is available.

Schoonschip has not had the reputation of being "user friendly," whatever that may mean. Error reporting, as well as syntax, has been a concern. A few changes have been made in error reporting; but while its importance was well recognized, it was not given the highest priority. On the other hand, a number of improvements in syntax have been made.

square brackets
`DELETE`
`Printer`
`Bdelete`
`Lprinter`
case sensitivity

There are a few minor incompatibilities with older versions. Square brackets `[ ]` now have a new function, and cannot be used any more as regular brackets. The statements `DELETE` (for deleting blocks) and `Printer` (to request line printer type output, that is, longer line length) have been replaced by `Bdelete` and `Lprinter`. That was necessary because there is no longer any distinction made between upper and lower case symbols for commands and statements. (Case sensitivity for symbol names, including those of built-in functions, and for block and do-loop names, has been retained.) There has also been an attempt to make the commands recognizable by the first few (four at most) characters.

conditionals

The condition code system has been thoroughly reviewed. Although quite powerful, it was cumbersome. Instead, there is now a system employing familiar logical constructs: `IF`, `AND`, `OR`, `NOT`, `ELSE` and `ENDIF`.

exponents
`Overflow`

The handling of exponents has been modified. Before, the addition of exponents was circular, going through 128 to -128. That is no longer true. Now there is an overflow error message, that can, however, be switched off (`Overflow off` statement).

Finally, there is a new, alternative notation for gamma matrices.

Acknowlegements: Thanks are due to Drs. E. Remiddi and J. Vermaseren for inspiring several of the improvements incorporated in this (and older!) versions of Schoonschip, and to S. Titard for helpful comments on the manuscript.

Chapter I

Introduction

Schoonschip is a program written in assembly language intended especially for handling large algebraic expressions. Very little is built in in the way of function properties, integral tables, etc.; but there are many facilities for operating on expressions. For example, if the user knows what kinds of integrands are going to occur it is easy to write procedures to do the actual integrations. Schoonschip never tries to think for the user, it just makes it possible to use the computer as a tool for doing algebra.

There are provisions for building up libraries of Schoonschip statements. Blocks of statements that perform well-defined operations (such as differentiation or integration) may be defined, stored, and recalled when needed with a single line. In this way the user can build an environment to suit his own needs.

The way the system works is basically like this. One gives Schoonschip an expression, followed by instructions that operate on it. Schoonschip first works out the expression term by term, and then each term passes through a series of levels. At each level the user has an opportunity to work on the term. The results are collected in an output storage area and like terms are added. The result at that point can be handled in a variety of ways: it can be returned to the input for a new round; it can be written to disk for later use, etc. Normally the result is also printed.

This manual explains the available features by means of examples. It is helpful to have some knowledge of how Schoonschip works, and details about that will be given at various points. The manual is meant to be read having at hand the file of examples, supposedly named `Examples.e`. To see, then, how Example 5 works, one types *examples file*

```
    Schip +5 Examples.e
```

output to a text file

If the output fills more than one screen, the user will probably want to specify a text output file, either by the `>filename` output redirection option available in many systems, or by simply giving the output file name as the last argument:

```
    Schip +5 Examples.e outp
```

The file `outp` can then be inspected later. Alternative ways to specify input and ouput are described in Appendix A, *Command Line Options*.

The input text of the examples is reproduced in this manual, but the output text generally is not.

This manual is intended to serve not only tutorial but reference functions as well. The reference functions are embodied in the Index, with index items echoed in the margins to make them easier to locate on the pages where they occur, in Chapter XXII, *Summary*, which is a “reference card” for Schoonschip, in Chapter XII and Chapter XVIII, *Substitution Commands* and *Statements*, which provide complete alphabetical listings of commands and statements with brief summaries of their actions, in the Table of Contents, and in the attempt to lay out the text for visibility. Note that Chapter XXII, *Summary*, Appendix A and Appendix B are intended for direct reference, and there are no references to them in the Index.

Chapter II

Basics

The first example shows what is meant by working out an initial expression. The expression must be given on a line with a Z in the first column, followed by a blank or tab in column 2. The general convention is that a line with a blank or tab in column 1 is a continuation of the previous line.

Z *expression*

continuation

Here is the start of the first example:

```
C Example 1.

Z XX = (a + b)^2
```

Note the C in column 1 of the first line. That denotes a comment line, which Schoonschip ignores except for printout. The notation ^ in the second line means exponentiation; **, as in Fortran, may also be used. XX is an invented name that identifies the expression in case it is to be reused, written to disk, etc. It may be part of another expression in later problems. All that Schoonschip does at this point is to work out the right-hand side to a series of terms: a^2, a*b, b*a and b^2. If no further operation is done on the expression, that is what will go to the output. For algebraic variables (which a and b are taken to be by default) commutation is assumed; thus a*b is recognized to be the same as b*a. Schoonschip handles that by rearranging every term, putting the symbols into an internally well-defined order. For quantities of the same kind, that is generally the same as the order in which they first appear in the input.

C *line*

exponentiation

ordering of variables

In the present case no further instructions are to be given, and the program may proceed to do its work. The signal for that is a line with a * in column 1. The remainder of the * line instructs Schoonschip what to do next, after this task has been performed. Right now the idea is to go on to another example, and we use the word begin to make Schoonschip start over. After the *begin line, a, b and XX are lost.

* *line*

*begin

```
*begin
```

grouping terms

For massive expressions, it is often desirable to group terms in the output. They may be grouped, for instance, with respect to the variable `a` and the exponents with which it occurs. The way to do that is to give a list of quantities to be factored out, on a line with `B` as its first character. At most 130 quantities are allowed on `B` lines.

`B` *line*

```
B a

Z XX= (a + b)^2

*end
```

`Exclusive`

There is an option to designate a `B` list as `Exclusive`, which means that *only* symbols in the list will be factored out. The syntax is, for example,

```
B Excl,a,b
```

which ensures that only `a` and `b` will be factored out. See Chapter IV, *Type Definition*, and the discussion of the `Freeze` statement in Chapter XIX, *Handling Large Problems*, for more details.

About the only other controls the user has over the output format are facilities that affect how numbers are printed. By default Schoonschip tries to write every number as a rational fraction, but that is not always possible. Internally all numbers are represented as floating point numbers, with a 14-bit exponent and a 96-bit fraction. When printing, Schoonschip tries to convert to a rational, using 70 bits as input. The rational is then reconverted to a floating point number, and agreement to 84 bits is required. If that works, the number is printed as a rational; otherwise it is printed as an integer or floating point number, depending on the number. The user has control over the number of decimal digits to be printed for floating point numbers. The rationalization procedure can also be modified. That is done by means of one or more of three statements:

internal numbers

```
Precision #
Rationalization #
Digits #
```

As often with Schoonschip syntax, only the first three (or if no confusion is possible, two) characters of these statements are significant, and they may be in upper or lower case. Thus

case sensitivity

```
PRE #
```

is acceptable instead of `Precision #`. The `#` character in this manual represents literal integer numbers (symbols or expressions not included), which

`#`

are usually written as decimal digits, but may also be written in hexadecimal. Schoonschip's hexadecimal notation is described later in this chapter.

`Digits` `Rationalization`

If `#` is absent in the above statements, the defaults are used. The number following `Digits` specifies the number of digits to be printed for floating point numbers. The default is 5. The number following `Rationalization` specifies how many decimal digits should be used (number of numerator plus denominator digits) when attempting rationalization. The default is 22. If 0 is specified, no attempt at rationalization is made. For other than the default value, no check on the remaining digits is done. That can be used to find rational approximations for numbers.

`Precision`

Numbers are always a problem in symbol handling programs, and the approach chosen in Schoonschip has grown out of many years of trying various possibilities. As already noted above, every number is put into a floating point format with a fraction of about 29 decimal digits (96 bits) and a 14-bit exponent. In the course of the calculation one will lose precision, and stray bits will accumulate in the least significant part of the number. Cancellations will thus not generally be exact. Schoonschip assumes that two quantities cancel if 25 decimal digits (80 bits) of the coefficients cancel. That number can be changed by means of the `Precision` statement, where `#` is the desired number of decimal digits. To be precise: `1.E15 + 1 - 1.E15` is 0 if `#` is 15, and is 1 if it is 16.

The above should make it clear to what extent numbers are treated exactly. In general, rationals will be converted properly if the sum of the number of decimal digits in the numerator and denominator is less than 21.

numbers

Numbers may be written in various ways:

- without exponent or fraction: `1230`, `17893`
- with fraction: `1.978`, `33.127`
- with (signed) exponent: `12E13`, `178E-17`
- with (signed) exponent and fraction: `23.47E15`, `18.49E-200`
- in hexadecimal notation: `0x10ABC`, `0xA4FFF`

`Directory`

The last two are the same as `68284` and `675839`. The hexadecimal notation is actually needed in certain statements (the `Directory` statement).

Schoonschip will work out multiplications and divisions of numbers directly; thus `2/3` is immediately converted to `0.6666666`...

`P` *line*

The effects of the above statements as well as some other features are shown in the following example. An expression is given, printed, and redirected to the input; and then some new instructions are given. Normally, when rerouting an expression to the input, no output is printed; and one must explicitly ask for printout by including a line with a `P` in column 1. Schoonschip echoes such a line to the output, prefixed with an additional > sign. The instruction that redirects the output is the word `yep` on the `*` line. Another way to include comments is also illustrated. A `!` simply means that the rest of the line is to be ignored. It should also be noted that blanks and tabs are generally ignored, except for their role as line continuation characters when they occur in column 1.

`*yep`

`!` *comments*

```
C Example 2.

P output          !Only the first two characters of 'output'
                  !are significant.

Digits 3
Rationalization 3          ! 0 leads to no rationalization
                           ! attempt.

Z XX=3.14 + 1/3*a1 + 7.1357689E20*a2    !E20 or E+20
                                        !implies 10^20.
*yep

Digits                     !This restores the default
                           !option, 5 digits.
Rationalization            !Default: 22 digits, with check.

*end
```

The `end` on the `*end` line signals termination after the problem has been worked out. As with most Schoonschip input, `*` lines are format free, except that the `*` must be in column 1. But one may write, for example, `* en  d`.

`*end`

Chapter III

Input Facilities

Schoonschip offers great flexibility in input. Two basic features are available: *blocks* and *do-loops*. A block is a set of lines that may recur with minor variations and that can be called with a single line. The structure is much like that of macros in assemblers. Do-loops are similar to Fortran do-loops.

1. Blocks

A block is delimited by two lines. The first line contains the word `BLOCK` in columns 1-5, followed by a name, followed by arguments separated by commas and enclosed in curly brackets. The last line must contain the word `ENDBLOCK` in columns 1-8. When the block is called, arguments that are enclosed between single quotation marks in the block definition (the double quote mark may *not* be used for this purpose) are substituted for. A block name may have 1-6 characters.

`BLOCK`

`ENDBLOCK`

block name

```
C Example 3.

BLOCK text{xx,yy}
C This argument is substituted in a call: 'xx'.
  This is not: xx.
  Arguments can be glued together: 'xx''yy'.
ENDBLOCK
```

To call a block, simply give the name with the arguments to be substituted. Be sure to start in column 1, and to enclose arguments in curly brackets. An example of a call:

```
text{yes,sir}
```

`tape9`

Blocks are stored on disk in a file called `tape9`, and are recalled as needed. Schoonschip keeps a list of the names in memory.

`BDELETE`

A block is remembered over `*begin` lines. It can be deleted by giving the word `BDELETE` starting in column 1 (four characters are significant, three if no confusion is possible, with lower case also accepted), followed by the name of the block to be deleted.

```
BDELETE text
```

˜ *line*

There is a simple way to delete entire lines when calling a block. A line with the character ˜ in column 1 is totally ignored by Schoonschip, not even printed. Suppose we want lines 1 and 2 of a block in one case, and line 3 in another case. Example:

```
BLOCK lines{A,B}
'A'C line 1
'A'  line 2
'B'  line 3
ENDBLOCK
```

Now call the block and use ˜ to delete lines:

```
lines{,~}

lines{~,}
```

Note that an empty argument leads to an empty substitution. Actually, if the final arguments are empty it is not even necessary to type commas. Thus, `lines{~}` gives the same result as the last example.

commas in block arguments

It may happen that one wants to call a block with an argument containing a comma. To do that, enclose the argument in curly brackets `{ }`. When substituting, Schoonschip peels away the outer layer of curly brackets. For example, `text{{a, b}, and c}` leads to:

```
C This argument is substituted in a call: a, b.
  This is not: xx.
  Arguments can be glued together: a, b and c.
```

2. Do-Loops

`DO`
do-loop name

Do-loops are also delimited by two lines. The first line must have `DO` in columns 1-2 followed by `name=`$\#_1$`,`$\#_2$`,`$\#_3$. The word `name` may have (as is generally the case with names in Schoonschip) up to six characters. The three numbers represent a range of numbers to be substituted for `name` in the rest of the do-loop. The first two specify the range, the last, the increment. The do-loop is terminated by a line containing the word ENDDO in columns 1-5. As in the case of blocks, substitution is done if name occurs enclosed between single quotation marks:

ENDDO

```
DO var=1,7,2
P input     !Normally only the first DO round is printed.
C value of var: 'var'
ENDDO
```

The numbers may also take negative values. Schoonschip handles that by attaching the prefix % to the number, which is interpreted as an intrinsic minus sign when encountered in front of a number. If it occurs elsewhere, the % sign is part of whatever it occurs with.

% *prefix for minus*

```
DO var=-1,1             !The third number is 1 by default.
P input
C value of var: 'var'.
C This may be part of a name, for example XX'var', or an
  index, for example YY(3+'var'). In the latter case
  Schoonschip will interpret the % sign as minus and
  correctly work out the number.
ENDDO
```

Premature exit from a do-loop may be forced at execution time by means of the substitution command Dostop (see Chapter IX, *Substitutions*, and Chapter XII, *Substitution Commands*). This is a somewhat tricky affair, due to the fact that do-loop expansion occurs strictly at the input level, while the Dostop command functions during execution, i.e., after the input has been read up to a line with a * in column 1. A Dostop command takes effect only after such a line. Forced exits should therefore normally be used only for do-loops that have * lines as their last statements.

forced exit

Dostop *command*

There is also a Dostop statement, which acts just before execution. It serves to preset the Dostop flag, and at execution time, a forced do-loop exit may thus be either nullified or set. Again, a * line should be the last before the ENDDO. The syntax of this statement is:

Dostop *statement*

```
Dostop on
```

or

```
Dostop off
```

This sets or clears the do-loop forced exit flag. Normally, a do-loop with a Dostop substitution command should also have a Dostop statement. The statement should occur preferably just before the first command of the set in which the `Dostop` command occurs.

nesting

Blocks and do-loops may be nested, up to 10 levels deep. As indicated above, this is all done strictly on the input level; only pure character substitution (apart from the `%` sign) is involved. It is up to the user to make sure that sensible Schoonschip input results. Use the statements `P input` and `P ninput` to see or to suppress the text that is actually generated.

`P input`
`P ninput`

3. Read Statement

Since the same blocks may occur in many different problems, it is useful to have the blocks stored in a file, and to let Schoonschip read them in before starting a problem. That can be done, and is in fact not restricted to blocks. A line with the word `Read` in columns 1-4, followed by a file name, causes Schoonschip to open the file and read it as normal input. When a line with the word `End` in columns 1-3 is encountered in the `Read` file, Schoonschip returns to the original input file. The input stream can be switched in this way any number of times. All blocks read in are stored in `tape9`, and can be used in all subsequent problems.

`Read`
`End`
`tape9`

Restriction: A `Read` within a `Read` file is not allowed.

Chapter IV

Type Definition

Several different types of symbols may occur in algebraic expressions in Schoonschip, and the user must generally declare the symbols and their types beforehand. The following types may occur: *symbol types*

- algebraic symbols
- indices
- vectors
- functions
- expressions

These are defined on lines having `A`, `I`, `V`, `F` or `X` in column 1.Actually there are additional types of expressions, characterized by `D` or `T` in column 1, but we will discuss those later.

1. Names

Names may have up to six characters, with both upper and lower case letters and numerals allowed, as long as the name is distinguishable from a number. Upper and lower case are distinguished. The underscore character (`_`) may also be used in names; and if it is not the first character, it is not included in the count. The underscore has a special function in certain contexts, to be explained later. *symbol names* *underscore*

There is a limited facility for handling long names. Long names containing characters other than those just mentioned can be very helpful for mnemonic purposes. Such names must be enclosed in square brackets. They can contain just about any character except control characters (such as a tab or a linefeed), or square brackets themselves. Their use is restricted to algebraic symbols. *long names* *square brackets*

Examples:

```
Abcd  abc_def  12A3  12_ab  f1  f2  [a + b/3]
```

Illegal names:

```
Abcdefg  12E3
```

The last name would actually be understood as a number: `12 * 10^3`.

2. Basic Algebraic Rules

exponents
short integers
`Overflow`

The most common symbols are simple commuting variables that may have an exponent. The exponents must be *short integers*, that is, integers in the range −128 to +127 inclusive. The extreme values are taken in case of overflow, if error trapping has been suppressed with the `Overflow` statement, to be explained below.

indices

Indices may also be used as algebraic variables, but they are really meant to be used as vector indices, or function arguments.

functions

Functions are treated as noncommuting quantities, whether or not they have arguments. If they are supposed to commute, that can of course be arranged; but if they represent matrices (with or without indices as arguments), then they ought to be treated as noncommuting. In short, functions may be used to represent anything that is not one of the other types. Whatever calculational rules they obey must be given by the user.

vectors
dot products
summation convention
Kronecker delta

Vectors are treated according to the standard rules of vector algebra. In particular, dot products may be formed, denoted by a capital `D` between two vector names. Thus, if `p` and `q` are vectors, then `pDq` denotes the dot product of `p` and `q`. Schoonschip follows the usual summation convention that repeated indices are summed: `p(i1)*p(i1)` is interpreted as `pDq`. The Kronecker delta is also available; it is the special function `D`, which is built in. Here is an example showing various rules:

```
C Example 4.

A a1,a2,a3                !Algebraic symbol list.
F f1,f2,f3                !Functions.
I mu,nu                   !Indices.
V p,q,k                   !Vectors.

Z XX = p(mu)*{ a3^-20*a1*q(mu) + a2*a1^7*D(mu,nu)*k(nu)

        + a3*f2(mu,k,q)*f1}     !Note the use of {}.
```

```
            + q(mu)*{a3*a2*a1 + f3*f2*f1}

    *end
```

Note that functions are always treated as if they are on the B list, to be factored out. (Unless Exclusive is mentioned at the head of the B list, in which case functions can be explicitly included in the list, and only those included are factored out.) Vectors not part of a dot product are also factorized. Note also the rearrangement of the symbols a1, a2 and a3, which is not done for functions such as f1, f2 and f3 (caution: except when the Excl option is used). Furthermore, Schoonschip pays no attention to the fact that f2 occurs both with and without arguments. As shown, curly brackets may be used instead of the usual brackets, but they must be paired. The user must also make sure that they cannot be confused with block definitions and calls. Special characters like *, +, etc. occurring in front of the { cause no problem in that respect.

factorization, ordering

Exclusive

curly brackets

Functions and vectors may not have exponents. For functions that is a mild restriction, because one can always add an argument, and consider that to be the exponent.

no exponent rule

Vectors may have numerical arguments, for example, p(3). This is taken to mean the third component of p, but only in a rather formal sense; and Schoonschip has no objection against, for example, p(-3). The only restriction is that the number must be a *short-short number*, that is, less then 32. These numbers are treated modulo 32, ranging from 0 to 31. Thus p(-3) becomes p(29). In the following, a quantity like p(3) will be called a vector component. If an index occurs all by itself, not as the argument of a vector or function, it is treated as an algebraic symbol. Thus one could have mu^3. Vector components and dot products may also have exponents.

vector arguments

short-short numbers

Like Fortran, Schoonschip has default assignments for names not mentioned in any list. The assignments may seem pretty obvious in a case like ff(a1,a2), but a case like p(k) is ambiguous. Is p a function or a vector? Again, as in Fortran, the default assignment for xx in a case like f(a1,xx) is to make it an index if the first character of xx is in the range I–N or i–n, and otherwise to make it an algebraic symbol. It is always possible to see what Schoonschip assumes by using the P list statement, which causes a printout of all the lists of all quantities in a given problem. There will be more about this statement later.

default types

P list

3. Exponents

algebraic symbols

The rules for algebraic symbols are the usual ones. They are supposed to commute with themselves and everything else, including functions. They may appear with or without an exponent (which must be a short number), and may be preceeded by a `/`, which simply means a sign reversal of the exponent.

exponents

Thus `a/b^3 = a*b^-3`. The exponent may also be an expression in brackets, provided the expression evaluates to a number when the problem is worked out.

essentially numerical expression

That number is then truncated and rounded to the nearest integer, which must be a short integer. Such an expression is called an *essentially numerical expression*. Thus `b^(2-2/3)` becomes `b^1 = b`.

Overflow occurs if the exponent of an algebraic expression exceeds 127 or is less than −128, in which case Schoonschip generates an error message. That

`Overflow`

can be suppressed with the `Overflow` statement:

```
Overflow off
```

Then an overflow (or underflow) causes Schoonschip to set the exponent to 127 (or −128), representing infinity. Error trapping can be turned on again with the statement:

```
Overflow on
```

`*fix`

The `Overflow` statement may be given in the `*fix` section (to be explained later), in which case the option given with it becomes the default.

vectors

The rules for dot products, vector components and indices not appearing in a vector or function argument are precisely those of algebraic symbols.

4. Reality Properties

The type lists are also used to assign further properties to the various objects.

complex conjugation

If complex conjugation occurs in a problem, reality properties have to be defined. An algebraic symbol may be real (the default), imaginary or complex. If it is complex, then its complex conjugate must be designated. That is done

`=i`

in the `A` list.If a symbol in the `A` list is followed by `=i`,it is taken to be imaginary; and the conjugate of the symbol is minus the symbol. A symbol may also be

`=r`

explicitly designated as real by the assignment `=r`. If the symbol is followed

`=c`

by `=c`, it is assumed to be complex; and Schoonschip creates a new name by

`A` *list*

appending `C` to the old name, and places that name next in its internal `A` list. For example,

```
A a1,a2=i,a3=c
```

means that `a1` is real, `a2` is imaginary and `a3` is complex. Schoonschip has a built-in complex conjugation operator, `Conjg`, which in this case yields `Conjg(a2) = -a2`, `Conjg(a3) = a3C` and `Conjg(a3C) = a3`. If desired, the symbol `a3C` created by Schoonschip can be renamed with the declaration Conjg

```
    Oldnew a3C=xxx
```

where `xxx` stands for the new name. Then `Conjg(a3) = xxx` and `Conjg(xxx) = a3`. The `Oldnew` declaration can actually be used to change any existing name, including the built-in names. Oldnew

Indices, vector components and dot products are taken to be real. Functions may also be real, imaginary or complex, and in addition *undefined*, which will be explained below. *undefined*

Except for `=c`, the definitions can be overridden; and the most recent definition holds. That is useful for changing the reality properties of built-in quantities. Overriding with `=c` has the effect that the next symbol in the list is considered to be the conjugate, which may be fine if it was not already defined to be something else. On the other hand, redefining an `=c` assignment leaves its conjugate symbol as real, so the conjugate symbol must be redefined explicitly if it is not to be real. *overriding reality*

5. Truncation

Algebraic symbols may also have truncation numbers assigned to them. The numbers must be between −32 and +32, and their function is to facilitate working with power series. If one assigns the number 4 to a symbol, for example, it is assumed that powers 4 or higher of the symbol can be ignored; and if a term containing the symbol to such a power arises it is immediately set to zero. Such a facility is very important for dealing with large expressions. In particular, it is extremely useful, if not indispensable, for substituting series into series. *power truncation*

Warning: Truncation numbers must not be assigned to symbols in certain situations where both negative and positive powers appear. For example, if 2 is assigned to `a`, then `a^3` may be deleted before it gets multiplied by, say, `a^-2`, giving an erroneous result.

In a somewhat similar way one may assign `z` to a vector. If the vector then occurs in a dot product, with an index, or as a vector component, it also is immediately set to zero. Function arguments are ignored in this process. That is ambiguous in a case such as `p(mu)*f1(...,mu,...) = f1(...,p,...)`; such =z *vector truncation* ambiguities must be resolved by the user, but do not seem to cause much of a problem in practise.

```
C Example 5.

A a1,a2=i,a3v=c,a4v=c,a5=i=3
V p,q=z,k

Z XX = Conjg(a1 + a2 + a3 + a4 + a5)
Z YY = Conjg(a3C + a4C)
Z ZZ = (a1 + a5)^5
Z AA = p(mu)*{p(mu) + q(mu)} + D(mu,3)*{p(mu) + q(mu)}
         + pDk

Oldnew a4C=b4,k=K

*end
```

Note how `a5^3` and higher powers have been eliminated. Note also that the vector `k` is replaced by `K`. The notation `D(mu,3)` is explained below.

6. Vector and Tensor Algebra

vector algebra
`D` *function*

The vector algebra rules are:

```
p(mu)*q(mu) = pDq
p(mu)*D(mu,nu) = p(nu)
D(mu,nu)*D(nu,la) = D(mu,la)
D(mu,mu) = n
```

`I` *list*
index range

In the above, `mu`, `nu` and `la` are indices supposedly defined in an `I` list. The quantity `n` is the range or dimension of the indices, and may be a number, but can also be symbolic. Thus in three-dimensional space, `n` would be 3; in four dimensions it would be 4. The default value is 4 (since 1905). The value can be specified in the `I` list. One writes the number after the index:

```
I mu=3,nu=3,la=3
```

numerical dimension

Numerical dimensions are treated as integers modulo 128, with values from 0 through 127, except that 0 is ignored in favor of the default dimension of 4. It should be noted that even if a range is assigned to a quantity in the `I` list, it can take on values outside that range in expressions where it is treated as an algebraic symbol.

Obviously one is not supposed to mix indices with different ranges in a `D` function. Schoonschip makes no check for that. If Schoonschip encounters

`D(mu,mu)`, the number from the `I` list is substituted. A symbol may be used in the `I` list instead, but it must be only one character in length. When Schoonschip encounters such a symbol in an `I` list, it immediately enters it into the algebraic symbol list, along with the same character followed by an underscore. For example,

symbolic index range

underscore

```
I mu=N,nu=N,la=N
```

If Schoonschip now encounters, say, `D(mu,mu)*D(la,la)`, the result is `N^2`. The names `N` and `N_` are both entered into the symbol table. The name `N_` is used extensively in gamma algebra with indices of symbolic range `N`.

index conventions

As an extension of these well-known rules, there is the additional convention that numbers and vectors may occur wherever indices may occur. Thus,

```
p(q),   D(p,q),   D(p,3),   D(3,4)
```

are perfectly legal, and are worked out as one might guess:

```
p(q) = pDq,   D(p,q) = pDq,   D(p,3) = p(3)
```

Finally, if n_1 and n_2 are numbers, then `D(`n_1`,`n_2`) = 1` if $n_1 = n_2$, and `= 0` if $n_1 \neq n_2$.

These conventions imply a rule for `D`, namely that

```
D(-x,y) = -D(x,y)
```

This applies when `x` is a vector or an index, but not when it is a number. The same rule applies to the second argument `y`.

If an index occurs as an argument of a function, the treatment is conventional:

```
D(mu,nu)*f(...,nu,...) = f(...,mu,...)
```

where now `mu` may be an index, vector or number, with the logical extension:

```
p(mu)*f(...,mu,...) = f(...,p,...)
```

index ambiguity

This seems fairly simple, but there is a complication that is really quite a problem. Consider the following expression:

```
D(mu,nu)*f1(...,mu,...)*f2(...,nu,...)
```

There are two possible answers, namely,

```
f1(...,mu,...)*f2(...,mu,...)
```

and

```
f1(...,nu,...)*f2(...,nu,...)
```

The answer is clearly not well defined. The two should in fact be identified. If two such terms arrive at the output, Schoonschip should add them up, to obtain `2*f1(...)*f2(...)`.

The solution to this problem is to rename the indices. Indices that are repeated summation indices, whose names are not really relevant, may be declared in a `Sum` list:

`Sum` *list*

```
Sum mu,nu
```

In both cases, Schoonschip then renames the indices in the same way, and the results are identical. The `Sum` list must come *after* the `I` list. It is best placed near (before) a line with a `*` in column 1. The indices appearing in a `Sum` list must have been defined before, but not in the `*fix` section (to be explained later). The following example illustrates the `Sum` declaration:

`Sum` *list rules*

`*fix`

```
C Example 6.

I mu,nu,la,ka

Z  XX = D(mu,nu)*f1(a1,a2,mu)*f2(a3,a4,nu)
      + f1(a1,a2,mu)*f2(a3,a4,nu)*D(mu,nu)

Z  YY = D(la,ka)*f1(a1,a2,la)*f2(a3,a4,ka)
      + f1(a1,a2,la)*f2(a3,a4,ka)*D(la,ka)

Sum la,ka

*end
```

Note how the indices `la` and `ka` are replaced by the invented name `Naa`.

7. Functions

function arguments

Next, we come to functions. Functions may have arguments, which must be single quantities separated by commas. There is an extension involving tables, to be explained later. Expressions in brackets are also allowed as arguments. As a special facility, functions may have characters as arguments. Only single characters are legal, and they must be prefixed with a `"`. Thus `f1("A,"s)` is legal. See the discussion of the `Compo` command in Chapter XII, *Substitution Commands*, for examples. Finally, one may have essentially numerical expressions, which, as described earlier, work out to short integers at execution time, after truncation and rounding. These expressions may include almost anything that works out to a number. They may even contain previously evaluated expressions; but such expressions must be “`Keep`” files (see Chapter VI,

characters

essentially numerical expression

`Keep`

Z Files), enclosed in brackets, which themselves give a single number.

A function containing an expression in brackets as an argument must somehow be replaced before it reaches the output store, because Schoonschip retains the definitions of expressions in brackets only until then. That means there must be an equation (or rather, a *substitution*, as it will be called) leading to the replacement of such a function, before the next * line.

Consider the following example of a function with legal arguments:

```
f1(a1,p,q(3),"X,pDq,17-22,(23*a5-b5),f2)
```

This function must be replaced before the * line because of the next to last argument. There is no such problem in the following case:

```
f2(a3,7-2,pDq,"A)
```

It is not legal to start with numbers outside the short number range occurring all by themselves. Thus

```
f1(1E25)
```

is illegal and produces an error. But `f1(2.341)` is accepted and changed into `f1(2)`. And

```
f1(1-7E25)
```

is accepted as input, but gives an overflow message at execution time.

This is pretty academic, and we mention it only for completeness. For a full understanding, one has to know what really goes on inside Schoonschip. Since numerical function arguments must be short integers, any numerical expressions that occur as function arguments must be "integerized." There is an operational symbol for that, namely, `Integ`. At input time, Schoonschip analyzes this situation very superficially, and when in doubt applies the `Integ` operator to the argument. That happens as soon as a / or *, etc., is encountered. Thus `f1(1E25-1)` is translated to `f1(Integ(1E25-1))`. When this expression is seen at execution time, the argument is normally rounded and truncated. Overflow occurs if the number is outside the range −128 to 127.

short integers

`Integ`

By default, `Integ` truncates a number towards zero, i.e., `1.9` becomes `1` and `-1.9` becomes `-1`. That action may be changed with the `Round` statement:

`Round`

```
Round on
```

Numbers are subsequently rounded to the nearest integer (i.e., `1.9` becomes `2`, `-1.9` becomes `-2`). The statement

```
Round off
```

restores the default option, truncation towards zero. In either case, a floating point number is first rounded in the last 10 bits, to allow for stray bits.

avoid composite arguments

A consequence of this implicit conversion to integers is that composite nonnumerical arguments are to be avoided; for example, `f1(3-a)` gives an error report, not upon input, but when execution occurs. Since it may well happen that the symbol `a` is a dummy, to be replaced by a number during the calculation, things have to be treated this way, and cannot be flagged at input time. That will become clear later.

`Conjg` *for functions*

The following rules for the complex conjugation of functions are built in. When `Conjg` is applied to a product of functions, Schoonschip

- reverses the order of all functions;
- conjugates those quantities whose reality properties are known, including the function arguments;
- postpones complex conjugation of functions that have been classified as `u` (for undefined) in the `F` list by the assignment `=u`.

`F` *list*
`=u`

`u` *function*

In this context, an expression in brackets is considered to be a `u` function.

```
C Example 7.

A a1,a2=i,a3v=c
F f1,f2=i,f3v=c,f4=u

Z xxv=conjg{ f1(a1,a2,a3)*f2(a1,a2,a3,f3)*f4(a1,a2,a3) }

*end
```

non-`u` *conjugation problem*

A special problem arises if an expression in brackets occurs as an argument of a non-`u` function. In such cases the expression in brackets is conjugated, that is, the operator `Conjg` is applied to it. Unfortunately, that is not possible for symbols of `u` type; thus, if a `u` function occurs as an argument of a non-`u` function subject to complex conjugation, then Schoonschip stops. In such cases the function of which the `u` function is an argument must be declared to be of `u` type, too, which is actually quite logical. A `u` function will generally be replaced by something else later on, and then the complex conjugation can be worked out.

8. Built-in Functions

`Epf` *function reality*

Several functions are built in, along with their reality properties; and here another problem arises. The `Epf` function (the totally antisymmetric Weyl tensor) is real in three dimensions, where it represents the cross product of vectors; but it must be considered as imaginary in four-dimensional Minkowski space (because one of the components must be 4). The default situation in Schoonschip is four dimensions, and thus `Epf` is considered to be imaginary. That can be changed by mentioning `Epf` in the `F` list with the option `=r`.

`X` *expressions undefined*

`X` expressions are always considered to be undefined with respect to complex conjugation. They are discussed in the next chapter.

built-in quantities

The built-in quantities are:

- **algebraic symbols**: `i=i`.
  This is the usual `i`, and the relation `i^2=-1` is built in, although not always instantaneously applied.
- **functions**: `D`, `Epf=i`, `G=i`, `Gi`, `G5=i`, `G6v=c`, `G7`, `Ug=c`, `Ubg`, `DD`, `DB`, `DT`, `DS`, `DX`, `DK`, `DP`, `DF=u` and `DC`.

Special rules are built in for these functions. Some of these functions must not be used, because they serve special purposes. The function `DD` is used internally for all kinds of things.

`D`*, Kronecker delta*

`Epf`*, Weyl tensor*

linearity

The function `D` is the Kronecker delta function and has been explained before. The `Epf` functon is the Weyl tensor, and it may have any number of arguments. If it has six arguments, it is assumed to be the Weyl tensor in six-dimensional space. A special command that reduces products of `Epf` to at most one `Epf` is built in. It is furthermore assumed that `Epf` is totally antisymmetric in all its arguments, and that it commutes with all other functions. Both the `D` and `Epf` functions are taken to be linear in their nonnumerical arguments, in particular, `D(-x,y) = D(x,-y) = -D(x,y)`, and similarly for `Epf`.

gamma matrices

Dirac spinors

The functions `G`, `Gi`, `G5`, `G6`, `G7`, `Ug` and `Ubg` refer to gamma matrices and Dirac spinors, which occur in relativistic quantum mechanics. They are discussed in Chapter XVII, *Gamma Algebra*. Users not interested in that subject may forget about them. They account for about 8% of the code in Schoonschip, not worth removing.

numerical functions

The functions `DB`, `DT`, `DP`, `DK` and `DC` are numerical functions that are useful in many situations. Their arguments are mostly short numbers (essentially numerical expressions), for which we use the notation m, n, n_1, …, j, j_1, …, etc., in the following.

factorial

binomial

`DB` with one argument is the factorial function, and `DB` with two arguments is the binomial function. They are not allowed to have negative arguments. The definitions are:

`DB(`n`) = `$n!$
`DB(`n`,`m`) = `$n!/\{m!(n-m)!\}$

Internally, the calculation of the binomial function is word (=16 bits) bound, and fails for `DB(19,`m`)`, $m = 8, 9, \ldots$ and higher.

theta function

The function `DT` is the theta function or step function, and is very useful for switching terms on and off. It may have any number of numerical arguments, is one if all arguments are positive or zero, and is otherwise zero:

```
DT(n1,n2,...) = 0   if one or more ni negative
              = 1   otherwise
```

permutation symbol

The function `DP` is the permutation symbol. The arguments are permuted until they are in ascending order, and the value of `DP` is 0 if any two arguments are equal, and otherwise it is ± 1, depending on the sign of the permutation:

```
DP(n1,n2,n3,...) = +1, -1 or 0,
                    0  if two numbers are equal,
                   +1  if the permutation to ascending order
                       is even,
                   -1  if the permutation is odd.
```

For example, `DP(2,1) = -1`.

numerical Kronecker delta

The function `DK` is something like a numerical Kronecker delta. It is one if the arguments are equal; otherwise it is zero:

```
DK(n1,n2) = 1   if  n1 = n2,
          = 0   otherwise.
```

The fact that it is explicitly a numerical function makes it different from the `D` function. For instance, `DK` may be part of a function argument expression as in `f1(...,4-2*DK(`n_1`,`n_2`),...)`.

conservation function

The function `DC` is unity if the sum of its arguments is zero, and is otherwise zero (it may have any number of arguments). It is primarily intended for situations where one has a conservation rule. It is also used for character handling, and then there are various rules, to be discussed in Chapter XVI, *Character Summation*.

summation function

The function `DS` is the summation function. Its application for character summation will be explained in Chapter XVI. Except for that, it must appear in the form:

```
DS(J,j1,j2,( F(J) ),( H(J) ) )
```

or

```
DS(J,j1,j2,( F(J) ) )
```

Here F and H are expressions that may depend on `J`. The quantities j_1 and j_2 must be essentially numerical expressions, with $j_1 \leq j_2$; and H must be a numerical expression (not limited to short numbers or integers), keeping in mind that the values of `J` will be integers. When working out this function, Schoonschip makes `J` run from j_1 through j_2, and creates a sum of terms. Each term is of the form

F`(`j`)*`G`(`j`)`

where G`(`j`)` is defined to be one for the first G (thus G`(`j_1`)`$= 1$), and subsequent G's are equal to the product of the previous G and the numerical expression H for that value of `J`. The second version of `DS` corresponds to $H \equiv 1$. In either case, if $j_1 > j_2$, the result is zero.

That sounds more complicated than it is. The expression H simply defines a recurrence relation among the coefficients of the sum. For example, 21 terms of the series for e^x are given by:

```
DS(J,0,20,(x^J),(1/J))
```

The recurrence relation G`(`j`) = `G`(`$j-1$`)*`H`(`j`)` for the coefficients gives $1/j!$ as coefficient G`(`j`)` of the jth term. This is a better way of handling the factorial, both from an economical and numerical point of view, than writing $1/j!$, that is, `1/(DB(`j`))`.

summation function restriction

There is an important restriction. If more than one `DS` function occurs in an expression, one must use a different name for the summation variable in each (`J` in the above example). Thus use `J1`, `J2`, etc.

```
C Example 8.

Digits 20
Ratio 0
Z xx = DS(J,0,20,(1.),(1/J))

*begin

Digits
Ratio
Z yy = DS(J,0,4,(X^J),(1/J))

*end
```

Chapter V

X, D and T Expressions

So far only `Z` expressions have occurred in the examples. They are expressions that Schoonschip evaluates. There are other kinds of expressions that may be defined but not immediately evaluated, which are called as they are needed in later calculations. Primary among these are `X` and `D` expressions.

1. X Expressions, Cosine Example

As a simple example, take the series for $\cos x$. The equation for $\cos x$ is cos

$$\cos x = \frac{\exp ix + \exp -ix}{2}$$

This series may be obtained as follows. First define $\exp y$, and then write $\cos x$ in terms of the exponential, arbitrarily keeping 6 terms: exp

```
C Example 9.

X EXP(y) = DS(J,0,6,(y^J),(1/J))

Z COS = { EXP{ (i*x) } + EXP{ -(i*x) } }/2

*end
```

Important rule: An `X` expression must be defined before it can be referred to. It is permissible to have `X` expressions appearing in other `X` expressions, provided they were defined before. `X` *expression order*

The rules for arguments of `X` expressions are the same as those for functions, which is why the argument `i*x` is enclosed in brackets. In this example it gets replaced on the right-hand side of the `EXP` definition, so that the expression in brackets indeed disappears as a function argument. That may seem like `X` *expression arguments* an inefficient method of defining COS, but it is actually a practical way to define it once and for all. In fact, it is possible to specify an argument for the Z expression, and to tell Schoonschip not to throw the expression away after printing it; one may then refer to COS(x) later as if it were also an X expression. How to do that will be discussed in Chapter VI, *Z Files*.

Z *expression arguments*

Arguments like those appearing in the definition of the X expression will often be called dummies in the rest of this manual. They are to be replaced by the arguments found in the call (in the Z expression) at execution time. In other cases some arguments may be dummies and others not, as we shall see. In those situations, a dummy has to be followed by a wiggle: ~. In X and D expression definitions, and also table (T expression) definitions, all arguments are dummies; and the wiggle need not be included.

dummy arguments

~ *dummies*

auto dummies

number allowed

One may have up to 500 X expressions.

2. D Expressions, Determinant Example

D *expressions*

D expressions are arrays of X expressions. They may be used, for example, to specify matrices. They are just like X expressions, except for the first argument. When a D expression is called, the first argument must be a number, which is then used to select the expression from the array. Since the expressions in X or D expressions may again be X or D expressions, the possibilities are quite extensive. For example, suppose there is a 3×3 matrix with elements (a11,a12,a13), (a21,a22,a23), (a31,a32,a33) in rows 1, 2 and 3. The first part of the following example produces the determinant of the matrix:

```
C Example 10.

B a11,a21,a31

D rc(n)=a11,a12,a13,
        a21,a22,a23,
        a31,a32,a33

X matrix(n,m)=rc(3*n+m-3)

Z xxx = DS{J1,1,3,(DS{J2,1,3,(DS{J3,1,3,
        { DP(J1,J2,J3)*matrix(J1,1)*matrix(J2,2)
           *matrix(J3,3) }

        } ) } ) }
*begin
```

3. Dummy Correspondence

As shown in the example, an X or D expression can have many arguments. In fact, X or D expressions may be arguments of X or D expressions. Here one must follow an important rule. To make it possible for Schoonschip to analyze the right-hand side of an expression correctly, a dummy must be of the same kind as the argument it is to replace. The "same kind" generally means one of four cases:

expression arguments

dummy correspondence

- symbols with or without an exponent
- vectors
- functions
- tables

Functions and expressions (and files, to be discussed later) are of the same kind. Algebraic symbols must be used as stand-ins for dot products, vector components, and also for short numbers (−128 to +127, the same as those that may appear as function arguments). Functions may stand for X or D expressions, as shown below in the second part of the example.

types

It is best to play this for certain, and really use dummies of the same kind as the arguments that will replace them, that is, X expression types for X expressions or D expressions, functions for functions, etc. In the example below, the use of ra, for instance, instead of f1, is advised in the definition of the X expression matrix. Vector components, dot products and numbers are illegal as dummies. If they are to be substituted in an X expression, use some algebraic symbol to represent them, as we did with n and m in these examples.

forcing dummy correspondence

4. Determinant Example Continued

In the second part of Example 10, we define a general X expression for a determinant, that may be called to compute the determinant of any matrix. Two determinants are computed.

3×3 determinants

```
F f1

B a11,a21,a31,b11,b21,b31

D ra(n)=a11,a12,a13,
        a21,a22,a23,
        a31,a32,a33
```

```
D rb(n)=b11,b12,b13,
        b21,b22,b23,
        b31,b32,b33

X matrix(n,m,f1)=f1(3*n+m-3)

X DET(f1) = DS{J1,1,3,(DS{J2,1,3,(DS{J3,1,3,
               { DP(J1,J2,J3)*matrix(J1,1,f1)
                  *matrix(J2,2,f1)*matrix(J3,3,f1) }
               } ) } ) }
Z detb=DET(rb)

Z deta=DET(ra)

*begin
```

$n \times n$ determinants

In the third part of this example, it is shown how the `X` expression `DET` can be made into a `D` expression that gives determinants for 1×1, 2×2 and 3×3 matrices.

```
F f1

B b11,b21,b31

D ra(n)=(a1-a2),0,
        0,(a1+a2)

D rb(n)=b11,b12,b13,
        b21,b22,b23,
        b31,b32,b33

D rc(n)=c11

X matrix(k,n,m,f1)=f1(k*n+m-k)

D DET(n,f1) =
   f1(1) ,

   DS{J1,1,2,(DS{J2,1,2,
      { DP(J1,J2)*matrix(n,J1,1,f1)*matrix(n,J2,2,f1) }
      } ) } ,
   DS{K1,1,3,(DS{K2,1,3,(DS{K3,1,3,
      { DP(K1,K2,K3)*matrix(n,K1,1,f1)*matrix(n,K2,2,f1)
         *matrix(n,K3,3,f1) }
      } ) } ) }

Z deta=DET(2,ra)
```

```
    Z detb=DET(3,rb)

    Z detc=DET(1,rc)

    *end
```

brackets in arrays

Several remarks are in order. First, note how items in a `D` expression array must be enclosed in brackets if the expression contains more than one term. The rule is that each entry must be one term only (an expression is a sum of terms, each term is a product of factors). An expression enclosed in brackets counts as one factor in a term. Secondly, there is the question of `DS` arguments. For the first argument, the summation index, one must use different variables for each `DS` function. Thus `J1`, `J2` and `J3` must not be used more than once. This was mentioned before, and in this example it is relevant.

terms, factors

5. Tables

`T` *expressions, tables*

`T` expressions, or better, *tables*, are for use as function arguments. There are also *character tables*, discussed in Chapter XVI, *Character Summation*. Tables represent arrays of quantities that may be selected as function arguments, and at execution time the appropriate argument is selected on the basis of the value of the first argument. The following example shows a complicated way of obtaining `f(a)+f(b)+f(c)`:

```
    C Example 11.

    T List(n)=a,b,c
    F f

    Z compl = DS(J,1,3,{ f(List(J)) } )

    *begin
```

The idea is really this: to enable summing over a series of arguments that can then be listed by using a table. The rules for entries in a table are simple: they must be valid function arguments. Tables can be nested, essentially allowing matrices of arguments. The first argument of the table in a call selects an item from the corresponding table list. If that item is again a table, then the second table list is looked up, with the second argument determining which argument to take. And so on. All arguments are treated as dummies, and table entries may depend on them. Since an expression enclosed in brackets may appear in the table list, complicated constructions are possible. If such expressions are essentially numerical, so that they must ultimately be converted to a short number, the operator `Integ` must be placed

table entries

nesting

auto dummies

essentially numerical arguments

`Integ`

Conjg

in front of them. `Conjg` is also allowed.

```
A x=c

T r1(n)=a11,a12,a13
T r2(n)=a21,a22,a23
T r3(n)=a31,a32,a33

T matr(n,m)=r1,r2,r3

T weird(n,a1,a2)=Conjg(a1+a2),Integ(3*a2)

X XX(a1,a2)=a2*a1

Z sqa13 = DS(J,1,3,{f1(matr(1,J))*f1(matr(J,3)} )

Z weirdo=XX(weird(1,x,7),weird(2,x,7))

*end
```

Conjg

There is a subtlety here. Whatever follows `Conjg`, enclosed in brackets, is considered to be an expression. It must not survive as a function argument beyond the `*` line. Thus it would have been illegal if in `weirdo` we had had `f1` instead of the `X` expression `XX`.

table entries

General rule: Anything that can occur as a function argument can occur as an element in a table list. That includes, for example, characters (notation: `"X` is the character `X`). Be careful with numbers; they are treated like function arguments, and nonintegers are truncated.

Chapter VI

Z Files

Z files very much resemble X expressions, but there is one important difference. A Z file contains the result of a Schoonschip calculation. A Z line defines a Z file, whose name follows just after the Z. A Z file may have arguments, can be stored in a variety of ways, and can be part of a problem, just like an X expression.

Z *file definition*

Schoonschip has been designed under the assumption that Z files tend to be large. Elaborate measures have been taken to allow them to be as large as possible. The number of Z files that can be handled at any one time is limited to about 200.

number of Z *files*

In earlier versions of Schoonschip, what we now call Z files were simply called files. The terminology "Z file" has been introduced to try to avoid confusion with the kind of disk file that is managed by the host operating system, and which of course Schoonschip also uses. In contexts where it seems pretty clear that an operating system file is not meant, we may revert to the old language.

1. Local and Common Files

First of all, Z files are divided into two categories, *local* and *common.* All of the Z expressions that have been shown so far are local. Local files are normally lost over a * line, except over a *yep. A local file may be kept over a *next line, provided it is mentioned in a Keep list:

local files
*yep
*next
Keep

```
Z xxx(a,b)=(a+b)^2
Keep xxx

*next
```

```
Z yyy=xxx(c,d)

*begin
```

`*begin`

After a `*begin`, the local files are all lost. Schoonschip treats local files as efficiently as possible. They are kept in memory if there is room, and if they are too big they are stored on disk and recalled as needed.

common files
`*end`

Common files are kept until they are explicitly deleted, or until the `*end` line is reached. They can be written to disk for use in a later session. Schoonschip keeps all common files in a sequential file on disk. Great care has been taken to see that such expressions are read with the least amount of input/output activity when needed in a problem. Normally the user will not notice these read/write activities, but he should nevertheless be aware of them when dealing with large problems.

simplifying expressions

Problems handled by computers have a tendency to become large; that is simply in the nature of computers. Except for the easiest cases, it is really quite impossible for a computer to simplify an expression. While it is trivial to simplify `a^2 + 2*a*b + b^2` to `(a+b)^2`, it becomes an entirely different matter when dealing with $1,000$ or more terms. Of course, with a little help from the user, simplification can be done, on a rather primitive level. If one recognizes a pattern such as the one above and substitutes `a^2 = apb^2 - b^2 - 2*a*b`, the output collapses to `apb^2`.

indexed `Z` *files*

Besides the classification into *local* and *common* categories, `Z` files may be *indexed* or *nonindexed*. That makes it possible to have arrays of `Z` files, much like `D` expressions. Indexed files are defined as follows:

local indexed files
index expressions

- A *local* file is indexed if its first argument on the defining `Z` line is a number in the range 0–32,767. There are provisions for allowing that number to be an expression, involving, for instance, do-loop or block variables; but at execution time such an expression must work out to a number. Only numbers and do-loop or block variables may appear in these expressions, and numbers must be substituted for the latter variables.

common indexed files
`Common`

- A *common* file is indexed if it appears in the `Common` list with an index, preferably the number zero. The implication is not the existence of the file with the particular index zero, but rather that the file name represents a class of subscripted common files.

Thus,

```
Common abc,Xyz,fgh(0),jkl
```

defines `abc`, `Xyz`, `fgh`, and `jkl` as common files, and `fgh` as an indexed common file. When subsequently found on a `Z` line, such files are flagged and written to disk at the end of the calculation (but are not kept beyond the `*end` line).

Local files may be preserved for a subsequent calculation by using the command `Keep`: `Keep`

```
Keep aaa,bbb,ccc(1),eee
```

This line, which should occur shortly before a `*next` line, causes the files in the list to be kept over the `*next`. For an indexed local file, one may explicitly mention an index (as with `ccc` above); and then only that file is kept. If several `ccc` files are to be kept, one need only mention `ccc` without an index; then all `ccc` files are kept. `*next`

2. Common File IO

There are several statements that relate to common files. First, common files may be written to a named disk file for use in a later problem. That can be done with the statement `Write`

```
Write xxx
```

All common files that exist at that moment are written (sequentially) to a disk file named `xxx`. Thus `xxx` will in general contain many common files. This statement must be all by itself, and not embedded in some problem. In other words, before and after the `Write` line there must be a `*` line not containing `yep` or `next`. `*yep` `*next`

When one or more of the common files contained in `xxx` is needed in a subsequent problem, one may `Enter` them: `Enter`

```
Enter xxx
```

This statement should be one of the first in a Schoonschip program. It must occur before a `*fix` line, which is explained in Chapter VIII, *Section Termination*. Schoonschip opens the disk file `xxx` and enters the names of the common files it contains into the common file table, and the files themselves are copied to a disk file local to Schoonschip (`tape7`). One may have several `Enter` and `Write` statements. `*fix` `tape7`

Note: Do not confuse `Enter` with `Read`. The latter is for reading normal Schoonschip text from an external file. `Read`

Common files may be deleted: `Delete`

```
Delete abc,Xyz,fgh(3)
```

Again, mentioning `fgh` without any index results in the deletion of all `fgh` files.

As the reader might guess, it is quite a problem to keep the name lists straight when common files are entered. Schoonschip normally treats that only when the common files are called in a particular problem. Usually there is no trouble, but difficulties can arise when there are summation indices (see the `Sum` declaration in Section IV.6). For that reason, it may be necessary to declare the names of the quantities contained in the common files (not the names of the common files themselves) before actually starting on the problem. The statement

summation index problem

`Names`

```
Names abc,Xyz,...
```

is meant for that purpose. In this example, it causes the name lists for the symbols that occur in the common files `abc`, `Xyz`, ... to be added to those already declared in the problem so far. If this directive is not given, and Schoonschip runs into a problem, it aborts, with an error message asking for the `Names` line.

3. IO Example

Example 12 shows a very simple case. Note that the local file `xxx` is lost over the `*begin` line, and `xxx` is taken to be a function. Similarly for the common file `ccc` after a `Delete`. When this problem is run, the disk file `fileC` is created. Make sure that there is not already a file by that name.

```
C Example 12.

P lists

Common yyy,ccc(0)

Z xxx(a,b)=(a+b)^2
Z ccc(3,a,b)=(a+b)^3
Z ccc(4,a,b)=(a+b)^4

Keep xxx

*next

F f1,f2
V p

Z yyy(p)=xxx(c,d) + p(nu)*f2(mu)
```

```
Sum mu,nu

*begin
Write fileC
*begin

V q

Z zzz=a1*yyy(q)
Z Abc=xxx(a,b)

Delete ccc,yyy          !The actual delete occurs when
                        !this section is done.
*begin

Z xyz=ccc(3,e,f)

*end
```

The statement `P lists` at the beginning of this example causes Schoonschip to print all the symbols it knows before execution, along with details such as reality properties or the dimensionality of indices. For `X`, `D` and `T` expressions, the type as well as the nesting depth is shown. Brackets count in determining the depth.

`P lists`

nesting depth

4. Large Files

At this point it must be emphasized once more that one needs to be conscious of the use of local and especially common files in a problem. If, for example, `yyy` is a rather lengthy common file, then factors such as `(yyy)^4` may lead to very lengthy calculations. If `yyy` has 100 terms, that gives rise to 100^4 terms, or 100 million terms that must be sorted. That would surely fill up all of the available disk space, to mention just one effect; and 100 terms is not really that much as such things go. To avoid that particular disaster, Schoonschip has a built-in output limit of about 0.5 megabytes per file. The limit can be changed by the statement

`Outlimit`

```
Outlimit,#
```

where `#` is the new limit in bytes. What to do when really big expressions must be evaluated will be discussed later. In this context, an expression is called big when it exceeds 50,000 terms.

big expressions

Chapter VII

Print Options

The following statements affect text output. Only the first two characters of the word following `P` are relevant.

`P brackets`	print working out of brackets
`P heads`	print only factors outside brackets (`B` line)
`P input`	print input lines (default)
`P lists`	print lists of all symbols (vectors, functions, etc.)
`P ninput`	do not print input lines
`P nlists`	do not print lists (default)
`P noutput`	print no output (default in case of `*yep`)
`P nstats`	print no statistics (default)
`P output`	print output (default, except in case of `*yep`)
`P stats`	print statistics

The following statements affect the format of the text output:

`Screen`	try to keep lines less than 80 characters long (default)
`Lprinter`	print up to 130 characters/line

There is also some control over the printing of individual `Z` files, but only if they are currently being processed:

`Print  xxx,yyy,`...	print `Z` files `xxx`, `yyy`,...
`Nprint xxx,yyy,`...	do not print `Z` files `xxx`, `yyy`,...
`Punch  xxx,yyy,`...	write `Z` files `xxx`, `yyy`,... to a file named `tape8` in Fortran-compatible form

`tape8`

The last statement is useful if the output is to be used for numerical input to a Fortran program, or for plotting curves, etc.

`P heads` Print options normally stay in effect until they are canceled. Exception: `P heads`.

`P output` `*yep` The `P output` option, often used to force output printing in a `*yep` situation, survives only one `*yep`.

Chapter VIII

Section Termination

The sections of a Schoonschip program are terminated by `*` line directives. Here are the possibilities:

`*fix`	Do not start execution. All names and expressions defined before this statement remain valid until the `*end` line.
`*yep`	Start execution. Write the output to disk and feed in again as input.
`*next`	Start execution. Keep all name lists, and those local files mentioned in `Keep` lists. A `Keep` list extends over one `*next` only.
`*begin`	Start execution. After that, start a new problem. Discard all names except those mentioned in the `*fix` section.
`*end`	Start execution. Stop after completion.

`Keep`

The `*fix` directive has not been explained up to now. It is used when one wants to define names, or `X`, `D` or `T` expressions, that are to be valid for all of the problems that follow, including those after a `*begin`. The `*fix` section must be the very first section, and is not allowed to contain any `Z` expressions. If common files are to be `Enter`'d, that must be done in this section.

`*fix`

`Enter`

The sequence shown above reflects the general structure of any Schoonschip program. First there is a `*fix` section, and then there may be several `*yep`, `*next` and `*begin` lines before the final `*end` is encountered.

program structure

The `*yep` command is extremely useful, but can only be explained properly later, after substitutions have been discussed. It is rather unique to Schoonschip, and is of crucial importance when dealing with large problems.

`*yep`

Chapter IX

Substitutions

Most of the discussion up to now has been about defining expressions, not working on them. The main tools for operating on expressions are called *substitutions* and *commands*. A *substitution* basically amounts to looking at an expression term-by-term and, if certain patterns match, removing something from the term and substituting something else. This fundamental tool, introduced for the first time in the 1964 version of Schoonschip, is the essential ingredient in symbol manipulation. *substitutions*

There are two distinct functions here, pattern recognition and substitution. Versions of Schoonschip since 1984 have made it possible to separate the two. One can first look for a pattern, setting a flag based on the result of the inspection, and then act on the flag at some later point.

Commands define certain operations to be done on the terms, that are of sufficient generality to warrant building them in. For instance, there is a command that allows the user to specify which functions commute with each other. Schoonschip implements that by putting the functions into an internally well-defined order, so that terms which differ only in the order of placement of commuting functions become equal. Such an operation could also be done with substitutions, but would be tedious. *commands*

1. Substituting for Patterns

An important concept for substitutions is that of dummies. These quantities have already been introduced in connection with `X`, `D` and `T` expressions; and here they become even more useful. Let us begin with a simple example. *dummies*

Suppose a quadratic polynomial in x

$$ax^2 + bxc$$

is to be integrated from x_1 to x_2. The integration amounts to using the rule $x^n \rightarrow x^{n+1}/(n+1)$, and then putting in the integration limits. That can be done as a substitution. First look for x to any power (this is the pattern). Then remove the pattern, and substitute the appropriate expression, which is the integral of x to that power. The following example shows how to do this:

```
C Example 13.

Z integr = a*x^2 +b*x

Id,x^n~ = x2^(n+1)/(n+1) - x1^(n+1)/(n+1)

*begin
```

Schoonschip looks for the pattern "`x` to any power," removes it if found, and inserts instead the right-hand side of the substitution above. The value of the power found is substituted for `n` in the expression. Here `n` is a dummy. To let Schoonschip know that it is a dummy, a wiggle (`~`) must follow directly after it on the left-hand side.

`Id` The notation `Id` refers to a name used often in the past for this operation, namely *identity*.

Consider now the complications that can arise with different kinds of patterns. As an example, take the same integral, but suppose there is a term without any x. What should Schoonschip do? In the present case the term should be treated as having a factor x^0, but there will be many cases where such terms ought to have no substitution made. In the spirit of pattern recognition, a term without x is taken not to fit the pattern `x^n~`; and such a term does not get worked on.

But how to deal with that in our case? A classic method is to use an extra symbol, which we call `dx` here:

```
Z integr = (a*x^2 + b*x + c)*dx

Id,x^n~*dx = x2^(n+1)/(n+1) - x1^(n+1)/(n+1)

Al,dx = x2 - x1

*end
```

What happens is that if dx along with x to any power excluding zero is encountered, then the first substitution is done. That leads to the removal of dx as well. If a term without an x (the c term) passes by, the first substitution is *not* done, and the dx survives. But then the second substitution is performed. The notation Al is an abbreviation for *also*, but there is more to it than that, as we explain further in Section 1 of Chapter XIV on *Internal Procedures*. Generally, Al is used to best advantage with mutually exclusive patterns. One should not use Al if the first substitution generates a pattern that the second would act upon, and care is needed in case the first substitution uses more than one level, as we explain further in Chapter XIV. Since there is a limit on the number of Id statements, it is often necessary to use Al; and its use needs to be well understood.

Al

There is an equivalent notation for Al, namely, placing the corresponding substitution on the same line as the first, preceded by a semicolon (;):

Al *equivalent*

```
Id,x^n~*dx = x2^(n+1)/(n+1) - x1^(n+1)/(n+1)  ;dx = x2 - x1
```

There may be several such *also* statements on one line, but lines should not exceed 150 characters.

multiple also's

line length

The above example shows just about all of the essentials of a substitution. Of course more complicated patterns than just an algebraic symbol to some power can be specified, but the principle is the same. There are substitutions that are performed if certain functions with certain arguments are present, and one may specify whether the order of those functions is important or not. One may operate on function arguments, on vectors in a dot product, etc. Basically, operations have been added through the years as they have been needed amd found useful. Most of them are quite natural, like the ones above.

2. Levels, Partial Integration Example

Here is another example, which shows several applications. In the integral above, more complicated terms could have occurred, such as $x^3 \cos x$. To evaluate these, one first does partial integrations:

partial integration

$$\begin{aligned} x^3 \cos x &= x^3 \frac{d}{dx} \sin x \\ &= \frac{d}{dx}\left(x^3 \sin x\right) - 3x^2 \sin x \end{aligned}$$

Before doing the x-integration, one does partial integrations as many times as necessary for the highest power of x that occurs. Assuming that at most x^{10} occurs, one would write:

```
Id,10,x^n~*cos(x) =   x2^n*sin(x2) - x1^n*sin(x1)
                    - n*x^(n-1)*sin(x)
Al,10,x^n~*sin(x) = - x2^n*cos(x2) + x1^n*cos(x1)
                    + n*x^(n-1)*cos(x)
```

There are more elegant ways to do this, using `X` expressions; but that is not the point here. A general method is shown in the latter part of Example 13, in the examples file.

levels

In the above lines, the substitutions are repeated on up to ten subsequent levels, as long as x to some power occurs. The final step, to be taken if no more powers of x are present, would be the integration of $\cos x$ or $\sin x$:

```
Id,sin(x) = cos(x2) - cos(x1)   ;cos(x) = - sin(x2)
                                          + sin(x1)
```

3. Conditional Substitutions

conditional substitutions

If negative powers of x also occur, one has to treat them separately. Here, *conditional substitutions* come in handy. One can test for the presence of negative exponents, and set a flag depending on the outcome of the test:

```
IF Multi,x^-1
```

`Multi`

`IF ...`
`ELSE ...`
`ENDIF`

This tests for exponents that are a multiple of the exponent shown, `-1` in this case. If any are found, a flag is set. If the flag is set, the substitution statements following the `IF` line are done, until a line with `ELSE` is encountered. If the flag is not set, the substitutions between the `ELSE` line and the terminating `ENDIF` line are done. A typical sequence would be:

`IF Multi,x^-1`
substitutions to treat `x^-n` *type terms*
`ELSE`
substitutions to treat `x^+n` *type terms*
`ENDIF`

`NOT`

A `NOT` may be inserted following the `IF`, and the following sequence gives identical results:

`IF NOT Multi,x^-1`
substitutions to treat `x^+n` *type terms*
`ELSE`
substitutions to treat `x^-n` *type terms*
`ENDIF`

IF's may be nested, up to eight deep. *nested* IF*'s*

A further extension uses AND and OR statements, possibly combined with NOT. For example, if there are also terms without $\sin x$ or $\cos x$: AND*,* OR*,* NOT

```
IF cos(x)
OR sin(x)
..IF Multi,x^-1
..substitutions to treat x^-n type terms
..ELSE
..substitutions to treat x^+n type terms
..ENDIF
ELSE
substitutions to deal with x^n without sin or cos
ENDIF
```

A new notation, purely for readability, has been introduced. A dot (.) in column 1 causes Schoonschip to skip all periods that follow on the line. In this way, one may indent nested IF's to show IF depth. See Chapter XI, *Conditional Substitutions*, for more details. *indentation of conditionals*

4. Survey of Patterns

A few of the patterns that can appear on the left-hand side of a substitution have been introduced above. The need for such patterns becomes clear as soon as one gains a little familiarity with problems of this kind. A set of patterns has been defined that appears adequate to deal with the situations that have arisen in practice. To give a complete description is somewhat tedious, but necessary. Let us do that in two rounds: first somewhat sloppily here, then more precisely in the next chapter.

4.a. Algebraic Symbols

As long as one deals with a single quantity (including an exponent or index or arguments) one has various options specified by keywords. One such keyword appeared already above, namely, Multi. For algebraic symbols, the following substitution patterns are possible, where n is a dummy, and n, m are numbers: *algebraic substitutions*

Id,a= ... Substitute n times if a^n is found with n positive.

Id,a^n= ... Substitute once if precisely a^n is found.

Id,a^n~= ... Substitute once for a, any nonzero exponent.

`Id,Multi,a^`n`=` ... Substitute m/n times if `a^`m is found, and m has the same sign as n; leave the rest.

`Id,Once,a^`n`=` ... Substitute once if `a^`m is found, with $m/n \geq 1$; leave the rest.

`Id,Funct,a=` ... Substitute for `a` if it appears as a function argument. The action taken is to enclose the expression on the right-hand side in brackets and insert it, instead of the argument `a`. Earlier remarks about the use of expressions in brackets as function arguments apply here. Such functions may not cross `*` lines, and must be subjected to some substitution before that.

`Id,Funct,f1,f2,...,a=` ... The same as above, but only in the specific functions mentioned.

Thus, `Id,Once,a^2=` ... leads to one substitution when `a^7` is encountered, leaving `a^5`.

`Al` Just as before, `Al` (or `;` on the same line) may be used in conjunction with `Id` for mutually exclusive patterns.

4.b. Single Functions

Similar things may be written down for dot products or vector components. For single functions, one may give the function, including arguments, some or all of which may be dummies; and one may also have repeated dummies. For example, *repeated dummies*

```
Id,f1(a~,b,c,a~)= ...
```

leads to a substitution if the function `f1` is found with `b` and `c` as second and third arguments, while any argument in the first or fourth place will do, as long as it is the same in both places. Here `a` is a repeated dummy. One could have specified another dummy for the last place:

```
Id,f1(a~,b,c,d~)= ...
```

Then the substitution is done for any arguments in the first and fourth places, including equal arguments.

`D` *substitution* The built-in function `D` is special insofar as it is not generally allowed as part of a substitution pattern. It may, however, appear all by itself, with or without dummies. Examples:

```
Id,D(al,be)= ...
Id,D(al~,be)= ...
```

For single functions, one keyword may appear. If a function has nothing but dummies as arguments, and if the keyword `Always` appears, then the substitution will be done even if the argument count is not the same. Thus

`Always`

```
Id,Always,f1(a~,b~)= ...
```

is also applied to `f1(xx,yy,zz)`. The argument `zz` is then simply dropped. Make sure that no case can occur where there are more dummies than arguments.

Finally, one may even specify a dummy function:

dummy function

```
Id,f1~(a,b,c~,...)= ...
```

The substitution is done for any function whose arguments and argument count agree.

4.c. Vectors

There are several possibilities for vectors:

vector substitutions

`Id,p(mu)= ...` Substitute for `p(mu)`.

`Id,p(mu~)= ...` Substitute for `p` with any index.

`Id,Dotpr,p(mu~)= ...` Substitute for `p` with any index if it appears in a dot product or as a vector component.

`Id,Dotpr,f1,f2,...,p(mu~)= ...` The same, but only in the specific functions mentioned.

`Id,Funct,p(mu~)= ...` Substitute if `p` appears as a function argument.

`Id,Funct,f1,f2,...,p(mu~)= ...` The same, but only in the specific functions mentioned.

In these cases things are worked out as one would expect for vectors. In all but the first case it is assumed that the right-hand side depends on the index `mu`. For example, one might have

```
Id,Dotpr,p(mu~)=k(mu)+q(mu)
```

Then suppose `pDq` occurs. Schoonschip reads this as `p(q)`, and substitutes `q` for the index `mu`, which leads to `kDq + qDq`, as it should. If `pDp` occurs, then Schoonschip invents an index, say `Naa`, substitutes this index for `mu`, and writes:

```
{ k(Naa) + q(Naa) } * { k(Naa) + q(Naa) }
```

A similar thing is done if `p` occurs as a function argument in `Funct`-type substitutions:

```
f1(...,p,...)
```

becomes

```
f1(...,Naa,...)*{ k(Naa) + q(Naa) }
```

which leads to

```
f1(...,k,...) + f1(...,q,...)
```

Thus the linearity of `f1` in `p` is assumed. This substitution should therefore not be applied unless that is the case.

4.d. Generalities

function order `Adiso` `Ainbe`

Substitutions are straightforward for other patterns involving more than one quantity. The only extra options that one has have to do with the ordering of functions. The keyword `Adiso` (*allow disorder*) means that the substitution must be done no matter how the functions are ordered. The keyword `Ainbe` (*allow in-between*) means that the substitution is to be done if the functions are in the same order, even if there are other functions in-between. If none of these keywords is present, then the substitution is done only when exactly the same pattern is found. The pattern may involve dummies for exponents, vectors, indices, functions and function arguments, and one may have repeated dummies.

absent factors

As a special option, one may also have factors `xx^0`, where `xx` is an algebraic symbol, vector component, or dot product. In that case, Schoonschip requires that `xx` specifically be absent.

Dummy correspondence: We stress here that one must (repeat: *must*) use the same kind of symbol for a dummy as the one which is to replace it at execution time. See the discussion on this subject in Section V.3, *Dummy Correspondence*.

substitution levels `*yep` *brackets*

Forty substitution levels are available. If more levels are needed, a line with `*yep` must be inserted. The level count is printed by Schoonschip next to every substitution. More levels are needed for a given substitution when the bracket depth is larger. It is often advantageous not to use brackets unless they are strictly necessary. For example, it is better to write

```
Id,x^n~ = x2^(n+1)/(n+1) - x1^(n+1)/(n+1)
```

which needs two levels rather than

```
    Id,x^n~ = (x2^(n+1) - x1^(n+1)) /(n+1)
```

which needs three levels. Schoonschip generates a substitution for every bracket pair. If you want to see what Schoonschip does, use the option `P brackets`.

`P brackets`

Chapter X

Patterns

In this chapter we give a more systematic discussion of substitution patterns. It is intended mainly for reference and for special situations, and may be skipped on a first reading.

Patterns in Schoonschip can be organized into about 17 classes. In practice one will rarely realize to what class a given pattern belongs; but we nevertheless discuss the classification, for completeness. For each pattern we describe the action that occurs when it is found, keeping in mind that that may be influenced, postponed or even inhibited by means of condition flags.

left-hand side substitution rules

The classification is based on what appears on the left-hand side of the `=` sign on an `Id` or `Al` line, or following `IF`, `AND`, `OR` or their negated versions. Only products of factors may occur (thus there is no `+` or `-` sign, except in exponents or function arguments); and no brackets may occur, other than those associated with a function and its arguments or a vector and its index. Thus, a pattern like `a+b` or `a*(c+d)` is illegal. Besides a pattern, a keyword may be given, which influences how Schoonschip reacts to the pattern. Finally, we will encounter *repeated dummies.* That means that the same dummy appears two or more times on the left-hand side of a substitution. In such a case, Schoonschip verifies when the pattern is matched that what appears in the places corresponding to the dummy positions is the same, whatever it may be.

repeated dummies

1. Classes

Here are the various classes, with examples given in the next section. In the following, `a`, `b`, ... stand for algebraic symbols, n, m, ... for short numbers, `p`, `q`, ... for vectors, `f1`, `f2`, ... for functions, etc.

0. *Pattern*: `f1(a~,b~, ...)` Function, all arguments nonrepeated dummies.

   Optional Keyword: `Always`

   Action: Substitute each time the function is found, provided the number of arguments agrees.

   If the keyword `Always` is given, then substitute irrespective of the number of arguments in the function found. It is up to the user to see to it that the number of arguments in the version of `f1` that is found equals or exceeds the number of arguments actually appearing on the right-hand side of the substitution.

1. *Pattern*: `a^`n Symbol with exponent, no dummy.

   Keyword: `Multi`

   Action: When `a^`m occurs, divide m by n, and if the result truncated to an integer comes out to be one or more, substitute that many times. The remaining exponent is put back. Thus `a^2` applied to `a^7` leads to three substitutions, and one factor of `a` is left. The exponent may be negative: `a^-2` applied to `a^-7` leads to three substitutions and a remainder of `a^-1`. If n and m have opposite signs, no substitution is done.

   If the exponent n is absent, the keyword `Multi` is understood, and an exponent of `1` is used.

2. *Pattern*: `pDq^`n Dot product with exponent, no dummy.

   Keyword: `Multi`

   Action: Same as in Class 1.

3. *Pattern*: `a^`n Symbol with exponent, no dummy.

   Action: Substitute once if exactly `a^`n is found.

4. *Pattern*: `pDq^`n Dot product with exponent, no dummy.

   Action: See Class 3.

5. *Pattern*: `p(mu)` Vector with index, no dummy.

   Action: Substitute once if `p(mu)` is found.

6. *Pattern*: `p(mu~)` Vector with dummy index.

   Action: Substitute for every `p(...)` for any index. The substitution does not apply if the vector `p` appears as a factor in a dot product or as a function argument, and also does not apply to vector components such as `p(3)` (i.e., if the index is a number).

7. *Pattern*: `a` or `mu` or `pDq` or `p(`n`)` Symbol (index, dot product, vector component).

   Keyword: `Funct`

   Action: Replace `a` (or `mu` etc.) when it occurs as a function argument.

   Note: Such functions must be substituted for before the `*` line, because an expression as a function argument cannot survive a `*`. If functions are mentioned between the keyword and the argument, then only those functions will be affected.

8. *Pattern*: `a^`n or `pDq^`n Symbol or dot product with exponent.

   Keyword: `Once`

   Action: Substitute once if `a^`m occurs with m/n positive and equal to or larger than one. Keep the remainder. Thus if the pattern is `a^2` and `a^7` occurs, there is one substitution and `a^5` remains.

9. *Pattern*: `a^`n`*` ...`f1(mu,b,` ...`)` One function and anything else.

10. *Pattern*: `f1(mu,a,b~,` ...`)` One function.

11. *Pattern*: Anything.

    Keyword: `Adiso`

    Action: Substitute if all the factors are found, even if the functions found are not in the same order as in the left-hand side of the substitution.

    Note: `Adiso` stands for *Allow disorder*.

    Note: The expression is put at the location of the first function removed.

12. *Pattern*: Anything.

    Keyword: `Ainbe`

    Action: Substitute if all factors are found with the functions in the same order, although there may be other functions in-between.

    Note: `Ainbe` stands for *Allow in-between*.

    Note: The expression is put at the location of the first function removed.

13. *Pattern*: Anything.

    Action: Substitute if all factors are found, with the functions in the same order and no other functions in-between.

    Note: The expression is put at the location of the first function removed.

14. Not used.

15. *Pattern*: `p(mu~)` Vector with dummy index.

    Keyword: `Dotpr`

    Action: Substitute for `p` if `p` is part of a dot product appearing with a positive exponent.

16. *Pattern*: `p(mu~)` Vector with dummy index.

    Keyword: `Funct`

    Action: Substitute if the vector `p` is found as a function argument. If one or more function names is mentioned between the keyword and `p(mu~)`, then the substitution will be done only in those functions.

All of this sounds a lot more complicated than it is. These classes cover most cases that one can imagine, but the user does not have to know to which class a given substitution belongs. Most substitutions are natural, in the sense that it is fairly obvious what actions they take.

2. Examples

`*fix`
`*begin`

We now give examples, essentially going through all the possibilities. The same starting expression will be used in all cases, and we use a `*fix` section to specify it. Then the several cases follow, all separated by a `*begin` for a fresh start.

```
C Example 14.

V p,q
A a,b,c,d,e,f,g,h
I mu,nu
F f1,f2,f3

X expr=f1(a,b,p)*f2(a,c,q)*f3(d,e)*f1(g,h)*
        { a^7 + a^-7 + a^2 + pDq^2 + pDp + p(mu) + p(nu) }
*fix

C Class 0, no keyword.

Z xxx=expr
Id,f1(a1~,a2~)=a1^10*a2^20
*begin

C Class 0, keyword Always.

Z xxx=expr
Id,Always,f1(a1~,a2~)=a1^10*a2^20
*begin
```

```
C Class 1, keyword Multi.

Z xxx=expr
Id,Multi,a^3 = xyz + hij
*begin

C Class 2, exponent 1, no keyword.

Z xxx=expr
Id,pDq = XYZ + HIJ
*begin

C Class 3, no keyword.

Z xxx=expr
Id,a^2 = a1^7/15
*begin

C Class 5.

Z xxx=expr
Id,p(mu) = - q(mu)
*begin

C Class 6.

Z xxx=expr
Id,p(mu~) = - q(mu)
*begin

C Class 7, keyword Funct.

Z xxx=expr
Id,Funct,a = a27
Id,f1(a1~,a2~,a3~) = 200*a1*a2*a3
Al,f2(a1~,a2~,a3~) = a1^10*a2^11*a3^12
*begin

C Class 8, keyword Once.

Z xxx=expr
Id,Once,a^2 = XXX
*begin
```

```
C Class 9.

Z xxx=expr
Id,a^2*f2(a1~,c,p~) = F2(a1,c,p)
*begin

C Class 10.

Z xxx=expr
Id,f1~(a,b~,p~) = F(a,b,p)
*begin

C Class 11, keyword Adiso.

Z xxx = expr
Id,Adiso,f1(g,h)*f1(a~,b~,c~) = F1(a,b,c,g,h)
*begin

C Class 12, keyword Ainbe.

Z xxx=expr
Id,Ainbe,f1(g,h)*f1(a~,b~,c~) = F1(a,b,c,g,h)
Id,Ainbe,f1(a~,b~,c~)*f1(g,h) = F2(a,b,c,g,h)
*begin

C Class 13.

Z xxx=expr
Id,f1(x~,b,p)*f2(x~,c,q) = F(x,b,p,c,q)
*begin

C Class 14, not used.

C Class 15, keyword Dotpr.

Z xxx=expr
Id,Dotpr,p(mu~) = - q(mu)
*begin

C Class 16, keyword Funct.

Z xxx=expr
Id,Funct,p(mu~) = - F1(a,b,mu)
*end
```

Chapter XI

Conditional Substitutions

`IF`, `ELSE`, `ENDIF`, `AND`, `OR` and `NOT` have already been introduced in earlier chapters. Here, we expand a bit on their use. The general structure is:

> `IF` *pattern*
> *substitutions*
> `ELSE`
> *substitutions*
> `ENDIF`

The `ELSE` is optional. The `IF` statement may be followed by `AND` and/or `OR` statements. `IF`, `AND` or `OR` may be followed by `NOT`, which inverts the action. Finally, the pattern may be followed by an `=` symbol and an expression. In such a case the substitution is actually done only if the pattern fits. The pattern may belong to any of the classes discussed in the previous chapter.

`ELSE` *optional*

`IF` *with* `AND`*,* `OR`*,* `NOT`*,* `=`

Within Schoonschip, `IF` statements set or clear a bit, one bit being reserved for each `IF` depth. Nesting is allowed up to a depth of eight, so eight bits are provided. Successive `AND` and `OR` statements act as filters; that is, whether the statement and those that follow it are processed, up to the next `ELSE` (if present) or `ENDIF` at the same nesting level, depends on how the bits are set at that stage. Here are the rules for setting the bits:

nested `IF`*'s*

> A successful `IF` (or unsuccessful `IF NOT`) causes the bit to be set. An `AND` (or `AND NOT`) statement will be inspected only if the bit is set. If successful (or not successful, for `AND NOT`), the bit remains set; else it is cleared. An `OR` (or `OR NOT`) statement will be inspected only if the bit is not set. If successful (or not successful, for `OR NOT`), the bit is set. The substitutions or commands following such a sequence will be inspected if and only if the bit

logical syntax

is set. The substitutions or commands following an ELSE will be inspected if and only if the bit is cleared.

ambiguous logic

Do not put any substitutions or commands between the IF, AND or OR statements if there is an ELSE. The reason is that substitutions and commands following an ELSE are on the same level as those following the preceeding IF, AND, OR group. The ELSE becomes ambiguous in such a case. Also do not put AND or OR after an ELSE, which again becomes ambiguous.

polynomial integration

The fact that an expression may be given on the IF, AND, OR lines allows for very compact code. For example, to integrate any polynomial in x (see Example 15):

```
IF NOT x^-1 = [Log(x)]
AND NOT x^n~ = x^(n+1)/(n+1)
Al,Addfa,x
ENDIF
```

`Addfa`

`Al`

The command `Addfa` attaches the factor given, `x` in this case. It will be executed only if none of the two previous substitutions applies. Note the use of `Al` (rather than `Id`) for the `Addfa` command. This may be done because the statement does not use anything that might be produced by the previous IF and AND statements.

Note the symbol (we stress, the *symbol*) `[Log(x)]`.

levels

Here are some comments about substitution level assignments for conditional statements. Schoonschip assigns IF, AND and OR statements to the same substitution level. IF is like Id, OR and AND are like an Al level assignement. A subsequent ELSE is assigned to the same level as the associated IF. The first substitution following an IF, AND, OR sequence may have Id or Al; the level is assigned as usual. If none of the IF, etc., substitutions actually performs a substitution, then one should use Al. However, if such a substitution needs the completion of the previous IF, etc., substitutions, then Id should be used. The same holds after an ELSE, where Id must be used for the first substitution following the ELSE, if that substitution needs the completion of the IF, etc., substitutions. Note that these last substitutions may be far up.

If an AND or OR substitution depends on the result of previous IF, etc., substitutions, then it should not have the Al type assignment that Schoonschip would give it. To force level alignment for such a case, simply put a line with Id before the AND or OR statement, as in

```
IF a1=2*(a1+a2)
Id
AND a1=(a4+a5)
```

Here is an example:

```
L 1     IF a3=2*(a1+a2)
L 1     AND a7=5*a1*(a7+a8)
L 3     Id,a1=0
L 4     Id,Count,x,a1,1

         ...

L 1     ELSE
L 3     Id,a1=3
        ENDIF

L 5     Id,...
```

The last level shown (`L 5`) is the maximum of: {level when reaching `ELSE`, level when reaching `ENDIF`}.

Compare with the case where `Al` is used:

```
L 1     IF a3=2*(a1+a2)
L 1     AND a7=5*a1*(a7+a8)
L 1     Al,a1=0
L 3     Id,Count,x,a1,1

         ...

L 1     ELSE
L 1     Al,a1=3
        ENDIF

L 4     Id,...
```

Chapter XII

Substitution Commands

Certain operations are difficult or impossible to do with simple substitutions, and a number of facilities have been built in that deal with special situations. They are generally called *substitution commands*, or *commands* for short, and they perform actions at the levels where they are placed. They generally require at least one level, and sometimes the number of levels they use affects performance. The most common syntax for a substitution command is:

commands and levels

command syntax

`Id,`*keyword*`,`*parameters separated by commas*

in which case Schoonschip uses a number of levels that depends on the command. An alternate usage is

`Id,#,`*keyword*`,`*parameters separated by commas*

where `#` is some number < 40, in which case Schoonschip sets aside `#` levels for the command. The `Gammas` command is one for which this is especially relevant.

`Gammas`

For example, suppose that some function is symmetric in its arguments, say `f1(a,b,c)` = `f1(b,a,c)` = ... In such a case one would like the arguments to be arranged always in the same order. The command

`Symme`

```
Id,Symme,f1
```

causes Schoonschip to scan every instance of the function `f1` that passes by at the current level, and to rearrange its arguments in a well-defined way (first-defined symbols first). This is much easier than the perhaps very long set of substitutions that would do the same thing. One would have to know beforehand what arguments could appear, and then go through all the combinations.

1. List of Commands

command names

For commands, only the first four characters of the name are relevant, the first three if no confusion is possible. They may be either upper or lower case. The following commands are available:

`Absol`	Make the sign of the numerical coefficient plus.
`Addfa`	Add factor.
`All`	Round up (collect) all appearances of a vector, or appearances of all vectors in a specified function.
`Anti`	Scan functions for characters with an underscore (`_`).
`Asymm`	Function is antisymmetric under interchange of arguments.
`Beep`	Send CTRL G to the terminal (normally a beep).
`Coef`	Inspect numerical coefficient.
`Commu`	Rearrange commuting functions.
`Compo`	Compose names from characters.
`Count`	Count certain occurrences.
`Cyclic`	Function is invariant under cyclic permutation of arguments.
`Dostop`	Clears, sets or tests the do-loop forced exit flag.
`Epfred`	Reduce products of `Epf` functions.
`Even`	Function is even in sign of arguments.
`Expand`	Expand frozen files.
`Extern`	Call user defined subroutine.
`Gammas`	Throw the book at gamma matrices.
`Iall`	Inverse of `All`.
`Ndotpr`	Construct anomalous dot products (for use with gammas).
`Numer`	Insert numerical values.
`Odd`	Function is odd in sign of arguments.
`Order`	Order functions based on the first two arguments.
`Print`	Print message to standard output.
`Ratio`	Rationalize.
`Spin`	Sum over spins of spinor functions `Ug`, `Ubg`.
`Stats`	Count how often this command is called.
`Symme`	Function is symmetric under interchange of arguments.

2. Internal Ordering

Internal ordering in Schoonschip is always done on the basis of "smallest first," not only for the purposes of the commands of this section, but also for the printout of quantities not mentioned in the `B` list. That means the following ordering for quantities of different type: *type order*

- index
- vector
- function
- algebraic symbol
- character
- number (short number)
- vector component
- dot product

For quantities of the same type, the ordering is according to when a quantity is first encountered. For dot products, the earliest defined vector counts first. To guarantee a certain ordering, the quantities can be given in the appropriate list in the desired order.

number ordering The ordering for numbers is not entirely what one might expect: 0 before positive numbers up to 127, then $128 = -128, -127, -126, \ldots, -1$. This is because of the two's complement notation for numbers.

3. Description of Commands

We now discuss the commands one by one. The general format or *command template* is shown first, and examples sometimes follow the discussion. Optional command arguments are indicated in a few templates by square brackets [ ] in the normal text font. Commands may also occur in `IF` ... `ELSE` ... `ENDIF` statements, but what success or failure means is often not well-defined. As a rule, success means "some action has been taken." For example, `Addfa` always gives success.

`Id,Absol`

The numerical coefficient of the term inspected is forced to be positive. This command needs no levels.

`Id,Addfa,`*expression*

The expression specified is attached as a factor. This command is mainly intended for use with conditional statements; the expression can be attached depending on previous actions. There are no particular limitations on the sorts of expressions that can occur here; anything allowed in a `Z` expression or on the right-hand side of a substitution is allowed, except that there must be no dummies.

```
C Example 15.

C Integration of a polynomial.

A a,b,c,d,e,x

Z xxx = a*x^2 + b*x + c + d/x + e/x^2

IF NOT x^-1=[Log(x)]
AND NOT x^n~=x^(n+1)/(n+1)
Id,Addfa,x
ENDIF

*end
```

<u>`Id,All,p,[N,]F[,"F_][,"D_][,"V_]`</u>

<u>`Id,All,Fx,[N,]Fq`</u>

The command `All` (and its inverse `Iall`) has two different syntaxes. In the first syntax, the first argument is a vector, which must have been declared before. The list may be followed by optional character arguments `"F_`, `"D_` or `"V_` (always separated by commas, and possibly an index list, see below), which inhibit certain actions.

In the second syntax, the first argument is a function, which must have been declared before.

The parameters `N` and `F` must have been declared previously as algebraic symbol (it must be a one character symbol) and function. The dimension parameter `N` may be absent.

First syntax: If any occurrence of the vector `p` is found, the function `F` is appended, with arguments that depend on the context in which `p` is found:

- If `p(mu)` with any index `mu` is found, then `p` is removed and the argument `mu` is added to the argument list of `F`.
- If `pDq` is found with any `q` (except `p`) and a positive exponent, say `pDq^`n, then `q` is added as an argument of `F` n times.
- If `pDp` is found with a positive power, say n, then n new indices are created. Each index is added twice as an argument of `F`.

- If p is found as the argument of any function, then an index is created. The new index replaces p as the function argument, and furthermore the index is added as an argument of F.

In the above, whenever an index is created it is given the symbolic dimension N, or the default dimension (=4) if no N is specified. The main use of this command is to facilitate integration with the vector p as integration variable.

The optional character arguments inhibit actions as follows. If "F_ occurs, no function arguments are inspected; "D_ inhibits the inspection of dot products; "V_ turns off inspection for single vectors. In the example

```
Id,All,p,N,F,"F_,"V_
```

only dot products are considered.

Second syntax: All arguments of the function Fx are scanned. When a vector argument is found, it is replaced by a created index. The function Fq gets two new arguments, namely the vector and the created index. In the final result the arguments of Fq are arranged with all the vectors first, followed by all the corresponding indices. For example, All,Fx,N,Fq applied to Fx(p,k,q) gives Fx(Naa,Nab,Nac)*Fq(p,k,q,Naa,Nab,Nac).

```
C Example 16.

V p,q
F F
I mu=N,nu=N

Z xx= pDp^2 * pDq^3 * F1(p,p,q,p) * p(mu) * q(nu)

Id,All,p,N,F
P output

*yep

C Showing the dimensionality of the created indices:

Id,F(i1~,i2~,i3~,i4~,i5~,i6~,i7~,i8~,i9~,i10~,i11~)=
   F(i1,i2,i3,i6,i7,i8,i9,i10,i11)*D(i4,i5)

*begin

C A more realistic example.

I al,be,mu,nu
A N,N_
V p,k
F F,F20,F22
```

```
Z xx = G(1,1,al,be,p,al,be,p)

Id,All,p,N,F
P output

*yep

Id,F(i1~,i2~) = D(i1,i2)*F20(k) + k(i1)*k(i2)*F22(k)

*end
```

It should be clear that the `All` command generally creates indices. Such indices are often renamed by Schoonschip, in order to produce a well-defined result in the output. Sometimes the renaming is undesirable; in particular a subsequent `Freeze` command can give trouble if a created index occurs in a pair, one outside and one inside a bracket. To handle such cases, one can assign a list of indices to be used by the `All` command, after the last function (and the possible `V_`, etc.). These indices must have been declared in the `I` list, with the same dimension as that specified in the `All` command.

`Freeze`

For example,

```
I al1=N,al2=N,al3=N
   ...
Id,All,p,N,Fx,D_,al1,al2,al3
```

It may be that more indices are needed than are given in the list. There are two options here. In the above case, Schoonschip will issue an error message when the list is exhausted. If instead the list is terminated with the character `"C`,

```
Id,All,p,N.Fx,D_,al1,al2,al3,"C
```

then Schoonschip will create further indices as they are needed, after using up `al1`, `al2`, `al3`.

<u>`Id,Anti,TA`</u>

All function arguments are inspected for character arguments (`"X`). If a character with underscore is found, the character table `TA` is consulted. If the character appears in that table, the underscore is removed. This command will be discussed further in Chapter XVI, *Character Summation*.

<u>`Id,Asymm,F1,2,3,4,F2,F3,4,5,...`</u>

The `Asymm` command specifies that `F1` is antisymmetric in arguments `2`, `3` and `4`, `F2` in all arguments, and `F3` in arguments `4` and `5`. If the arguments are not in “first defined first” order, they are permuted. If the permutation is odd, a minus sign is attached. If a number in the list is larger than the number of arguments in the function it follows, it is ignored. When `Asymm` is used in a conditional, success is flagged when a permutation is actually done; no permutation means failure.

There is a second syntax for this command that can be used when the arguments occur in groups:

```
Id,Asymm,F1:2,3,4:7,8,9:12,13,14:
```

The groups of numbers within colons denote argument groups. They need not to be sequential. Only the first argument of a group is used in the comparison. If, however an interchange is needed, both groups are interchanged as a whole. For example, if in the above case argument `7` is to be placed before argument `2`, then `2` is interchanged with `7`, `3` with `8` and `4` with `9`.

If the arguments are sequential, an even shorter syntax is allowed:

```
Id,Asymm,F1:1-3:7-9:12-14:
```

The number of arguments in each group must of course be the same.

Note: For symmetric functions see the command `Symme`.

```
C Example 17.

V p,q,k
I mu
A a,b,c,d,e,f,g,h
F F1,F2,F3

Z xx = F1(e,d,c,b,a) + F2(e,d,c,b,a) + F3(e,d,c,b,a)
          + F2(-125,-30,-1,0,30,125)
          + F2(pDq,pDk,kDq,p(3),q(2),F2,7,mu,p,a)
          + f1(e,f,g,d,a,b,c,h,k)

Id,Asymm,F1,2,3,4,F2,F3,4,5
Al,Asymm,f1:1-3:5-7:

*end
```

`Id,Beep,#`

This command sends a CTRL G to the terminal, which normally results in a beep. It may be used to audibly signal certain occurrences. Be careful though; one quickly gets a lot of noise. The number `#` limits that—at most `#` beeps will be issued. Here `#` must be positive and less than 128. Example:

```
IF a^10
Al,Beep,3
ENDIF
```

`Beep` *statement*

The `Beep` statement (not this command) may be used to generate some noise upon termination, or if an error occurs.

<u>`IF Coef,`*option*</u>

The command `Coef` is specifically for use with conditional substitutions. It essentially inspects the numerical coefficient of any term passing by. Of course, this command can also be used with `AND` or `OR`, with or without a `NOT`.

`Coef` *options*

The possible options are:

`pl`	plus
`ng`	negative
`eq,#`	equal
`lt,#`	less than
`gt,#`	greater than
`ib,`$\#_1$`,`$\#_2$	in-between

`eq`

`Precision`

Here `#` represents a number (any number, not just a short number). When a term passes by, the coefficient is inspected or compared, and the `IF` bit is set or cleared depending on the result. In the case of `eq`, two numbers are considered equal if they agree within the number of decimal digits specified by the `Precision` statement (default = 25). The correspondence between how the `IF` bit is set and the naming of the options is the following:

$$\texttt{IF Coef,}\mathit{option}\begin{cases} \mathit{yes} & \text{set } \texttt{IF} \text{ bit (or clear if } \texttt{IF NOT}\text{)} \\ \mathit{no} & \text{clear } \texttt{IF} \text{ bit (or set if } \texttt{IF NOT}\text{)} \end{cases}$$

```
C Example 18.

A a1,a2,a3,a4,alt1,agt2,aeq15,ai12

Z xxx= 0.5*a1 + 1.5*a2 + 2.5*a3 + 1.E-20*a4

IF Coef,lt,1.       !Add factor alt1 if coefficient < 1.
Id,Addfa,alt1
ENDIF

IF Coef,gt,2.       !Add factor agt2 if coefficient > 1.
Id,Addfa,agt2
ENDIF
```

```
IF Coef,eq,1.5      !Add factor aeq15 if coefficient = 1.5
Id,Addfa,aeq15
ENDIF

IF Coef,ib,1,2      !Add factor ai12 if coefficient
Id,Addfa,ai12       !in-between 1 and 2.
ENDIF

IF Coef,lt,1E-15    !Delete the term if coefficient
Id,Addfa,0          !< 1E-15.
ENDIF

*end
```

The last lines of this example show how the `Coef` command can be used effectively to eliminate "dirt."

<u>`Id,Commu,F1,F2,F3, ...`</u>

commuting functions

The command `Commu` is used for ordering commuting functions. The ordering is done with respect to:

- the name of the function (smallest first, see Section 2 of this chapter)
- the number of arguments
- the arguments

If the commuting functions occur nested between other functions, they are removed and placed after the other functions. Thus the `Commu` command may have an effect even if only one function is named, and even if that function occurs only once in a given term.

```
C Example 19.

A a1,a2,a3,a4,a5
F F1,F2,F3

Z xx = F3(a1,a2)*F1(a1,a2,a3)*F1(a4,a5)
 + F1(a1,a2,a4)*F2(a1,a2,a3)*F1(a1,a2,a3)
 + F3(a1,a4)*F2(a1,a2,a3)*F1(a1,a2,a4)*F2(a3,a4,a5)

Id,Commu,F1,F3

*end
```

The `IF` bit is set if any of the functions named is present.

Id,Compo,["X,][TA,]<*options*>,F1,F2,<*options*>,F3,F4, ...

The `Compo` command is quite complicated, and it is beyond the scope of this manual to describe realistic applications. We nevertheless give a complete description of the command itself. It does character manipulation, and can be used to generate new symbols, including functions. There is a way to correlate the name of a function created in this way with specified arguments. When combined with the use of tables (tables can also have characters in their lists), it is a powerful tool for generating large numbers of symbols, or functions as well, that reflect a topology. For a very simple application, see Example 21, which demonstrates the command `Count`.

`Compo` *with tables*

The character argument `"X` may be absent, or it may be `"S` or `"S_`. The character table argument `TA` may also be absent. If present, it specifies characters for which `"X_= "X` (antiparticle = particle). If not present, the default is either none, or characters defined by means of the `Anti` statement.

`Anti`

Briefly, `Compo` takes the character arguments found in `F1`, etc., permutes them into a well-defined order (unless `"S_` is specfied for `"X`) and at the same time permutes argument groups in `F1`. The characters are then glued together to make a name, which may be identified with an existing name, or entered into a name list if it doesn't exist; or the term may be discarded, depending on what options have been specified.

The functions `F1`, `F2`, ... are scanned. The arguments are expected to have the following structure:

```
F1(c1,c2,c3, ...,cn,/,d1,d2, ...,dm,*,arg1a,arg1b, ...,
   *,arg2a,arg2b, ...,*,argka,argkb, ...)
```

Here `c1`,...,`cn` and `d1`,...,`dm` are either characters prefixed by `"`, in which case the legal characters are the upper and lower case characters and the numerals, possibly followed by an underscore, or they may be vectors, in which case the character(s) of the vector name without a `"` are used. Either `n` or `m` may be zero, but not both. If `m` is zero, the `/` may be omitted. The arguments `arg1a`, `arg1b`, etc., correspond to character `c1`; `arg2a`, `arg2b`, etc., to character `c2`; and so on. The sum of `n` and `m` must not exceed 6 (not counting the underscores), and must not exceed 2 if a vector name is to be constructed.

character arguments

The characters `c1`, ..., `cn` are subject to the ordering process, while `d1`, ..., `dm` are not.

As for the argument groups (`arg1a,arg1b, ...`), (`arg2a,arg2b, ...`), ..., (`argka,argkb, ...`) note the following:

- An argument group may be empty. That is denoted by two successive `*`'s separated by a comma.

- The number of argument groups may be less than the number of `c`-characters (i.e., `k` less than `n`). In that case argument groups `k+1`, `k+2`, ... are considered to be empty.
- The number of argument groups may be larger than the number of `c`-characters (i.e., `k` larger than `n`). Access groups are not considered in the actions described below and are simply reproduced.

When it encounters such a function, Schoonschip inspects the characters and permutes them according to the standard character ordering (`A-Z`, `a-z`, `0-9`; characters followed by an underscore are ordered just before the corresponding character without underscore: `A`,`B`,`C_`,`C` etc.). The sets of arguments

```
arg1a,arg1b, ...,arg2a,arg2b, ...
```

are permuted in tune with this. Thus if, for example, only `c1` and `c2` are exchanged, the argument list will read:

```
*,arg2a,arg2b, ...,*,arg1a,arg1b, ...,*,arg3a, ...
```

No permutation is done with respect to the characters `d1`, `d2`, ...

No permutation is done if the character variable `"S_` (*no symmetrization*) is given in the `Compo` argument list.

Next, all characters are glued together, forming a name. The name is searched for in the lists mentioned in <*options*>; if found, the name is identified with the corresponding variable. If not found, what happens depends again on the option specification. If a list is mentioned twice, then the new name is added to that list and new occurrences will be identified with the new name. If no list is specified twice in <*options*>, then the whole term is discarded. `Compo` *options*

In the option list, enclosed between `< >`, the characters `X`, `A`, `F`, `V` and `I` may appear once, and the characters `A`, `F`, `V` and `I` may also appear twice. Thus no new entries can be made to the list of `X` expressions, but the `X` table (which also includes `D` and `T` expressions and files) can be searched. At most one character can occur twice. There is a default option, `<AXIVFA>`, which means: search all lists, and if not found insert in the algebraic symbol list.

After the name is identified, the corresponding symbol is forced to be the first argument of the function, replacing all `c`- and `d`-characters and the optional `/`. Then all `*`'s are removed. The result is a function which has the constructed symbol as its first argument, and the arguments that follow are whatever results from the permutations among the argument groups. In subsequent substitutions one may work this function out to whatever is required.

```
C Example 20.

F F1,F2
A a,b,c,d

Z xx =    F1("c,"b,"a,/,"e,*,a1,a2)
        + F1("c,"b,"a,/,"e,*,a1,a2,*,xy)
        + F1("c,"b,"a,/,"e,*,a1,a2,*,xy,*,a2)
        + F1("c,"b,"a,/,"e,*,a1,a2,*,xy,*,b2,*,xz)

        + F2("a,"b,/,"F,*,x,*,y,*,z1)
        + F2("b,"a,/,"F,*,x,*,y,*,z1)
        + F2("F,"c,"b,"a,*,*,z,*,y,*,x)
        + F2("F,"c,"b,"a,*,*,z,*,yp,*,x)

Id,Compo,<AVIFXA>,F1,<FF>,F2
Id,Always,F2(F1~,a~,b~,c~,d~,e~) = F1(a,b,c)
Id,F1~(a~,b~,z1) = F1(a,b)
*begin
```

Notes: In the first group, `F1`, various possibilities for the number of argument groups are demonstrated. The option specified for that in the `Compo` command is actually the default option and could have been omitted. In the second group, it is shown how various functions with arguments can be created, even with different numbers of arguments. For that purpose the fake argument `z1` is introduced in the first two cases. The function substitution for `F2` works no matter how many arguments are present, but one must take care that no undefined quantities appear on the right-hand side. That means the function `F2` should not occur with fewer than three arguments. Unfortunately, it is not possible to handle empty arguments here. The fake argument `z1` is removed in the last substitution.

function dummies

Also note the use of dummies in the substitutions. For function dummies, we use a function symbol defined in the `F` list (`F1` in this case). Disaster may result if this is not done. It is extremely difficult to analyze for all the possible cases that may occur when such an error is made, many of which will lead to cryptic, seemingly unrelated error messages.

circuit example

Example 20 continues with a second part that shows the use of `Compo` together with tables. Although somewhat contrived, it shows the main ideas. Suppose an electric circuit must be built up from four, 3-pole elements connected by diodes, in the configuration shown:

```
    C Circuit diagram:

                      j1
              i1   ---0-------0---  i3
                   j4 |       |j2
                      |       |
              i2   ---0-------0---  i4
                      j3
```

The currents `i1` and `i2` are flowing inwards, `i3` and `i4` outwards; `j1`–`j4` are taken to flow clockwise. The 3-pole elements have three external connections called `A`, `B` and `C`. The connecting elements are diodes, and the connections are called `B` and `C`. The `B` of a diode may be connected to the `B` from a 3-pole, `C` to the `C` of a 3-pole. One might think here of the emitter and collector of a transistor. The main object is to generate all possible configurations, which is very simple in this case—one may either have all the diodes oriented clockwise, or all oriented counterclockwise. One also wants to exhibit the currents flowing through the various objects. Once the expression involving all elements with the currents is obtained explicitly, one can go ahead and work out whatever else is wanted. If the connecting elements are for instance a diode, resistor and battery in series, one can compute all currents as functions of the input/output currents `i1`–`i4`; but we will leave that to the imagination of the reader.

In the example below the function `v3` represents the 3-pole. The only combination to be recognized is the combination `ABC`. Thus only the function `ABC` is defined, and among the `Compo` options the `F` list is mentioned only once. Things like `AAB` or `ACC` will then be thrown away. Below we simply sum over all possibilities for the connecting lines (i.e., `AA`, `BC` and `CB`), leaving it to `Compo` to keep only those combinations that contain exclusively the functions `ABC` and `BC`.

A new feature in this example is the occurrence of characters in the lists for the tables `Ch` and `Cg`. While not exhibited here, there is another facility built-in. Normally one refers to a table element by means of the table name followed by a number. One may also refer to table elements by means of the table name followed by a character. Then the number associated with that character (given above) is used to pick out the correct element. This is useful if certain properties have to be associated with a character.

```
    T Ch(n)="A,"B,"C
    T Cg(n)="A,"C,"B

    F BC,ABC
    A i1,i2,i3,i4,j1,j2,j3,j4

    Z solu = square("A,"A,"A,"A)
```

```
Id,square(c1~,c2~,c3~,c4~) = DS{L1,1,3,(DS{L2,1,3,
        (DS{L3,1,3,(DS{L4,1,3,(
                v3(c1,Ch(L1),Cg(L4),*,i1,*,-j1,*,j4)*
                con(Ch(L1),Cg(L1),*,j1,*,-j1)*
                v3(c3,Cg(L1),Ch(L2),*,-i3,*,j1,*,-j2)*
                con(Ch(L2),Cg(L2),*,j2,*,-j2)*
                v3(c4,Cg(L2),Ch(L3),*,-i4,*,j2,*,-j3)*
                con(Ch(L3),Cg(L3),*,j3,*,-j3)*
                v3(c2,Cg(L3),Ch(L4),*,i2,*,j3,*,-j4)*
                con(Ch(L4),Cg(L4),*,j4,*,-j4)
                ) } ) } ) } ) }

Id,Compo,<F>,v3,con
Id,v3(f1~,a1~,a2~,a3~) = f1(a1,a2,a3)
Al,con(f1~,a1~,a2~) = f1(a1)

*end
```

<u>`Id,Count,xxx,arg1,`$\#_1$`,arg2,`$\#_2$`, ...`</u>

The command `Count` has grown out of the need to have a facility that selects terms on the basis of certain asymptotic properties. One may know how various functions and other quantities behave as a function of some variable or for certain limiting values of a variable, for example, the behavior as a function of x for very large x. With `Count` one can, for instance, select terms that behave at large x like x^n, with n larger than some given number. That is the most typical application. But the command allows many other possibilities.

selecting powers

In case `xxx` is a function or `X` expression, the format is:

```
Id,Count,Fx,arg1,#,arg2,#,...,# :arg3,#,
    ...,#:  arg4,#,...
```

The colons separate groups of arguments.

As an option, function names preceded by `"F` may be given directly after the first argument of `Count` or after the colons, if any:

```
Id,Count,Fx,"F,Fy,"F,Fz,arg1,#,arg2,#,
    ...,# :"F,Fc,arg3,#,...,#...
```

In the template above, `arg1`, `arg2`, etc., may be indices, vectors, algebraic symbols or functions. What happens when a term is first scanned is that a number called the *count* is constructed, which is a sum of counts resulting from individual factors. The count for an individual factor is the exponent of that factor (if applicable), multiplied by the number following the corresponding argument in the `Count` list. Thus if `x,7` occurs in the `Count` list, then `x^3` in a term contributes 21 to the count being constructed. If there is no exponent (as would be the case for a function or vector) the number from the list is added in once. The contributions of vector components or dot products are computed on the basis of the values assigned to the corresponding vectors in the list. Thus `p,2,q,5` leads to 21 for `pDq^3` and 10 for `q(3)^2`. Function arguments and vector indices are ignored in constructing the count. However, function arguments of functions specified with the `"F` option are counted as single occurrences. Thus, the statement

```
Id,Count,xxx,"F,Fa,a,35,b,1
```

leads to a count of 38 for the term `b^2*Fa(a,b,c)`.

What happens next depends on what occurs as the first argument in the `Count` list (i.e., `xxx` above):

- If `xxx` is a number, then the count is compared to `xxx`. If the count is greater than or equal to `xxx`, the term is kept, otherwise the term is deleted.
- If `xxx` is an algebraic symbol, vector component or dot product, then the count becomes the exponent of that quantity, and the combination is attached to the term.
- If `xxx` is a function or `X` expression, then the count becomes the argument of that function or `X` expression, and the combination is attached to the term.

Multiple counts can also be made. The occurrence of a colon (`:`) in the count list signals that subsequent counts must be put in subsequent function arguments. Thus, if two colons occur, a three-argument function is created. The list up to the first colon is used to compute the value of the first argument, the argument list between first and second colon determines the value of the second argument, etc.

As an example, consider the term

```
a^3*b^2*pDk
```

with vectors `p` and `k`. The command

```
Id,Count,Fc,a,1,b,2 : p,2,k,3
```

(`Fc` having been declared before as a function) generates a two-argument function `Fc`:

```
Fc(7,5)*a^3*b^2*pDk
```

The fact that Fc may also be an X or D expression allows for very compact but quite complicated counting. If Fc is a nonnumeric X or D expression, then several substitution levels are possibly needed to work out the result; and Schoonschip does not automatically take that into account. For purely numeric X and D expressions, only one level is needed. To ensure proper level spacing in case of a nonnumeric Fc, write a dummy substitution involving Fc directly before the Count command, and use Al for the Count command:

```
Id,XxxX=Fc
Al,Count,Fc,a,1,b,2 : p,2,k,3
```

where XxxX does not occur anywhere else.

Compo
Keep
*next

In the following example we not only demonstrate Count, but also show another application of Compo. The coefficients a1-a5 are constructed and entered into the A list (remember that Compo has the default options <AFVIXA>). The use of Keep and *next is shown as well.

```
C Example 21.

A x,y
T tt(n) = "1,"2,"3,"4,"5

C Here is a complicated way to make
C pow(x) = a1*x + a2*x^2 + a3*x^3 + a4*x^4 + a5*x^5.

Z pow(x) = DS(J,1,5,{f1(/,"a,tt(J))*x^J})

Id,Compo,f1
Id,f1(y~) = y

Keep pow
*next

Z xx = pow(x)

Id,Count,3,x,1

Keep pow
*next

Z xx = pow(y)

Id,Count,f1,y,2,a3,10

Keep pow
*next
```

```
V p,q
A AA

Z xx = pow(y)*f1(a2,a3)*pDq^2

Id,Count,AA,y,2,f1,-4,p,1,q,3

*end
```

Id,Cyclic,F1,2,3,4,F2,F3,4,5, ...

Id,Symme,F1,2,3,4,F2,F3,4,5, ...

The commands `Cyclic` and `Symme` are analogous to `Asymm`, except that invariance under the cyclic permutation or simple exchange of arguments is implied.

For the command `Symme`, alternative syntaxes can be used to denote groups of arguments:

```
Id,Symme,F1:2,3,4:7,8,9:12,13,14:
Id,Symme,F1:1-3:7-9:12-14:
```

In the case of `Cyclic`, the arguments are permuted following the pattern specified in the `Cyclic` list, which can make things pretty obscure, as the following example shows. The arguments are permuted to achieve the order "smallest first." As with `Asymm`, success in a conditional results when a permutation is done, while no permutation means failure.

```
C Example 22.

A a,b,c,d,e

Z xxx = f1(e,d,c,b,a) + f2(e,d,c,b,a) + f3(e,d,c,b,a)

Id,Cyclic,f1,2,5,4
Id,Symme,f2,f3,2,3,4

*end
```

Id,Dostop,on

Id,Dostop,off

Id,Dostop

IF Dostop

The command `Dostop` may be used to set, clear or test the do-loop forced exit flag. If the flag is set, then the next `ENDDO` encountered is taken to be satisfied. This command can be used to execute a set of substitutions and commands followed by a `*yep` repeatedly until some condition is satisfied.

Here is a typical example:

```
Z xx = a^10

DO J=1,100
Dostop off
Id,a1^n~=a1^(n-1)*b
IF NOT a1
..Id,Dostop,on
ENDIF
*yep
ENDDO

*end
```

The do-loop sequence will be re-executed 100 times, or until `a1` is no longer present. In the case shown, the cycle is run 10 times, and the result is `b^10`. Note that:

- the `Dostop` statement and command are within the same program segment (i.e., there is no line with a `*` in-between);
- the last line of the do-sequence contains a `*` in column 1.

Refer to the remarks on the `Dostop` statement in Chapter III, *Input Facilities*, for more details.

The `Dostop` command may also have `off`, or nothing as a parameter:

```
Id,Dostop,off
Id,Dostop
```

In the latter case the `Dostop` flag is tested, but not modified. The test mode may be used in conjunction with an `IF`:

```
IF Dostop
 ...
ENDIF
```

A set flag corresponds to "yes."

`Id,Epfred`

The command `Epfred` causes the reduction of products of `Epf` with equal numbers of arguments into `D`'s and at most a single `Epf`.

The simplest case is that of two arguments:

```
Epf(i1,i2)*Epf(i3,i4) =   D(i1,i3)*D(i2,i4)
                        - D(i1,i4)*D(i2,i3)
```

In three dimensions this is the equation that rewrites the product of two cross products in terms of dot products. If `p`, `q` and `k` are vectors, then the pseudoscalar $\texttt{k} \cdot \texttt{p} \times \texttt{q}$ is given by `Epf(k,p,q)`. If `pp`, `qp` and `kp` are also vectors, then the product $(\texttt{k} \cdot \texttt{p} \times \texttt{q})(\texttt{kp} \cdot \texttt{pp} \times \texttt{qp})$ can be rewritten in terms of dot products. Similar relations hold in higher dimensions.

Note: The product of two `Epf`'s of different dimension is not reduced.

```
C Example 23.

V p,q,k,pp,qp,kp

Z xxx =   Epf(k,p,q)*Epf(kp,pp,qp)
        + Epf(i1,i2,i3,i4)*Epf(i1,i2,j3,j4)

Id,Epfred

*end
```

`Id,Even,F1,2,3,4,F2,F3,4,5, ...`

`Id,Odd,F1,2,3,4,F2,F3,4,5, ...`

The commands `Even` and `Odd` specify that certain functions are even or odd with respect to a change of sign of certain arguments. Minus signs in front of such arguments are removed.

```
C Example 24.

F f1,f2,f3,f4

Z xxx =   f1(-a1,-a2,a3,-a4) + f2(-a1,-a2,a3,-a4)
        + f3(-a1,-a2,a3,-a4) + f4(-a1,-a2,a3,-a4)

Id,Even,f1,2,3,f2
Id,Odd,f3,2,3,f4

*end
```

`Id,Expand,fname`

`Freeze`

The command `Expand` occurs in combination with frozen files. See Chapter XIX, *Handling Large Problems*. In the syntax template above, `fname` is the name of a frozen file. All frozen subfiles referred to through the function `DF` are expanded.

`Id,Extern,arg1,arg2, ...`

The command `Extern` allows one to enter external routines as an integral part of Schoonschip. First one needs the statement `External`:

External

```
External filename
```

which loads the external file `filename` into Schoonschip's work space. At execution time, the `Extern` command causes a jump to the first location loaded from the file, whereupon manipulations can be performed on expressions, if the file contains position-independent, executable code. That requires an understanding of the internal formats used by Schoonschip, and presupposes the ability to generate such a file.

The Schoonschip interface provides access to all arrays, and also to internal utilities such as print routines, etc. That information is not contained in this manual, but is available elsewhere. Roughly speaking, a whole series of addresses is passed as a structure on the stack. The arguments are passed in the form of an array.

In principle this facility allows the creation of special packages for specific problems.

`Id,Gammas,`*loop indices plus options*`,Trace,`*loop indices plus options*

This command will be discussed in Chapter XVII, *Gamma Algebra*.

`Id,Iall,p,F`

`Id,Iall,F`

These commands are the inverse of `All` for the two possible syntaxes. Do not use `Iall` if the indices that occur as arguments have dimensional restrictions (`Dim_N` or `Dim_4`), such as can arise in the process of doing gamma algebra with the `Gammas` command. The command `Ndotpr` is to be used in such cases.

`Id,Ndotpr,F1`

The `Ndotpr` command is closely related to the `Gammas` command and is discussed in Chapter XVII, *Gamma Algebra*.

`Id,Numer,arg1,`$\#_1$`,arg2,`$\#_2$`, ...`

The command `Numer` can be used to insert numerical values for certain quantities. The numbers in the command template above need not be short numbers, they can be anything. Function arguments are not considered. The arguments may be algebraic symbols, vector components or dot products. In a conditional, success results if any of the quantities in the list is present, otherwise the result is failure.

```
C Example 25.

V p,q

Z xxx = a1^2*pDq^3/p(4)

Id,Numer,a1,2,pDq,1.E10,p(4),1.E5

*end
```

Id,Order,F1,F2, ...

Id,Order,"C,F1,F2, ...

Id,Order,#,F1,F2, ...

Id,Order,#,"C,F1,F2, ...

In the first template above, the `Order` command searches for the functions `F1`, `F2`, ... and builds chains of factors, one chain for each function in the list, on the basis of the first two arguments of the functions, which act as indices that link elements of a chain. First the function `F1` is sought. Its second argument is then looked for as the first argument in other `F1` factors. If found, the new `F1` factor is chained to the previous, etc. The same is done with `F2`, etc.

Only one index is kept in the factors in the collected `F1` chain, namely, the first one found; and similarly for `F2`, etc.

This index scheme is used in a new (to this version of Schoonschip) gamma matrix notation for keeping track of loops, described in Chapter XV and Chapter XVII, *Particle Physics* and *Gamma Algebra*.

`Order` *options*

The other templates correspond to additional options for the `Order` command. The second template, with the character `"C` (for collect) as the first command argument, causes the collection of all of the arguments of the factors in a chain, except for the first two index arguments, into the arguments of a single function for the chain.

The third and fourth templates, with the number `#` as the first command argument, cause `#` to be inserted as the first function argument (in case it is nonzero) for each of the collected factors of the `F1` chain, `#`+1 to be inserted for the `F2` chain, etc. The option `"C` is combined with this in the fourth template, which collapses each chain into a single function with many arguments, but each with a single index as first argument, `#`, `#`+1, etc. If `#` = 0, the first argument is omitted; no index is kept, whether `"C` is present or not.

Here are some single chain examples, including the output:

```
Z x1 = Faa(4,5,ddd)*Faa(2,3,bbb)*Faa(1,2,aaa)*Faa(3,4,ccc)
Id,Order,Faa

x1 = + Faa(1,aaa)*Faa(1,bbb)*Faa(1,ccc)*Faa(1,ddd) + 0.


Z x1 = Faa(4,5,ddd)*Faa(2,3,bbb)*Faa(1,2,aaa)*Faa(3,4,ccc)
Id,Order,"C,Faa

x1 = + Faa(1,aaa,bbb,ccc,ddd) + 0.


Z x1 = Faa(4,5,ddd)*Faa(2,3,bbb)*Faa(1,2,aaa)*Faa(3,4,ccc)
Id,Order,90,Faa

x1 = + Faa(90,aaa)*Faa(90,bbb)*Faa(90,ccc)*Faa(90,ddd) + 0.


Z x1 = Faa(4,5,ddd)*Faa(2,3,bbb)*Faa(1,2,aaa)*Faa(3,4,ccc)
Id,Order,90,"C,Faa

x1 = + Faa(90,aaa,bbb,ccc,ddd) + 0.


Z x1 = Faa(4,5,ddd)*Faa(2,3,bbb)*Faa(1,2,aaa)*Faa(3,4,ccc)
Id,Order,0,Faa

x1 = + Faa(aaa)*Faa(bbb)*Faa(ccc)*Faa(ddd) + 0.

Z x1 = Faa(4,5,ddd)*Faa(2,3,bbb)*Faa(1,2,aaa)*Faa(3,4,ccc)
Id,Order,0,"C,Faa

x1 = + Faa(aaa,bbb,ccc,ddd) + 0.
```

`Id,Print,#,`*message*

This command is used to print a message to the standard output at execution time. Here `#` limits the number of times the message is printed. Everything after the third comma, including blanks through the end of the line, is printed when this command is encountered at execution time. For example:

```
Id,2,Print,  Hello world.
```

The message "`  Hello world.`" is printed at most twice.

<u>`Id,Ratio,a1,a2,a3`</u>

The command `Ratio` is designed specifically to facilitate integrations. It is used to rationalize a product of factors that are linear in the integration variable. The simplest case is:

```
1/(x+a) * 1/(x+b) = {1/(x+a) - 1/(x+b)}/(b-a)
```

Since expressions in Schoonschip cannot have negative exponents (except when they are purely numerical), one cannot write such things directly. The way to handle that is to introduce symbols for expressions that appear with negative exponents, for example, `xpa = x+a`. Then the equation becomes: *negative exponents*

```
xpa^-1*xpb^-1 = {xpa^-1 - xpb^-1}*bma^-1
```

Here the `[ ]` notation for symbols is very convenient. One may write: *square brackets*

```
[x+a]^-1 * [x+b]^-1 = {[x+a]^-1 - [x+b]^-1}/[b-a]
```

If the exponents are large this becomes quite complicated, and that is where `Ratio` comes in. In the command one must specify the two factors and their difference; in the case mentioned above, one would write:

```
Id,Ratio,[x+a],[x+b],[b-a]
```

`Ratio` works for all exponents that can occur, whether positive or negative.

```
C Example 26.

A a1,a2,a2ma1
F f1
B b2,b3,b4,b5

Z xx=f1(8,4)

Id,f1(n~,m~)=
        { b2*a1^n*a2^m
        + b3*a1^-n*a2^m
        + b4*a1^n*a2^-m
        + b5*a1^-n*a2^-m
        }

Id,Ratio,a1,a2,a2ma1

*begin

C Here the use of Ratio with [] names.

B [b-a]
```

```
Z xxx = 1/[x+a]^3 * 1/[x+b]^2

Id,Ratio,[x+a],[x+b],[b-a]
P output

*yep

C Just checking...

Id,[x+a]^n~ = (x+a)^(3+n)*(x+b)^2/[x+a]^3/[x+b]^2
Al,[x+b]^n~ = (x+a)^3*(x+b)^(2+n)/[x+a]^3/[x+b]^2
Id,b = [b-a] + a

*end
```

short difference While the first two arguments following Ratio must be symbolic, the third can also be a short number, -128 to 127 (not an expression).

Id,Spin,*loop indices or momenta*

This command is discussed in Chapter XV and Chapter XVII, *Particle Physics* and *Gamma Algebra*.

Id,Stats,#

The Stats command provides statistics. The counts are printed at the conclusion of a section delimited by a * line. Four counts may be kept, specified by the number #, which may be absent, implying 0, or one of the numbers 1, 2 or 3. No error message is given if the number exceeds 3 or is negative; Schoonschip takes the number modulo 4. Every time this command executes, 1 is added to the appropiate count. Example:

```
IF a^3
Id,Stats,2
ENDIF
```

This construction counts the number of terms with a^3. The result is printed at termination.

Chapter XIII

Conditional Input

Conditional input statements may be used to formulate small variations on a given problem. The idea is to define parameters near the beginning of a problem, and to read the input program text as a function of those parameters. Many variations on a given problem can then be run by simply changing a few parameters in the beginning.

conditional parameters

command line

The parameters are called `_Sw0`, ..., `_Sw9`, and their default values are zero. The parameter `_Sw0` is special because its initial value can actually be set on the command line when Schoonschip is invoked. The command line notation for that is `S=#`, where `#` must be a number in the byte range -128 to 127. For example:

```
    Schip S=7 Ifile
```

The initial value of `_Sw0` will then be 7.

`Set`

The `_Sw`*x*'s may be changed by means of a `Set` statement. The syntax is the following:

`Set _Sw`*x* `=` *expr*

with *x* a number `0`, ...,`9`, and *expr* an essentially numerical expression with value in the byte range (-128, 127). For example:

```
    Set _Sw3 = 7
```

Every occurrence of `_Sw3` is then replaced by the number 7. That includes occurrences within an expression; for example,

```
    ... a^_Sw3 ...
```

is read as `a^7`. Another example:

```
    Set _Sw5 = _Sw5 + 1
```

Gotoif
Gotoifn

The variables _Sw0, ..., _Sw9 may also occur in a Gotoif (or Gotoifn) statement. Here is the syntax:

Gotoif *expr*, *expr*$_1$ *?* *expr*$_2$

The expressions *expr*, *expr*$_1$ and *expr*$_2$ must be simple numerical expressions which evaluate to numbers. *Expr* must evaluate to a number in the range 0 to 99. The other two expressions must evalute to 16-bit numbers in the range -32768 to 32767. The question mark *?* may be >, < or =. The expressions are compared and if the condition is true all following lines are skipped until a line with an @ symbol in column 1 occurs. The @ symbol may be followed by a number, and if this number is equal to the value of *expr* the skipping stops, else it continues. If there is no number after the @ symbol, the skipping stops unconditionally.

The statement

Gotoifn *expr*, *expr*$_1$ *?* *expr*$_2$

is analagous to the above, except that skipping occurs when the condition is false.

Goto

Finally,

Goto *expr*

causes an unconditional skip.

expressions
logical operators

The expressions may contain numbers separated by the operators +, -, |, &, * and /. The operator | stands for the logical OR and & is the logical AND. The precedence for the arithmetic operators +, etc., is as usual, and | has the same precedence as +, & the same as *. Division is integer division; for example, 3/2 equals 1. Furthermore the variables _Sw0, ..., _Sw9 are considered to be numbers with the value to which they have been set (initially zero).

Synonym

Here is a simple example, which also shows the use of the Synonym statement:

```
Synonym Xxx = _Sw0

Set _Sw9 = 3
Set _Sw1 = 2-3*(22/2-4)
Set _Sw2 = 2-(22&2-4)*_Sw9
Set _Sw3 = 6
Set _Sw4 = 1 | 0x20

Z xx = (a + b)^3 + a0^'Xxx' + a1^_Sw1 + a2^_Sw2 + a3^_Sw3
       + a4*_Sw4
```

```
Gotoif 20, _Sw3 > 5

Id, b = 20*c
Goto 21

@20
Id,b = a + c

@21
*begin
```

Here is the output in case the value `62` was specified as the initial value for `_Sw0` from the command line, `Schip S=62 ...` If no `S=#` parameter had been present on the command line, the value of `_Sw0` would have been `0`, giving `1` instead of `a0^62`.

```
xx = a0^62 + a1^-19 + a2^8 + a3^6 + 3*a4 + 6*a*c^2
     + 12*a^2*c + 8*a^3 + c^3
```

Note the notation for a hexadecimal number, `0x20` (`= 32` decimal). An OR with `1` produces `0x21`, i.e., `33` decimal. The AND of `2` with `22` is `2` (in hexadecimal `22 = 0x16`, which gives `0x02` when AND'ed with `0x02`).

In the above case, since the condition is true, skipping starts immediately after the `Gotoif` statement, and continues until the line with `@20` is met. Thus the problem reads:

```
Z xx = (a + b)^3 + ...

Id,b = a + c
*begin
```

If `_Sw3` is not set, or is set to some value equal to or less than `5`, the problem will be read as

```
Z xx = (a + b)^3 + ...

Id, b = 20*c
*begin
```

The use of

```
Gotoifn 20, _Sw3 < 6
```

instead of the earlier `Gotoif` statement would give the same results.

Lines with an `@` in column 1 are ignored when there is no skipping.

Chapter XIV

Internal Procedures

Some insight into the inner workings of Schoonschip is indispensible, and it is the aim of this chapter to provide that. Although not reproduced here, Example 28, continued in Example 29, also illustrates some of the essentials by showing how to attack a problem that could easily explode if not approached carefully.

1. Levels

Consider the expression:

```
     Z XX = (a*x^2 + b*x)*dx
```

The process begins with the reading of the input, which is encoded and placed into memory. Names are placed in arrays. The encoding stops when a `*` line is reached. At this point, Schoonschip starts working out the `Z` expressions, one after the other. This happens at Level 0. At Level 1, one term after another passes by. In the case above, the first term to pass by is `a*x^2*dx`. No new term is emitted until that term has passed 40 levels, and at each level the user has an opportunity to work on it. When the term is finally collected in the output store, the next term is issued, `b*x*dx` in our case. The output store is a large area of more than 40 kilobytes; provisions for handling its overflow are discussed in Chapter XIX, *Handling Large Problems*, and Appendix A, *Command Line Options*.

levels

output store

The advantage of this procedure is that virtually no storage is needed for intermediate results. Symbolic programs often work out an entire expression before going on to the next level. One then finds the expression, transformed at the first level, inside the initial expression, and so on. The same expression in various stages of processing occurs many times in memory. That leads very quickly to unmanageable memory overflow.

collecting terms

On the other hand, a substantial reduction in the number of terms may take place when they are collected in the output store. To take a trivial example, the initial expression `(a+b)^2` leads to 4 terms; and each of these 4 terms goes through all 40 levels (actually Schoonschip notes how many levels are active, and skips the empty levels). If one were to collect these terms immediately after the first level there would then be only 3 terms: `a^2 + 2*a*b + b^2`. That is what happens at the output store: like terms are recognized and the coefficients are added, rather than keeping the terms separately (here that applies to `a*b`). Thus collecting terms leads in general to a reduction in their number, which in fact is often very substantial.

`*yep`

This was of course realized early on; and an option was included to collect terms at any time, in the midst of substitutions and commands. That is `*yep`. When Schoonschip finds a `*yep` line, it processes all substitutions up to that point, and then collects all terms in the output store. Next, the output store is emptied to the disk; and then it is read in again, term by term. Each term again passes through all the levels, possibly to another `*yep`, until all of the substitutions in the next set are exhausted. It is up to the user to place `*yep` lines judiciously when faced with a large problem.

As may be clear from the above, this procedure has the added advantage that after a `*yep` the full 40 levels can be used again. There is thus basically no limit on the number of substitutions and commands that can be done.

level options
`Id`
`Al`

The assignment of substitutions and commands to levels is essentially automatic, but the user has a number of options. A substitution with `Id` is assigned to the next level; with `Al`, it is put on the same level, except for a special situation described further below. Thus substitutions that do not interfere with each other can be put on the same level: the first with `Id`, the rest with `Al`. One may also want to have a substitution done at a number of consecutive levels. In such a case one simply places a number after the `Id` or `Al`. Thus, `Id,7,` ... causes the corresponding substitution to be attempted on 7 consecutive levels.

A special situation occurs when the command following `Id` uses more than one level. When such a command is followed by `Al`, the `Al` substitution acts at the last level used by the `Id` command, instead of at the first level it uses.

level printout

Upon reading the input, Schoonschip prints the corresponding level at the beginning of the line. To understand the numbering, one must realize that working out brackets requires levels, one per level of nesting. Thus in the last example, the first substitution is put at Level 2, because Level 1 is used to work out the brackets. In fact, working out a bracket is very much like a substitution; the initial expression is translated to something like `$AAA^2`, where `$AAA` is a symbol invented for that purpose; and the substitution

```
    Id,$AAA = a+b
```

is placed at Level 1. For that reason the `$` symbol must not be used in names. This process may be seen in detail by using a `P brackets` statement to print the `$` expressions. `X` expressions may also require levels; they are like built-in substitutions. As soon as an `X` expression materializes at a level, Schoonschip substitutes its definition.

`$`
`P brackets`
`X` *expression levels*

It may help to make the whole process clearer if we reproduce the contents of memory for a simple case, somewhere in the middle of a calculation. Consider:

```
    Z xxx = (a*x + b)*dx

    Id,x*dx = (x2^2 - x1^2)/2

    *end
```

At a certain point, the following may be found in memory:

levels example

Level 0:	`$AAA*dx`		
Level 1:	`dx`	and	`a*x + b`
Level 2:	`dx*a*x`		
Level 3:	`a`	and	`$AAB/2`
Level 4:	`0.5*a`	and	`x2^2 - x1^2`
Level 5:	`-0.5*a*x1^2`		
Levels 6-40:	skipped		
In output store:	`0.5*a*x2^2`		

The action at this point is at Level 5. A little later the term `-0.5*a*x1^2` will arrive at the output store, and that exhausts the work at Level 4. Schoonschip then goes back to Level 1 and produces the next term, `b*dx`. After that has passed to the output store (for simplicity we have not included the substitution to integrate this term) Schoonschip goes all the way back to Level 0, finds that the problem is done, and goes on to the next stage, namely, the printing of the contents of the output store.

2. Problem Size

This rather complicated procedure has the great advantage that very little memory is used at any one time. One finds bits and pieces from Level 0 all the way down to the output store, and Schoonschip keeps climbing up and down all the levels until all the work is exhausted at every level. To deal with large problems, one must have some understanding of this procedure in order to organize the calculation in the most efficient way. Any substitution with more than one term on the right-hand side leads to a multiplication of the number of terms generated, by some factor which depends on how many terms occur on the right-hand side, and also on how often the substitution is actually done.

`P stats`

multiplications and terms

To see how much work Schoonschip does, use the statement `P stats`. Then some statistics are printed, of which the most interesting in this context are the number of multiplications and the number of terms. One multiplication is counted for every 2 terms (not factors) that are multiplied together. Thus `(a*x + b)*(c*x + d)` yields 4 multiplications. Often the count comes out higher than one might expect, due for instance to the way brackets are worked out. The number of terms is simply a count of the terms that arrive at the output store. That would be 4 in this case.

Another example: `(a+b)^4` leads to $2^4 = 16$ multiplications, and then to 16 terms; and after sorting in the output store one will have 5 terms. In actual fact, the count for multiplications comes to 81 in this case, but we shall not bore the reader with the details. Mainly, that is because Schoonschip begins by setting a 1 before getting to the actual expression.

large problems

`*yep`

In practice the number of terms or of multiplications is a good indicator of the complexity of a problem, and the time needed is roughly proportional to those numbers. For large expressions with many terms, most of the time is spent sorting the terms in the output store. Try to avoid more than 100,000 terms. If one gets into this kind of situation, it becomes necessary to examine the organization of the problem in detail, and its separation into parts by means of `*yep`. Chapter XIX and Chapter XX, *Handling Large Problems* and *File Handling Commands*, should also be consulted.

times for multiplications and terms

`P stats`

The largest calculation known to the author involved as many as 1,000,000 terms, but it is not known how much disk space was used. The present version of Schoonschip needs about 20 seconds for 10,000 terms or 100,000 multiplications (8 MHz M68000), but this must be seen as a very rough, low estimate. Times may increase drastically where writing to disk is involved. The calculation just mentioned required 20 minutes on a CDC 7600, counting cpu time only, not actual time. The same calculation would probably take many hours on an M68000, assuming the availability of disk space. It is an important point that there is at least no limitation due to Schoonschip itself, that given enough time and disk space it will work its way through. The `P stats` statement prints statistics for every 8,192 ($= 2^{13}$) terms that pass through, so that one can see some action.

power series

In particular, a series expansion where the argument of the expansion is another series can very easily lead to an enormous number of terms. Suppose some function $f(y)$ is given as a series expansion in y, and suppose that y is also a series expansion in the variable x. Now suppose that the series expansion of f in powers of x through x^{10} is to be found. If y is a polynomial including powers up to x^9, thus 10 terms, substitution into y^{10} yields 10^{10} terms. That would take roughly 20×10^6 seconds $\approx 5,600$ hours, if nothing else were to grow out of bounds and stop the calculation. Such problems really must be worked out carefully. In Examples 28 and 29 we have reproduced a substantial part of such a problem, enough to explain the principle involved.

Chapter XV

Particle Physics

Schoonschip has a few built-in facilities that are particularly useful for particle physics. We refer to gamma matrix algebra and traces of products of gamma matrices.

built-in functions
gamma matrices
spinors

The special functions `G`, `Gi`, `G5`, `G6`, `G7`, `Ug` and `Ubg` are reserved to denote gamma matrices and spinors. Since more than one loop of gamma matrices can occur in a given calculation, all of these functions have a loop index. Gamma's with different loop indices are supposed to commute. The correspondence between the old Schoonschip notation and the standard physics notation for these quantities is the following:

old notation

`G(L,mu)`	gamma matrix with index `mu` of loop `L`
`Gi(L)`	the identity matrix of loop `L`
`G5(L)`	`= G(L,1)*G(L,2)*G(L,3)*G(L,4)`
`G6(L)`	`= Gi(L) + G5(L)`
`G7(L)`	`= Gi(L) - G5(L)`
`Ug(L,m,p)`	spinor loop `L`, particle mass `m`, momentum `p`, spin 1/2
`Ubg(L,m,p)`	`Conjg{Ug(L,m,p)}*G(L,4)`
`Ug(L,mu,m,p)`	spin 3/2 spinor
`Ubg(L,mu,m,p)`	`Conjg{Ug(L,mu,m,p)}*G(L,4)`

A new notation for gamma matrices has been introduced in the present version (January 1, 1989) of Schoonschip. The above notation is still accepted, but the new notation is more flexible, and we use it in this manual.

new notation

`Gammas`

In the new notation, gammas have two loop indices. They may also have more than one vector index. The order of these two-index gammas is immaterial; at some point, when tracing or spin summation or whatever is done, (usually with the command `Gammas`), the gammas are chained on the basis of the two loop indices. Example:

```
Ug(i3,M,q)*G(i1,i2,mu)*G6(i0,i1)*Ubg(i0,m,p)
   *G(i2,i3,nu,al)
```

`G`-*strings*

This will be chained into one gamma string (called a `G`-*string* in the following):

```
Ubg(i0,m,p)*G(i0,"s,"6,mu,nu,al)*Ug(i0,M,q)
```

equivalent to the old sequence

```
Ubg(i0,m,p)*G6(i0)*G(i0,mu)*G(i0,nu)*G(i0,al)
   *Ug(i0,M,q)
```

This takes the pain out of the process of ordering the gammas. Within `G`-strings, `"4`, `"5`, `"6`, and `"7` stand for `Gi`, `G5`, `G6` and `G7`.

antiparticle spinor

An antiparticle spinor may be represented as a particle spinor with `-m` instead of `m`.

anticommutation rule

The `G`'s satisfy the rule:

```
G(L,"s,"4, ...,mu,nu, ...) = - G(L,"s,"4, ...,nu,mu,...)
   + 2*D(mu,nu)*G(L,"s,"4, ..., ...)
```

which is the usual anticommutation rule.

gamma conjugation

Schoonschip treats all `G`'s as imaginary, corresponding to a hermitean set. Under complex conjugation, the order of the `G`'s is reversed (i.e., the order inside a `G`-string is reversed). `G5` is imaginary, `G6` and `G7` are complex conjugates of each other.

`Gammas`
`Spin`
n-dimensional gammas

The commands `Gammas` and `Spin` allow for the reduction of a product of `G`'s to at most two `G`'s, the taking of traces, and spin summation. The built-in equations can handle 4-dimensional and n-dimensional situations, with or without `G5`'s.

The command

```
Id,Spin,L1,p, ...
```

`Spin`

does spin summation. Schoonschip looks for pairs of spinors `Ug` and `Ubg` with the same arguments except for the loop index (one of the loop indices must be mentioned in the list), and replaces such pairs by the appropriate combination. A vector rather than a loop index may also serve as identification. The command

```
Id,Spin,p
```

or

```
Id,Spin,L1
```

replaces the expression

```
Ug(L1,m,p)*Ubg(K1,m,p)
```

by

```
- i*G(L1,K1,p) + m*Gi(L1,K1)
```

Similarly for spin 3/2 spinors: *spin 3/2*

```
Ug(L1,mu,m,p)*Ubg(K1,nu,m,p)
```

is replaced by

```
{ D(mu,nu) - 1/3*i*G(L1,K1,mu)*p(nu)/m

     + 1/3*i*G(L1,K1,nu)*p(mu)/m - 1/3*G(L1,K1,mu,nu)

     + 2/3*Gi(L1,K1)*p(mu)*p(nu)/m^2 }

        * { - i*G(L1,K1,p) + m*Gi(L1,K1) }
```

Chapter XVII, *Gamma Algebra*, explains how to use the `Gammas` command to manipulate `G`-strings.

Example 27 gives the complete calculation for muon decay in the *V-A* theory. It is not reproduced here. *muon decay*

The `Compo` command is very useful for calculating diagrams. The possible vertices can be given in terms of `X` expressions. For a given topology one sums over all possible propagators. Separate examples exist for such cases, and are available elsewhere. `Compo`

Chapter XVI

Character Summation

Characters have become quite useful in Schoonschip for generating diagrams. Typically one character is associated with each particle in the theory. The same character appended with an underscore (`_`) corresponds to the antiparticle. There is a way to let Schoonschip know when a particle is its own antiparticle.

particles and antiparticles

1. Vertices and Propagators

To generate diagrams one first needs a list of vertices. The name for a vertex is composed of the characters that correspond to the particles entering the vertex. The characters in the vertex name must be ordered in a well-defined way: alphabetically, with upper case before lower case before numbers, and with characters having an underscore before those without an underscore. Thus, `A_AB_BC_C`...`Z_Za_ab_b`...`1_12_2`... Here are some examples:

vertices

vertex names

```
    AWW   Aww   U_U   AG_G   Ue_e   Ubt_  U_b_t   U
```

Vertices have indices and momenta as parameters. Four-particle vertices have no momentum dependence.

The list of vertices must be given in terms of `X` expressions. Each vertex must have sufficient parameters to cover all cases. For a given particle, one may have momentum, a vector index and a spinor index. Even if the vector and spinor indices do not normally occur simultaneously, both need to be carried along. A 3-vertex will thus have 9 parameters.

vertex parameters

Here are some examples, the first of which is a vertex for three vector particles:

```
X U_UW(AL,al,P,BE,be,Q,GA,ga,K)
   = C*D(AL,BE)*(Q(GA)-P(GA)) + C*D(AL,GA)*(P(BE)-K(BE))
   + C*D(BE,GA)*(K(AL)-Q(AL))
```

The spinor indices `al`, `be` and `ga` are not used.

Here are two spinor vertices:

```
X Ubt_(AL,al,P,BE,be,Q,GA,ga,k)
   = i*G(ga,be,AL) + avc*i*G(ga,be,AL,G5)

X Ue1_(AL,al,P,BE,be,Q,GA,ga,k) = i*G(ga,be,AL,G6)
```

Note that spinor indices usually appear in reverse order. This is the vertex for a vector particle (charged *W*) coupled to a fermion and an antifermion (bottom and top quarks or electron-neutrino).

propagators

Here is a fermion propagator:

```
X t_t(L1,l1,L2,l2,K) = NOM(K,Mt)*G(l2,l1,K)
   + NOM(K,Mt)*Gi(l2,l1)*Mt
```

`NOM` is the usual propagator factor, `1/[KDK + Mt^2]`.

All vertices and propagators must be defined in this way.

2. *W* Boson Self-Energy Diagram

We give here a rather technical example, a program which calculates a diagram for the *W* boson self-energy.

After the type declarations, the vertices and propagators are given in terms of `X` functions. Only one diagram is evaluated in this example, simply because no more vertices are listed. If a complete set of vertices were to be given, the complete result would obtain. The charged *W*'s are represented by `U` and `U_`.

```
C Calculation of charged vector boson (U,U_) self-energy.

F Fx,F2,B22,B21,B1,B0,VE3,PROP

I AL=N,al,BE=N,be,GA=N,ga,MU=N,mu,NU=N,nu,MUP=N,mup,
  NUP=N,nup,L1=N,l1,L2=N,l2,L3=N,l3,L4=N,l4,L5=N,l5,
  L6=N,l6,L7=N,l7,L8=N,l8, L9=N,l9,L0=N,l0

I Mu4,Nu4

V P,Q,K
```

The use of the following character tables, the `Anti` statement and the `G` line will be explained later:

```
T TAP: W,Z
T TFE: t,b

Anti,TAP

G 1

X t_t(L1,l1,L2,l2,K) = i*NOM(K,Mt)*G(l2,l1,K)
   + NOM(K,Mt)*Gi(l2,l1)*Mt
X b_b(L1,l1,L2,l2,K) = i*NOM(K,Mb)*G(l2,l1,K)
   + NOM(K,Mb)*Gi(l2,l1)*Mb

G 3

X Ubt_(AL,al,P,BE,be,Q,GA,ga,K) = i*G(ga,be,AL,G6)
X U_b_t(AL,al,P,BE,be,Q,GA,ga,K) = i*G(be,ga,AL,G6)
X Wt_t(AL,al,P,BE,be,Q,GA,ga,K)
   = i*[1-8/3s^2]*G(be,ga,AL) + i*G(be,ga,AL,G5)
X Wb_b(AL,al,P,BE,be,Q,GA,ga,K)
   = i*[4/3s^2-1]*G(be,ga,AL) - i*G(be,ga,AL,G5)

G
```

First establish the topology, for example, the simple two 3-vertex, two-propagator diagram. Note that `_` may *not* be used in file names:

```
Z IUUB=SELF("U,"U_)
```

In the lines that come next, we do a character summation. The given input character is `I1`, with `"U` in this case representing the positively charged *W*. The sum in `DS` goes over all characters `J1` and `J2` such that the name `UJ1J2` properly symmetrized corresponds to some `X` expression. In this case, for `J1`, `J2` that will only be some combinations of `b`, `b_`, `t`, `t_`. The argument `Sym` in `DS` implies a symmetrization factor based on the characters that follow. If they are identical (which will not happen here) a factor 1/2! will be supplied.

The numbers `2`, `3`, `5` and `6` following `J1`, `J2` imply that only `X` expressions in groups 2, 3, 5 and 6 will be inspected. That is the meaning of the `G` lines above: they specify group numbers for the `X` expressions that follow. The default is 0; and if no groups are specified, all are considered.

When a match is found, it gives rise to a term in the summation, the function `DIB` with the characters `I1`, `J1`, `J2` and `I2` as arguments. The function `DC` is used for various purposes; here it provides a factor `-1^1` if both characters

J1 and J2 appear in table TFE. If one uses "F_ rather than "F the factor is supplied provided none of the characters appears in the table TFE.

The second number in DC, the exponent 1, may be omitted, and is then taken to be 1. This is the factor −1 for a fermion loop. The "F in the DC function stands for *factor*.

Other uses for DC are summarized later. Here is the character summation:

```
Id,SELF(I1~,I2~)
   = DS(I1;J1;J2;Sym;J1;J2,2,3,5,6,(DIB(I1,J1,J2,I2)
        *DC("F,TFE,-1,1,J1,J2) ))
```

The diagram is essentially defined below. Mu4 and Nu4 are the external U and U_ indices, P is the loop momentum, Q is the U momentum and K = P+Q. The *'s are used by the Compo command. For example, in VE3, Compo will rearrange the characters I1, K1 and K2 into the character order defined before. Every interchange also gives rise to an interchange of the argument groups delimited by the *:

```
Id,DIB(I1~,K1~,K2~,I2~) =
   VE3(I1,K1,K2,*,Mu4,mu,Q,*,L1,l1,-K,*,L3,l3,P)*
   VE3(-K1,I2,-K2,*,L2,l2,K,*,Nu4,nu,-Q,*,L4,l4,-P)*
   PROP(K1,-K1,*,L1,l1,K,*,L2,l2,K)*
   PROP(K2,-K2,*,L3,l3,P,*,L4,l4,P)
```

Next, Compo searches the X expression list for names composed of the first few characters that appear as arguments of the functions VE3, etc., and if successful keeps the term, otherwise makes it zero:

```
Id,Compo,<X>,VE3,PROP
```

Since Compo has composed the name and performed the necessary symmetrization, the functions themselves can now be substituted. Here AA is a dummy name:

```
Id,VE3(AA~,MU~,mu~,P0~,L2~,l2~,Q~,L1~,l1~,P~)
   = AA(MU,mu,P0,L2,l2,Q,L1,l1,P)
Al,PROP(AA~,L4~,l4~,Q~,L3~,l3~,P~) = AA(L4,l4,L3,l3,Q)
```

At this point the full expression for the diagram has been obtained. The next step is to do the loop integral. The loop momentum appears in the propagators through the functions NOM, and as arguments in the gamma matrices:

```
Id,Commu,NOM
Id,NOM(P,M~)*NOM(K,M0~) = F2(M,M0)
Id,Gammas,"C
```

Now remove K as a G argument, replacing it by the four-dimensional Q and the loop momentum P. No index is created by this operation, and the dimensionality of MU is irrelevant:

```
Id,Funct,K(MU~) = P(MU)+Q(MU)
```

Next collect all P's in the function Fx:

```
Id,All,P,N,Fx
*yep
```

Do the P integration:

```
Id,F2(M~,M0~)*Fx(MU~,NU~) = D(MU,NU)*B22(M,M0)
   + Q(MU)*Q(NU)*B21(M,M0)
Al,F2(M~,M0~)*Fx(MU~) = Q(MU)*B1(M,M0)
Al,F2(M~,M0~) = B0(M,M0)
```

Do anomalous traces:

```
Id,Gammas,"A
*end
```

3. Character Tables

character table definition

Character tables are distinguished from other tables that can be used as function arguments by means of a colon (:) following the name on the T line. To every possible character there corresponds a location in the table, zero by default. Any character mentioned gives a 1, and a -1 in the location for its antiparticle (denoted by an appended underscore). Specific numbers may be assigned, for example:

```
T TAB: A=3:5,B=7,a,z=3
```

This assigns 3 to A, 5 to A_, 7 to B, -7 to B_, 1 to a, -1 to a_, 3 to z and -3 to z_.

Important: all numbers appearing in character tables must be short integers, $-128 \leq$ # ≤ 127.

DC *antiparticle table* *fix DS Compo

The DC function uses character tables. Other character manipulations often need a table to specify which particles are their own antiparticles. Except for DC calculations, where there is a mandatory antiparticle table argument, an antiparticle table may be given before *fix, which makes it the default, for example, for the character function DS and the command Compo. In all cases one may specify a particular antiparticle table for a specific case: for DS as an argument before the group numbers, for Compo anywhere in the list.

Anti *statement*

The Anti statement

```
Anti,TAA
```

establishes character table TAA as the default antiparticle table. It becomes the default for DS and the Compo command, in which case no antiparticle table needs to be mentioned explicitly for them. The statement may be given in the *fix section and is then valid by default for all other sections. The command

*fix

Anti *command*

```
Id,Anti,TXX
```

causes all function arguments to be scanned. The table TXX is used to determine which particles are their own antiparticles, and any corresponding appended underscores are explicitly removed. This command must be used even if the table TXX has been established as the default antiparticle table by the Anti statement, if one wants to explicitly remove underscores for such particles. DS and Compo treat the selfconjugate particles in the default table implicitly, without the explicit removal of underscores.

4. Character Summation Function DS

The general format is

DS(C1;C2;C3;...;[Sym;C4;C5;...;][TX,][$\#_1$,$\#_2$,...,](*expression*))

The [ ]'s indicate optional arguments, the C's are either characters or dummies for characters, and the #'s are group numbers. The characters C1, C2, etc. must be followed by semicolons (;). A leading - sign adds or removes an underscore. The summation goes over all names that fit the name of some X expression. For example, if C1 = "A and only two additional dummies C2 and C3 are mentioned, then DS searches for all X expressions with a name consisting of three characters, one of which is an A. Then C2 and C3 become the other two characters, and as such may be used in *expression*. Sym is tricky, and needs to be checked in simple cases to be sure the result is what one expects. Examples are shown at the end of this section.

The optional arguments in DS have the following effects:

- If the symbol Sym is mentioned, then a symmetrization factor is provided based on the characters following it.
- If a character table TX is mentioned, it is taken to be the antiparticle table, listing all characters X for which X = X_.

- If group numbers are specified, only `X` expressions in those groups are considered.

Example:

```
DS("A;J3;-J3;"B;Sym;J3,-J3,2,(Dic("A,J3,"B) ))
```

Every `X` expression name containing characters of the form `AXX_B` (or `AX_XB`) yields a term in the sum. For example, the following names fit the pattern twice:

`AEBE_`	(with `J3="E` and `J3="E_`)
`Z_ZAB`	(with `J3="Z` and `J3="Z_`)

The order of the characters in the name is of no importance here. The order is important, however, for the command `Compo`, which orders the characters before searching the name lists.

The next example, including slightly edited output, shows the effect in a simple case with and without `Sym`:

```
X gq_q(a,b,c)=vert(a,b,c)

Z xx=DS("g;J1;J2;( Dia("g,J1,J2) ))
Z xxs=DS("g;J1;J2;Sym;J1;J2;( Dia("g,J1,J2) ))

*end

xx = + Dia("g,"q,"q_) + Dia("g,"q_,"q)

xxs = + Dia("g,"q_,"q) + 0.
```

5. Character Function DC

The value of `DC` is always some numerical factor depending on the character arguments. In the following, literal or dummy character arguments are denoted by `C1`, `C2`, ...

<u>`DC("F,TX,`$\#_1$`,`$\#_2$`,C1,C2,...)`</u>

<u>`DC("F_,TX,`$\#_1$`,`$\#_2$`,C1,C2,...)`</u>

For the argument `"F`, the value of `DC` is $\#_1^{\#_2}$ (if absent, $\#_2$ is taken to be 1) provided *all* characters `C1`, `C2`, etc. appear in the table `TX`. Otherwise it is 1. For `"F_`, the value is $\#_1^{\#_2}$ provided *none* of the characters appears in the table. Otherwise it is 1.

Here is an example of a character table `TX`:

```
T TX: t,b
```

<u>`DC("P,TX,#,C1,C2,...)`</u>

Compute and multiply with a permutational factor determined by the occurrence of identical characters in `C1`, `C2`, etc. The number `#` is used as an exponent. For example, with `#` $= 2$ and the characters `"A`, `"A`, `"A`, `"B`, `"B`, the value $(3! \times 2!)^2$ obtains. The table `TX` lists particles that are their own antiparticles (`X_ = X`). Here is an example of such a character table:

```
T TAP: W,Z
```

<u>`DC("C,TX,C1,C2,...)`</u>

<u>`DC("C_,TX,C1,C2,...)`</u>

<u>`DC("T,TX,C1,C2,...)`</u>

<u>`DC("T_,TX,C1,C2,...)`</u>

Compute the sum of the "charges" of `C1`, `C2`, ... as listed in table `TX`. If the total charge is 0, then `DC` is 1, otherwise it is 0. If the character `"C_` is used, the result is the opposite. If the character `"T` is used, keep the term if the sum is positive or zero. If `"T_` is used, keep it if the sum is negative.

Here is an example of such a character table: `1` is assigned to the particle `Z`, and also `1` to its antiparticle `Z_`. If the second number `1` had not been given, it would have been taken to be the opposite of the first number, which would have given `-1` for the antiparticle in this case:

```
T TX: Z=1:1
```

Chapter XVII

Gamma Algebra

The equations and algorithms used in this chapter will not be derived here. The derivations can be found in a separate publication (M. Veltman, "Gammatrica," *Nuclear Physics* B319 (1989) 253–270).

1. Other Dimensions

The treatment of gamma's has become quite complicated in the current version of Schoonschip, essentially because gamma algebra can now be done in other than four dimensions; but things have been organized so that for most cases the default options are sufficient. It is nonetheless essential that a number of things be well understood.

other dimensions only for indices

The only quantities for which other than four dimensions can be specified are indices. There is no way to specify the dimensionality of a vector or of a dot product. For gamma matrix manipulations, Schoonschip takes vectors to be four dimensional, i.e., `G(L,p)` contains only `G(L,1)` through `G(L,4)`. That might seem to be a restriction, but in fact it is not. Refer to the `"C` option discussed below.

loop momenta

In practice it is the loop momenta that are not four-dimensional vectors in the sense just mentioned. There are several ways to deal with them. The easiest seems to be to do loop integrations before doing gamma algebra. First the problem is prepared by using the `"C` option to collect the gamma's (see below). Then the loop momenta are replaced by external four-dimensional momenta, or by index pairs with `N`-dimensional range. These can be handled properly. For the moment it will be assumed that all momenta occurring as arguments of gamma matrices are four dimensional, but that indices may be either four or `N` dimensional. No provision has been made to handle indices with other ranges than these two, except that one may use any algebraic symbol instead of `N`.

2. New Notation

loop notation

A new notation has been implemented. Instead of one loop index, the gamma's may now have two loop indices. This very much eases the task of ordering the gamma's when they are generated by an automatic procedure. Gamma's can be rounded up (i.e., "corralled") and put in the proper sequence based on the indices. The notation is:

```
G(i1,i2,mu)  Gi(i1,i2)  G5(i1,i2)  G6(i1,i2)  G7(i1,i2)
```

Here `Gi` is the unit matrix. The meaning of `G6` and `G7` is as before:

```
G6 = Gi + G5,   G7 = Gi - G5
```

gamma ordering

The ordering is based on the loop indices:

```
G(i1,i2,mu) * G(i3,i4,nu) * G(i2,i3,al)
   = G(i1,i4,mu,al,nu)
```

which corresponds to the old notation

```
G(L,mu)*G(L,al)*G(L,nu)
```

One may use the multi-index notation `G(i1,i3,mu,al,nu)` directly. It has actually been used internally (but with only one loop index) for a long time. `Gi`, `G5`, `G6` and `G7` may also appear in the list:

```
G(i1,i2,mu,G5,al,be,G6,ga,Gi,la,G7)
```

traces
strings

Such a string is called a *trace* if the first and second indices are the same. Otherwise the expression is called a *gamma string*, or *string* for short. A string may be odd or even, depending on the number of arguments, excluding `Gi`, `G5`, `G6` and `G7`. Thus, the string above is odd. Traces of odd strings are always zero.

numerical loop indices

To avoid a proliferation of indices, it is preferable to use integer numbers instead. For example:

```
G(1,2,mu) * G(3,4,nu) * G(2,3,al) = G(1,4,mu,al,nu)
```

reserved loop range

When dealing with more than one string, number ranges may be reserved for the different strings, thus avoiding confusion. Example:

```
X Vert(n,mu) = G(n,n+1,mu,G6)

Z xx = Ubg(1,mm,p)*Vert(1,mu)*G(2,3,nu)*Ug(3,mn,q)
          *Ubg(10,mn,k)*Vert(10,mu)*Ug(11,me,qq)
```

3. Review of Gamma Operations

In this section, we review the sequence of operations, with various options, that Schoonschip carries out when it works on gamma matrices. For future reference, we mention here the characters that correspond to the options. The command syntax and how the option characters are used in it are described in Section 4 on the `Gammas` command.

3.a. Collection

gamma roundup

Schoonschip begins its work on gamma matrices by rounding them up. For two-index gammas the ordering is determined by the indices. For gammas with one index (the old notation) the ordering is based on the order in which they appear in the expression at hand. By default the strings are normalized, i.e., directly neighboring equal arguments are eliminated,

normalization

G(...,x,x, ...) = D(x,x)*G(..., ...)

and traces of odd numbers of gammas (not counting `G5`, etc.) are set to zero. Normalization does certain other things, depending on the situation, such as collecting `G5`'s in four-dimensional strings. It can be suppressed with the option `"N_`.

`"N_`

After rounding up, the strings and traces appear with only one loop index. That index is chosen from among the indices that are present: the smallest number (or earliest mentioned index) is taken. Indices are “smaller” than numbers.

loop index consolidation

```
G(1,2,mu,al) * G(3,4,nu,G5) * G(2,3,la)
     → G(1,"s,"4,mu,al,nu,G5,la)
```

The character argument `"s` indicates a gamma string rather then a trace. The `"4` is the unit `G` matrix; the need for this argument will become clear below. The `1` is now the loop index. The following example generates a trace, indicated by the character argument `"t`:

`"s`, `"4`

`"t`

```
G(i1,2,mu,al) * G(3,i1,p,G5) * G(2,3,la)
     →  G(i1,"t,"4,mu,al,p,G5,la)
```

G-*strings*

For brevity these objects will be called G-*strings* in the following. The arguments may be indices, vectors or Gi–G7.

collection
"N
"C

At this point we have two types of G-strings, namely, "s- and "t-strings. This very first phase of the work is called *collection*. Normalization ("N) is optional. There is an option to stop working at this point ("C).

3.b. Preliminary Reduction

loop selection

"S, "T

The next step is to apply all the known algorithms to these objects. One may specify which strings are to be worked on, either by a specific loop index or by a number range. We discuss that further below. Schoonschip starts by scanning all arguments in the G-string. If all indices have the (default) range 1–4 then the string becomes a four-dimensional G-string. Any vectors that occur are considered to be four dimensional. If a string becomes four dimensional, the character argument "s or "t is changed to "S or "T. Some of the manipulations that are done on the strings:

- Adjacent identical arguments are eliminated. For traces that includes the first and last arguments.
- For "S- and "T-strings, any G5, G6 and G7 are anticommuted to the left; and the result is inserted in place of the "4 argument.
- Simple cases such as the trace of the identity are completely worked out.

Examples:

```
I mu,nu,al=N,be=N

 G(1,"s,"4,mu,nu,al)      →  unchanged
 G(1,"t,"4,mu,nu,al)      →  0 (odd trace)
 G(1,"s,"4,mu,al,al,nu)   →  N * G(1,"S,"4,mu,nu)
 G(1,"t,"4,mu,al,be,mu)   →  4 * G(1,"t,"4,al,be)
 G(1,"t,"4,G5,mu,G6,nu)   →  G(1,"T,"6,mu,nu)
                          →  4 * D(mu,nu)
 G(1,"s,"4,G5,mu,G6,nu)   →  G(1,"S,"6,mu,nu)
                          →  G(1,"S,"4,mu,nu)
                                + G(1,"S,"5,mu,nu)
```

The last step also belongs to the class of doing simple cases. The expansion of G6 would not have been done if there had been more index arguments. One may choose to stop here, by turning off all the options that follow.

3.c. Sum-Splitting

What comes next is *sum-splitting.* N-indices in "s or "t strings are split into four dimensional and N-4 dimensional parts. The quantity N_ is taken to stand for N-4. Here are the equations for the case of two index pairs: *sum-splitting* N_

```
I mu,nu,al=N,be=N

    G(1,"s,mu,nu,al,be,...,al,be)  →
          G(1,"S,mu,nu,al,be,...,al,be)
        ± G(1,"s,"_,be) * G(1,"S,mu,nu,al,...,al)
        ± G(1,"s,"_,al) * G(1,"S,mu,nu,be,...,be)
        ± G(1,"s,"_,al,be,al,be) * G(1,"S,mu,nu,...)
```

The sign is determined by the convention that the indices to be split are first moved to the very left. They are taken to anticommute with all four-dimensional indices and vectors, and to commute with G5, G6 and G7. There is an option ("A), for traces containing an even number of G5's or for strings regardless of the number of G5's, to specify anticommutation with G5. In that case G6 and G7 are expanded before the sum-splitting is done. For traces with an odd number of G5's, commutation is used even if the anticommutation option is specified, simply because the trace would not otherwise be well defined. The outcome would not be invariant under a cyclic rotation of the arguments. "A

Note the "_ character argument. The range of the indices in a "_-string is from 4 to N (not including 4). Both the string and trace of the unit "_ are one: "_

```
G(1,"s,"_) = 1
G(1,"t,"_) = 1
```

Aside from that, "_-strings are treated like the other nonfour-dimensional strings. For example, the elimination of a direct pair:

```
G(1,"s,"_,...,al,al,...) = N_ * G(1,"s,"_,...)
```

Sum-splitting requires that all N-indices occur in pairs. That is not a real restriction, because the pairs need not occur in one and the same string. One index may occur in one string and the other in another string, or even in some other function. *index pairs*

If a string contains only N-range indices then it is called *pure*, and no sum-splitting is done. Index pair elimination and trace evaluation are done directly on such a string. *pure strings*

The default is to do sum-splitting ("S). One can stop at this point, by suppressing the options that follow. "S

3.d. Unification

unification
"U_
Chisholm's equation

The next step is the *unification* of G-strings. That is not done ("U_) as the default, because it is not always advantageous, and because it may be necessary to indicate explicitly which strings are to be unified. Unification is based on Chisholm's equation:

```
G(1,"X,a1,a2,...,mu,b1,b2,...) * G(2,"T,mu,c1,c2,...,cn)
   =   2 * G(1,"X,a1,a2,...,c1,c2,...,cn,b1,b2,...)
     + 2 * G(1,"X,a1,a2,...,cn,...,c2,c1,b1,b2,...)
```

where "X may be "S or "T. Its application requires:

- The G-strings involved must be four dimensional.
- They must have one index in common (mu in the above).
- At least one of them must be a trace.

advantage

The general rule is that unification is advantageous if the G-strings have arguments in common other than the index mu shown explicitly above. The common arguments may be either vectors or indices. Thus, for example, there is an advantage if c1 and b1 are the same, because in the second term on the right-hand-side that can be worked out. Work can be stopped after unification by turning off the options that come next.

3.e. Index Pair Elimination

Kahane algorithm
"I
"i

The next step is to eliminate index pairs in the G-strings. For four-dimensional strings there is the Kahane algorithm which eliminates all pairs in a given string in one sweep; for "s- and "t-strings there is a single pair algorithm that can be applied repeatedly. The default is to do pair elimination ("I). One may specifically allow ("i) or inhibit ("i_) index pair elimination in "_-type strings separately. To stop the work here, turn off the remaining options.

3.f. Vector Pair Elimination

"P

Vector pairs in four-dimensional strings are eliminated ("P) in the next step. This is also a single pair algorithm that can be applied repeatedly. It is called the *P*-algorithm. One can stop here, by turning off the remaining option.

3.g. Traces and Final Reduction

The final step is to actually evaluate the traces, and to reduce the four-dimensional strings to a standard form (`"W`). This is called the *W*-operation (for final work). `"W`

3.h. Summary of Operations

To summarize, the sequence of operations is the following:

- Rounding up all strings. This step cannot be avoided. All the `G`'s are rounded up, even those that are not going to be worked on. By default the strings are normalized (`"N`).
- Preliminary reduction. Basic algebraic relations are worked out.
- Sum-splitting (`"S`), done by default. The 't Hooft–Veltman convention is used consistently throughout. For traces with even numbers of `G5`'s, there is an option (`"A`) to have anticommutation of `G5`'s instead. Even if one specifies `"A`, the 't Hooft–Veltman convention is still used for traces with odd numbers of `G5`'s. The option `"A` does nothing to strings.
- Unification. Not done by default (`"U_`).
- Index pair elimination (`"I`), also for pairs in `"_`-type strings (`"i`) (the latter may be inhibited with the `"i`-option). This is done by default.
- *P*-algorithm for four-dimensional strings. Done by default (`"P`).
- *W*-operation. Working out reduction for four-dimensional strings and final traces (`"W`).

4. Gammas Command

Each option is characterized by a single character, namely, `N`, `A`, `S`, `U`, `I`, `P` or `W`. *options* The command that does all this is `Gammas`

```
Id,Gammas
```

The default is to work on all strings. For most cases that use the two-index `G` notation, this will do. If there are single-index `G`'s (old notation) then one has to specify which are strings and which are traces. The default is to take them to be strings. If a trace is intended, loop indices have to be mentioned:

```
Id,Gammas,Trace,i3,i4
```

In strings `i3` and `i4` the `"s` is replaced by `"t`.

specific loops

When one or more specific indices is mentioned in the `Gammas` command, only those loops whose loop indices appear in the list are worked on. Separate options can be specified for any individual loop index. Options can be on or off. They are turned on by the appropriate character argument, and turned off when the argument is followed by an undersore (`_`). Furthermore, in case numbers are used as loop indices, one can indicate ranges of loop numbers. Here is the syntax:

```
Id,Gammas,C1,C2,i1,1-4:C3,C4,Trace,i2:C5,C6,i3,9-10
```

options

The character arguments `C1`, `C2`, etc., may be any of the following:

`"N`	(`"N_`)	normalize
`"A`	(`"A_`)	do anomalous trace
`"S`	(`"S_`)	do sum-splitting
`"U`	(`"U_`)	unify strings
`"I`	(`"I_`)	do index pair reduction
`"i`	(`"i_`)	do index pair reduction in `N_`-type strings
`"P`	(`"P_`)	do `P` tricks
`"W`	(`"W_`)	do final work

`"C`

There is an additional option `"C`. It turns off all options except for `"N`, i.e., only collection and normalization are done, with the further difference that there is never a specialization to four dimensions. This is essential if there is an integration over a vector which is not four dimensional. When an `N`-dimensional integration is to be done, one typically first collects all `G`'s with the `"C` option, then does the integrations, and after that does the remaining work through another `Gammas` command.

To turn off normalization as well, include `"N_` along with `"C`.

default options

The default is defined by those character arguments that are mentioned first. If none is mentioned, the default is `"N`, `"A_`, `"S`, `"U_`, `"I`, `"i`, `"P`, `"W`.

`Trace`

`G`-strings with loop indices mentioned after the `Trace` argument are forced to be traces (`"t`).

loop index ranges

A loop index may also be specified by a number range. Any `G`-string whose loop index is included in the range is worked on. If the loop index or the range is followed by a colon, then the character arguments after the colon define options specific to those loops. They override the default options. Example:

```
Id,Gammas,"A,"U,1-9:"A_,10-20:"U_,"I_
```

Strings with loop indices 1–9 and 10–20 are considered. Unification is attempted on loops 1–9. Loops 1–9 have commuting `G5`'s with respect to sum-splitting, 10–20 are anticommuting for traces with an even number of G5's or for strings with any number of G5's. No index pair elimination is attempted on loops 10–20, but index pairs in N_-type strings are still worked on. To inhibit that, "i_ must also be mentioned.

Please note the use of the colon (:). It is there because one can in principle also use character arguments as loop indices for two-index G's (character arguments are ordered between numbers and indices). Several characters are illegal in that respect: "S, "T, "s and "t, at least if they occur as the second argument in a two-index G.

character loop indices

Finally, we note that it is sometimes necessary to do the Gammas command twice when a string and a trace are both present, for example,

```
Id,Gammas
*yep
Id,Gammas
```

In general, one may need such a repetition for every string/trace factor.

5. Vector Extraction

As noted before, all vectors are considered to be four dimensional in all gamma operations. There are at least two ways get around that restriction, and some special commands have been introduced to make it simpler to do so.

5.a. All and Iall Commands

Consider a case in which there occurs a momentum p that is to be integrated over N-space, and suppose p occurs as a G argument:

```
G(1,2,al,be,p,al,be,p)
```

(the two-index notation is used from now on). The first step is to get the p out. The following new command, essentially a combination of existing commands, can be used to good effect here:

All

```
Id,All,p,N,F
```

In this command p is a vector, N is an optional argument specifying the range of any indices to be created, and F is a function defined in an F list. The command rounds up all p's, including those occurring in functions and dot products, and collects them in the function F.

Here is an example (the S declaration is the old version of A, and still works):

```
V p
F F
S N

Z xx = pDq*pDp*F1(aa,bb,p)*p(mu)
Id,All,p,N,F
*end
```

This is the output:

```
xx = + F1(aa,bb,Naa)*F(Naa,mu,q,Nab,Nab) + 0.
```

Here the quantities Naa and Nab are indices that have been created, with range N. If one substitutes

```
F(i1~,i2~,i3~,i4~,i5~) = p(i1)*p(i2)*p(i3)*p(i4)*p(i5)
```

then the original expression reemerges. This can actually be done more easily with the command Iall, the inverse of All, Iall

```
Id,Iall,p,F
```

which searches for F and restores it to the original expression.

Both All and Iall have an alternate syntax. See Chapter XII, *Substitution Commands*, as well as the example further below.

The All command may now be used to eliminate p from the G-string. Next the integration may be done, for example,

```
Id,F(i1~,i2~) = D(i1,i2)*F20(k) + k(i1)*k(i2)*F22(k)
```

where k is a four-dimensional vector. After that the gamma work can be done.

Here is the full example with output:

```
I al,be,mu,nu
S N,N_
V p,k
F F,F20,F22

Z xx = G(1,1,al,be,p,al,be,p)

Id,All,p,N,F
Id,F(i1~,i2~) = D(i1,i2)*F20(k) + k(i1)*k(i2)*F22(k)
```

```
Id,Gammas
*end

xx = + F20(k) * ( 64 - 32*N_ )  +  F22(k) * ( 16*kDk )
```

It should be emphasized that the symbol N, occurring in the All command, must be defined in the symbol list, together with the symbol N_. This is done automatically if such a symbol already occurs as a range in an index list, but in the above example that did not happen. The All command will not work otherwise. The quantities p and F must also have been defined before, either implicitly or explicitly. N cannot be defined implicitly, because it would be taken as an index rather than an algebraic symbol. Again, N_ is supposed to be N-4.

predefinition of N, N_

Warning: For dimension N, one must be careful about the use of the Iall command, which is purely four dimensional. See in particular the discussion of the Ndotpr command in the following subsection.

The All command is really a combination of three existing substitutions, namely

```
Id,p(mu~)=...
Id,Dotpr,p(mu~)=...
Id,Funct,p(mu~)=...
```

In addition, All smoothly handles the creation of the function F with its undetermined number of arguments.

The substitutions above have been modified slightly in this version of Schoonschip. The range of the index mu is now taken into account. If an index is created in the course of working out these substitutions, it is given the range of mu.

5.b. All with Ndotpr Command

The second method for dealing with vectors in G-strings is as follows. Let there be a string containing several vectors as arguments:

```
G(1,1,mu,G5,p,q,nu,G5,q,p)
```

Now use the All command, but specify the function G rather than the vector p:

```
Id,All,G,N,F1
```

Again, N, N_ and F1 must have been defined before. This command takes all vectors out of the specific function mentioned (G in this case) and collects them, together with the indices that are created, into the function F1:

```
F F1
I mu,nu
V p,q

Z ft = G(1,1,mu,G5,p,q,nu,G5,q,p)

Id,All,G,N,F1
*end
```

The result is:

```
 ft = + G(1,1,mu,G5,Naa,Nab,nu,G5,Nac,Nad)
           *F1(p,q,q,p,Naa,Nab,Nac,Nad)
```

Now the gamma work can be done. The result is rather messy, and it is displayed here (slightly edited) for a simpler case:

```
F F1
I mu,nu
V p,q
A N,N_

Z ft = G(1,1,mu,G5,p,nu,G5,q)

Id,All,G,N,F1
P output
*yep

ft = + G(1,1,mu,G5,Naa,nu,G5,Nab)*F1(p,q,Naa,Nab)

Id,Gammas
P output
*yep

ft =    + 4*F1(p,q,Dim_N_,Naa,Dim_N_,Naa)*D(mu,nu)
        + 4*F1(p,q,Dim_4,mu,Dim_4,nu)
        + 4*F1(p,q,Dim_4,nu,Dim_4,mu)
        - 4*F1(p,q,Dim_4,Naa,Dim_4,Naa)*D(mu,nu) + 0.
```

`Dim_` Note that Schoonschip has included arguments ahead of the indices inside the function `F1` to show the range. The function `F1` with these prefixes cannot be handled by any substitution or command except the command `Ndotpr`, especially created for this situation. It works out the `F1` arguments and creates various symbols analogous to dot products (but they are not, they are symbols) showing the summation range:

`Ndotpr`

```
    Id,Ndotpr,F1
    P output
    *end

    ft =   4*p(mu)*q(nu) + 4*q(mu)*p(nu)
         + D(mu,nu) * ( 4*pq_ - 4*pq4 ) + 0.
```

Here pq_ means the dot product of p and q, but including only components between 4 and (including) N. Similarly pq4 denotes a dot product where only the first four components enter. Obviously pq4 + pq_ = pDq, where pDq is the normal dot product. Schoonschip sometimes generates a symbol pq0 as a replacement for pDq in these situations.

The symbols pq_, pq4 and pq0 are algebraic symbols. They are created during execution; and if a symbol list is requested in a later section (after another *yep, for example) the A list would be:

```
    A i=i, N, N_, pq_, pq4, pq0
```

A curiosity: if the "A option had been specified in the Gammas command, the term pq_ would have come out with the opposite sign. That is because there is one G5 between p and q. "A

Here is the example once again, but without the intermediate printout:

```
    F F1
    I mu,nu
    V p,q
    A N,N_

    Z ft = G(1,1,mu,G5,p,nu,G5,q)

    Id,All,G,N,F1
    Id,Gammas,"A
    Id,Ndotpr,F1
    *end

    ft =   4*p(mu)*q(nu) + 4*q(mu)*p(nu)
         + D(mu,nu) * ( - 4*pq_ - 4*pq4 )
```

6. Examples

That concludes the discussion. The rest of this chapter consists of more examples.

```
C G collection including spinors.

I mu=N,nu=N
V p,q

Z xx = G(i1,i2,mu)*G(i2,i3,nu)*Ubg(i1,M,p)*Ug(i3,M,q)

Id,Gammas,"C
*end

xx = + Ubg(i1,M,p)*G(i1,"s,"4,mu,nu)*Ug(i1,M,q) + 0.

C No problems with complex conjugation including a G6:

I mu=N,nu=N
V p,q

Z xx = Conjg( G6(i0,i1)*G(i1,i2,mu)*G(i2,i3,nu)
         *Ubg(i0,M,p)*Ug(i3,M,q))

Id,Gammas,"C
*end

xx = + Ubg(i3,M,q)*G(i3,"s,"4,nu,mu,G7)
         *Ug(i3,M,p) + 0.
```

For a pure string (containing only N-indices), all index pairs can be eliminated, and traces can be evaluated completely. Here is an example with a string and a trace:

```
I mu=N, nu=N, al=N, be=N, ga=N

Z xx =   G(i1,i2,mu,ga,nu,al,be,ga)
       + G(i1,i1,mu,ga,nu,al,be,ga)
Id,Gammas,"C
P output
*yep

xx = + G(i1,"s,"4,mu,ga,nu,al,be,ga)
     + G(i1,"t,"4,mu,ga,nu,al,be,ga)

Id,Gammas
*end

xx = + D(mu,nu)*D(al,be) * ( 8 - 4*N )
     + D(mu,al)*D(nu,be) * ( - 8 + 4*N )
     + D(mu,be)*D(nu,al) * ( 8 - 4*N )
```

```
        +   G(i1,"s,"4,mu,nu,al,be) * ( - N_ )
        - 2*G(i1,"s,"4,mu,be,al,nu) + 0.
```

Except for pair elimination, nothing has been done to the strings. One could actually bring them into some sort of normal form, but the advantages of that are unclear, and it is also uncertain what the best normal form might be. The work involved and the number of terms generated woud be considerable. For that reason nothing has been implemented. But a possible procedure will be discussed now.

Define the function AG to be totally antisymmetric in all of its arguments except for the first, which is the loop index. If the number of arguments beyond the first is m, then AG is equal to the totally antisymmetric combinations of m gammas divided by $m!$. Examples:

```
AG(L)          = Gi(L)
AG(L,mu)       = G(L,mu)
AG(L,mu,nu)    = 1/2 * { G(L,mu)*G(L,nu)
                             - G(L,nu)*G(L,mu) }
AG(L,mu,nu,al) = 1/6 * { G(L,mu)*G(L,nu)*G(L,al) - ... }
...
```

Any string can be written as a linear combination of such functions AG:

```
G(L,"s,m1,m2,m3,...) =   a0*AG(L) + a1*AG(L,n1)
                       + a2*AG(L,n1,n2) + ...
```

For example:

```
G(L,"s,"4,mu,nu) = D(mu,nu)*AG(L) + AG(L,mu,nu)
```

For the case of three indices:

```
G(L,"s,"4,mu,nu,al) = D(nu,al)*AG(L,mu)
   - AG(L,nu)*D(mu,al) + AG(L,al)*D(mu,nu)
   + AG(L,mu,nu,al)
```

The number of terms increases faster than the number of terms for a trace. For four indices, the number of terms is 10 (compared to 3 for a trace); for five it is 11; and for six it is 76 (compared to 15). Here is a program that computes this expansion for the case of four indices (where only a0, a2 and a4 are non-zero):

```
I mu=N, nu=N, al=N, be=N, b1=N, b2=N, b3=N, b4=N

Z xx = { 1/(4*DB(4)) * G(L,"t,"4,mu,nu,al,be,b1,b2,b3,b4) }
              * AG(L,b4,b3,b2,b1)
```

```
          + 1/(4*DB(2)) * G(L,"t,"4,mu,nu,al,be,b1,b2)
               * AG(L,b2,b1)
          + 1/4 * G(L,"t,"4,mu,nu,al,be) * AG(L)

   Id,Gammas
   Id,Asymm,AG,2,3,4,5,6
   *end
```

Note that AG was specified to be antisymmetric in more arguments than actually occur. Schoonschip simply ignores the excess.

The time required is considerable because of the large traces that have to be computed. For six indices, the time to evaluate AG is about 150 seconds (8 MHz M68000). The method shown is, however, not the most efficient.

Chapter XVIII

Statements

We provide here an alphabetical listing of Schoonschip statements, including some that are discussed more fully in succeeding chapters. A functional listing is given in Chapter XXII, *Summary*, which also includes a listing of declarations. The logical distinction between a "declaration" and a "statement" is not totally unambiguous. Essentially, a declaration is a passive statement that defines or restricts a symbol type, while statements manipulate symbols or files. Substitutions and substitution commands do manipulation, but we put them in their own special category, not included in this chapter. Conditional statements form their own category as well. Conditional input statements (Chapter XIII) are strictly speaking not part of the Schoonschip language at all, but come under the category of compiler/interpreter directives.

Here is the list of statements:

`Bdelete ...`	Delete blocks.
`Beep` *`options`*	If no options all beep requests are cleared.
	If `#,t` issue `#` beeps at termination.
	If `#,*` issue `#` beeps when concluding a section (`*` delimited).
	If `#,e` issue `#` beeps in case of an error.
`Common ...`	Declare common files.
`Delete ...`	Delete files.
`Digits #`	Set the number of digits to be printed for floating point numbers.
`Directory ...`	Define directories and file number ranges to be used for large files. See Chapter XX, *File Handling Statements*.

`Dostop on`	Sets the `Dostop` flag on from outside a do-loop, to force exit from the do-loop at a `*` line.
	`Dostop off` clears the flag to its default, off condition. See Chapter III, *Input Facilities*, and also the `Dostop` substitution command. Tricky.
`Dryrun #`	Do a “dry run”. Print at most `#` terms at completion.
`End`	End of external disk file. See `Read`.
`Enter fname`	Enter common files from disk file `fname`.
`External fname`	Read position independent, machine language disk file `fname` into memory. The command `Extern` executes the file.
`Fortran #, Brackets, Compression, .,A`	Fortran output in earlier versions of Schoonschip often needed considerable editing. That section of Schoonschip has been rewritten. The `#` argument specifies how many continuation lines Fortran statements may have. `Brackets` directs Schoonschip to place brackets around negative exponents. Next, the default Fortran output has only complete terms on a line (if possible), rather than the compressed format desirable in the old days when cards were used. `Compression` selects the old, compressed format. The `.` option instructs Schoonschip to supplement every integer with a decimal point, as required by some Fortran compilers. Finally, the character `A` specifies output for the A compiler. Every item in the statement is optional, the defaults being: 9 continuation lines/statement, no brackets, no compression, no `.` after integers, Fortran output only.
	Only the first character of `Brackets` and `Compression` is significant, and it may be upper or lower case.
	A minus sign in front of `Brackets` or `Compression` switches back to the default; e.g., `Fortran -Brackets` instructs Schoonschip not to put brackets around negative exponents.
	The exponent of a number of normally indicated by the character `E` (e.g., `1.23E+10`). If a number of digits larger than `5` has been specified by a `Digits` statement, then `D` is used instead (`1.23D+10`).
`Freeze xfile`	Freeze common file `xfile`. See Chapter XIX, *Handling Large Problems*.

`Keep ...`	Keep local files over a `*next`.
`Large #`	Here `#` is a number in the range 1 to 5 inclusive. The default is 2. Specify the number of intermediate output files to be used for large file sorting. See Chapter XX, *File Handling Statements*.
`Lprinter`	Print output lines up to 130 characters.
`M xx = ...`	Compare the expression `xxx = ...` to a preceding line of `R` input. To be followed by `*yep`. See Chapter XXI.
`Maximum #`	Specify the maximum size of intermediate output subfiles. The default is $110,000$. See Chapter XX, *File Handling Statements*.
`Names ...`	Declare name lists for quantities in common files. The common files have been read previously from external disk files by the Enter statement.
`Nprint ...`	Do not print `Z` files.
`Nshow`	Switch `Showl` off.
`Oldnew xx=yy`	Rename quantity `xx` as `yy`.
`Outlimit #`	Set a limit of `#` bytes on disk usage. For large file usage this sets the limit on total disk space used for the intermediate output files. Default: $500,000 = 0.5$ megabytes.
`Overflow off`	Suppress short number overflow error trapping (exponents, function arguments). May appear in the `*fix` section. `Overflow on` reactivates overflow error trapping.
`P` *`options`*	Set options for printout. See Chapter VII, *Print Options*.
`Print ...`	Specify `Z` files to be printed.
`Precision #`	This specifies the precision to be used in dealing with numbers. The number `#` applies to various internal situations. In the output, two terms are taken to cancel if their addition gives a cancellation up to `#` digits. Comparing numbers with the `Coef` command uses the same precision criterion. Also, when writing terms to disk, every number is truncated to 10 digits if `#` is 9 or less. That saves considerably on disk space. The default is 22 digits.

`Progress`	In case there is no output to the terminal, print a dot (`.`) to the terminal every 128 terms to show progress. Identical to `Progress on`. `Progress off` switches the printing of dots off after it has been turned on.
`Punch ...`	Specify `Z` files for which Fortran output is to be done.
`R xxx = ...`	Transform the Schoonschip text output expression `xxx =` ... into input for `*yep`. Useful for large expressions that would normally overflow the input space. See Chapter XXI.
`Rationalize #`	By default, Schoonschip tries to rationalize numbers. The number `#` specifies to how many digits the numbers must agree. For example, Schoonschip will rationalize $\pi = 3.141592654\ldots$ to $22/7 = 3.142857143\ldots$ if 3 is specified. Specifying 5 gives $355/113 = 3.14159292\ldots$, which actually agrees to 7 digits. It should perhaps be noted that after every rationalization Schoonschip actually performs the division and compares the result with the input number, just to be sure. If 0 is put for `#`, no rationalization is done. The default is 22 digits, with check up to 26 digits.
`Read fname`	Read external disk file `fname`.
`Round on`	Numerical expressions as function arguments are normally converted to short integers by truncation. After a `Round on` statement, the conversion goes by rounding to the nearest integer. `Round off` resets to the default conversion for function argument expressions, truncation.
`Screen`	Print output lines up to 80 characters (default).
`Showl`	Short for `Showlength`. Show statistics for large disk files as they are being written.
`Silence`	Instruct Schoonschip not to print `Ready` to the screen and wait for an answer for certain PC's that normally clear the screen immediately upon termination.
`Synonym name1 =name2`	Every occurrence of `name1` between single quotes is replaced by `name2`. This is very much like the replacement of `BLOCK` arguments, but is not limited to a block. The end of the range of validity of a set of synonyms is defined by a `Synonym` statement with no arguments. At most 16 synonyms are allowed.

`Traditional`	Do traditional sorting. See Chapter XX, *File Handling Statements*.
`Write fname`	Write all common files to disk file `fname`.

Chapter XIX

Handling Large Problems

Considerable attention has been focused on the question of large output. The traditional output sorting method of Schoonschip is inadequate for truly large problems, with more than 50,000 output terms. Moreover, as small computer systems do not have infinitely large disk space available (with 20 megabytes not uncommon), methods of economizing on disk space are important. In this chapter, we describe the methods and tools that Schoonschip makes available for attacking large problems. Most of the actual statements and their syntax are given in the following chapter, *File Handling Statements*, with the major exception of the `Freeze` statement, which is described in Section 4 of this chapter.

1. Disk Utilization

traditional sorting

The traditional system works as follows. When terms that have been generated arrive at the sorting routine, they are compared with the terms already present in the output store. If like terms are found, coefficients are added. If no match is found for a term and there is space left, it is entered into the output store. Otherwise the term is simply written to disk, and the addition of further terms to the output store is inhibited. This method is quite wasteful of disk space. The timing depends crucially on the disk system, and is generally slow.

new sorting
`Large`

The new method uses up to five output files; the number can be specified by the `Large` statement described in the next chapter. If the output store is full, its contents are dumped, say on `file1`, and sorting resumes. If the output is again full, the store is again dumped, say on `file2`, etc. If the maximum number of files is reached, then merging is done rather than straight dumping. Assume for example that a maximum of three output files has been specified. And assume that three files, `file1`, `file2` and `file3` have already been created. At this point, when the output store is again full, a merge takes place. The smallest file (say `file2`) is selected, and merged with the contents of the output store to produce a new file, called for the sake of argument `file2a` (the actual naming conventions are given below). And so on. When the whole problem is finished there are three possibly very large files. As a final step, these three files are piecewise merged into the output store and printed, etc.

With this method, disk usage is already considerably less than with the traditional method, because only sorted expressions are written to disk. Still, quite a bit more disk space may be used than needed for the final size; each of the above three files may well have the same order of magnitude as the final size.

If disk space is really crucial, the above method may be applied with the specification that only one file is to be used. The first overflow produces a file. The second causes that file to be merged with storage, giving rise to a new file. Disk usage is at a maximum when the last part of the file being merged is read, and the new file is almost written out. Disk usage is then about twice the size of the final result.

fractured files

To deal with this final factor of two, a fractured file system has been implemented. A fractured file is a file split up into smaller files. Each of these smaller files is a normal file as far as the the operating system is concerned, and only one at a time is open for reading or writing. Moreover, each is immediately deleted after being read. In the case described above, one would for example produce `file1`, fractured into five files called `file1_0`, . . . , `file1_5`. When merging this with the output store, `file2`, fractured into `file1a_0`, . . . , `file1a_7` (for example) is created. When the latter subfiles are being created, most of the subfiles of `file1` have been deleted. The maximum disk space used with this method is usually about equal to the size of the final file plus an overshoot. The overshoot may be the size of one extra subfile. Of course the overshoot is problem dependent; computing the difference between two large but equal expressions leads to a final size of zero, but the intermediate result may be as large as either of the original expressions. Schoonschip assumes that an overshoot is at least as large as the allowed size of the output memory. That is used to skip the last merge, by dumping the last output store onto `file2` and then merging the possibly very large `file1` with the much smaller `file2` while printing.

This method appears to be about the best one can do to minimize disk usage. The timing is reasonable, but goes up quadratically with the size of the final file. To give an idea, the merge time for a 6 megabyte final result is about 6,000 seconds, for a typical M68000 system (8 MHz).

merge time

If space is no issue, then one can specify more than one dump file. The sorting time decreases roughly linearly with the number of files. That is of course very problem dependent.

subfile names

The way files are actually named is the following. All names are of the form

`TAP`$xyyy$`L`

where x and yyy are hexadecimal digits. The equivalent of `file1_0`, `file1_1`, ..., `file1_10`, ..., in the discussion above is `TAP6000L`, `TAP6001L`, ..., `TAP600AL`, ... The equivalent of `file1a_0`, etc., is `TAPB000L`, etc. Then `file2_0` is `TAP7000L`, `file2a_0` is `TAPC000L`, etc. Thus the two groups of output files are `TAP6` through `TAPA` and `TAPB` through `TAPF`. When merged with storage, file `TAP6` produces file `TAPB`, and conversely.

`*yep`

The `yep` file has also been made into a fractured file. The name used is `TAP4`yyy`L`, with yyy starting at `000`. In this way, disk usage for `*yep` is minimized.

2. Other Factors

output memory

Output memory size is a very important factor. Sorting time decreases roughly linearly with increasing memory size. To allow for adaptation to individual systems, the size of the output memory can now be specified on the command line when Schoonschip is called. See Appendix A, *Command Line Options*. Be careful, however, with systems having dynamic memory management, such as Sun's. If more than the available memory is requested, those systems simply accept the request and swap memory onto disk as needed. That would of course be highly counterproductive, and would bring Schoonschip virtually to a halt.

number format

`Precision`

The format of numbers plays a role, because integers (the largest is 2,000,000,000) take less space on the disk than most other numbers (6 versus 14 bytes for any one term). Thus if all numbers can be made into integers by means of some overall factor, that will improve both speed and output size. The factor can be divided out later, if desired. If a precision of less than 9 digits is specified (`Precision 9`) then all numbers written to disk will be rounded so as to fit into 6 bytes.

factorization

`B` *list*

It is also important how an expression is factored. Factorization is under control of the user through the `B` list. In the output store, the terms are stored precisely as they are printed:

```
head1 * ( tail11 + tail12 + ...) + head2 * (tail21 + ...)
   + ...
```

Without factorization, much more space would be used:

```
head1*tail11 + head1*tail12 + ...
```

Too much factorization is also wasteful:

```
head1 * (tail1) + head2 * (tail2) + ...
```

3. Tuning Facilities

A number of facilities have been provided for tuning large problems.

`Dryrun` Notable among them is that one can execute a "dry run." In a dry run, Schoonschip writes no files, and simply discards any full output store. Estimates of the necessary time and disk space are printed out. The disk space estimate is for the minimum needed (i.e., only one output file to be used). There is an option to print a limited number of output terms, so one can inspect for optimal factorization.

`Directory` If more than one device is available for storage, then it may be desirable to split up the output files over different devices or directories. That can be done with the `Directory` statement, described in the next chapter, which uses the file naming system described at the end of Section 1.

`Maximum` Finally, the maximum size of subfiles can be specified.

4. Freeze Statement

`Freeze` A useful tool for handling large problems is the `Freeze` statement. Suppose some further work needs to be done on a large common file, and in particular suppose the work is to be done only on certain factors. If the file has, say 10,000 terms, then the work has to be done 10,000 times. To cut down the amount of work as well as the size of the problem, the following can be done.

`B` *list* First, when producing the `Z` file, specify those factors that are to be worked on in the `B` list. For example, if an expression must be integrated over some variable `x`, with `X1`, `X2` and `X3` denoting `x`-dependent factors:

```
Common xfile
B x,X1,X2,X3
```

```
Z xfile = ...
```
substitutions, commands
```
*begin
```

The result will be of the form:

xfile = x * ($terms_1$) + x^2 * ($terms_2$) + X1 * ($terms_3$)
   + ...

There may be many terms inside the brackets. As the `Z` file must be a common file, it now resides on disk. The statement Freeze

```
Freeze xfile
```

(placed after the `*begin` line) produces the following result. The common file `xfile` is read and split up into one main file and possibly many "frozen" subfiles. The main file will still be called `xfile`, and is as follows: *begin DF

```
xfile = x * DF(xfile,1) + x^2 * DF(xfile,2)
   + X1 * DF(xfile,3) + ...
```

The subfiles are denoted by `x_file`:

x_file(1) = $terms_1$
x_file(2) = $terms_2$
x_file(3) = $terms_3$
...

Frozen subfiles are not directly accessible like other files. They may be referred to only by means of the `DF` function. There is a command, `Expand`, that allows substitution of the subfile for the `DF` function. For example, DF Expand

```
Z xx = DF(xfile,3)
Id,Expand,xfile
*end
```

will produce the contents of `x_file(3)` for `xx`. This may also be used to restore the common file to its original shape:

```
Common xfilp
B x,X1,X2
Z xfilp = xfile
Id,Expand,xfile
Delete xfile
*begin
```

The new common file `xfilp` will be precisely the same as the original `xfile`. Deleting (but not replacing) a frozen main file causes the deletion of all frozen subfiles.

It should be clear that work on the frozen main file `xfile` will be much less voluminous, since it will have many fewer terms. After the work has been done, one can reestablish the complete common file by using the `Expand` command.

`Freeze` is a powerful tool, but it must be used with some care. An important point is that file arguments are *not*, repeat *not*, transferred to the subfiles. For example, defining

```
    Z xfile(a1,a2) = ...
```

and then freezing `xfile` results in a number of subfiles. If we now use `xfile` with different arguments,

```
    Z ha = xfile(mu,nu)
```

and then expand the subfiles, we do not replace `a1` and `a2` in the subfiles by `mu` and `nu`. Thus, as a general rule, files that are to be frozen should not have arguments.

Created summation indices are particularly tricky here. It can happen that one of a pair of indices appears outside of a parenthesis, and that the other appears inside and becomes part of a frozen subfile. Since Schoonschip often renames summation indices, the outside index may get renamed, while the one inside stays the same, giving an erroneous result. This kind of "splitting" of repeated indices is thus to be avoided when files are to be frozen. See the discussion of the `All` command in Chapter XII, *Substitution Commands*, for a way to avoid this problem for indices created by that command. `All`

Chapter XX

File Handling Statements

This chapter assumes that the reader is familiar with the preceding chapter on handling large problems. Like all statements, the statements described below may be in upper or lower case, and only the first two characters (three if confusion is possible) are significant.

To specify traditional sorting: `Traditional`

```
Traditional
```

The default method in effect for any particular version of Schoonschip is a matter of local experience.

To specify the new sorting method: `Large`

```
Large #
```

where `#` is a number in the range 1 to 5 inclusive. It specifies the number of output files to be used. The default is currently 2.

To specify the maximum size of subfiles: `Maximum`

```
Maximum #
```

where `#` is the maximum subfile size in bytes. The default is 110,000.

For each of the subfiles a directory prefix may be specified. One may either specify a collective prefix for all subfiles regardless of the first number (`6-F`), or specify individual prefixes. Here is the syntax: `Directory`

`Directory` $\#_1$`,prefix1,`$\#_2$`,prefix2,`$\#_3$`,prefix3,`...

For any subfile, the number designating the subfile is compared with the chain of numbers above, which must be in ascending order. The prefix between the two numbers (including the first but not the second) whose range includes the file number will be used. If the number is less than 1000 hex (= 4,096), then only the last three hexadecimal digits of the file number are used, otherwise all four are used. The prefix is simply glued in front of the name `TAP`..., as described in the previous chapter. The special notation `.` (just one period) means no prefix.

first statement `*fix`

If present, the `Directory` statement must be the very first in any Schoonschip program, and it must be in the `*fix` section.

hexadecimal notation

It should be recalled that Schoonschip accepts hexadecimal numbers according to the notation `0x`..., where ... are hexadecimal digits. For example, `0x100` is the same as `256`.

Here is an example:

```
Directory 0,/usr/tmp/,10,/user1/tmp/,100,.
```

Files `TAP`x`000L` through `TAP`x`009L` are prefixed with `/usr/tmp/`; for example,

```
/usr/tmp/TAP6000L
```

Files `TAP`x`00AL` through `TAP`x`063L` are prefixed with `/usr1/tmp/`. Files `TAP`x`0064L` and up get no prefix. There is a collective sharing of directories; for example, `TAP6001L` and `TAPB001L` get the same prefix.

Another example:

```
Directory 0x4000,/usr/tmp/,0x6000,.,0xB000,/usr/tmp1/
```

Here the prefix `/usr/tmp/` is assigned to files `TAP4`yyy`L` and `TAP5`yyy`L`. Files `TAP6`yyy`L` through `TAPA`yyy`L` get no prefix, and files `TAPB000L` and up get `/usr/tmp1/`. Files `TAP0`yyy`L` through `TAP3`yyy`L` (not yet used) get no prefix.

The default is no prefix.

`Outlimit`

The old statement

```
Outlimit #
```

specifies the maximum total number of bytes of disk space that can be used. Default: 500,000.

`Showlength`

To get information on the size of the various files being written, use the statement:

```
Showlength
```

`Nshow`

That causes Schoonschip to print the total size and the name of the last subfile, after completion of the file. The statement

```
    Nshow
```

Switches off `Showlength`.

The statement Dryrun

```
    Dryrun #
```

causes Schoonschip to do a dry run as described in the previous chapter. Here `#` specifies the maximum number of terms to be printed. If `#` is 0 or absent, no dry run is made. This statement may also appear after a `*yep`. *yep

Chapter XXI

R Input

Sometimes one wants to work on very large input expressions, larger than Schoonschip can accommodate in its input space. The `R` input facility is designed specifically for that purpose. More precisely, it is geared towards input that has exactly the same form as normal Schoonschip output. `R` input for one problem can be prepared by editing text output from another problem. Put an `R` in column 1 of the line containing the file name; Schoonschip will read what follows and transform it into the format that `*yep` uses. After a `*yep` the expression can be worked on.

`R` *input*

`*yep`

As an example, consider a case where a previous problem has produced the output

```
xxx = + [b-a]^-1
  * ( [x+a]^-1 - [x+b]^-1 ) + 0.
```

To ensure that all symbols retain the same meaning in the new problem, specify the `P lists` option in the problem that produces the output; and include the resulting lists with the new input. Then put an `R` in column 1 and a blank in column 2 of the line containing the file name `xxx`:

`P lists`

```
A i=i, [b-a], [x+a], [x+b]

F D, Epf=i, G=i, Gi, G5=i, G6=c, G7, Ug=c, Ubg, DD,
  DB, DT, DS, DX, DK, DP, DF=u, DC

R xxx = + [b-a]^-1
  * ( [x+a]^-1 - [x+b]^-1 ) + 0.

*yep
```

```
Id,[b-a]^-1=bma

*end
```

There is no limit on the length of the expression other than available disk space.

M

`R` input can be very useful for editing expressions or processing parts of large expressions. It can also be used effectively for comparing outputs. To do that, put the second expression after the first, but put an `M` (for minus) in column 1 rather than an `R`:

```
A i=i, [b-a], [x+a], [x+b]

F D, Epf=i, G=i, Gi, G5=i, G6=c, G7, Ug=c, Ubg, DD,
  DB, DT, DS, DX, DK, DP, DF=u, DC

R xxx = + [b-a]^-1
  * ( [x+a]^-1 - [x+b]^-1 ) + 0.

M xxx = + [b-a]^-1
  * ( [x+a]^-1 - [x+b]^-1 ) + 0.

*yep

B [b-a]

*end
```

Chapter XXII

Summary

The Schoonschip declarations, statements, commands and built-in functions are summarized in this chapter, and collected into functional groups.

1. Comments

Lines with a `C` in column 1 (excluding `Common`) are comment lines. Lines with a blank in column 1 are taken to be continuation lines, including `C` line continuations.

Lines with a `~` in column 1 are ignored.

A `!` signals the start of comments on a line, but is otherwise treated as the end of the line.

2. Program Sections

A `*` in column 1 terminates a program section and tells Schoonschip to start execution or go on to next section.

`*fix`	If present, this must be the first section. It includes declarations that persist until `*end`.
`*yep`	All names, but not expressions in brackets, are kept over a `*yep`.
`*next`	The same as `*yep`, except that local files not mentioned in a `Keep` list are deleted.
`*begin`	All except `fix`'d quantities and common files are deleted.
`*end`	Signals the end of the input.

3. Blocks and Do-Loops

`BLOCK Name{arg1,...}`

Defines block `Name`. Arguments are embedded in the text between `' '`. Call: `Name{brg1,...} Name` must start in column 1. If an argument contains commas it must be enclosed in `{ }`. Nesting: Schoonschip strips away one layer of `{ }`'s at a time.

`ENDBLOCK`

`DO L1=#`$_1$`,#`$_2$`,#`$_3$

The optional `#`$_3$ is taken to be 1, if absent. In-text argument `L1` must be between `' '`. Forced exit from a do-loop at a `*` line is controlled by the `Dostop` flag, which is off by default. See the `Dostop` statement and substition command.

`ENDDO`

4. Declarations

`A a1,a2=`x`, ...`	Algebraic symbol list. If x is a number, then powers of `a2` greater than or equal to x are set to zero. Alternatively x may be `c`, `r` or `i`, denoting reality properties. The default is `r`. Constructions such as `a2=3=c` are allowed.
`Anti TX`	Define character table `TX` to contain the default antiparticle list. May appear in a `*fix` section.
`B b1,b2,...`	List of symbols to be factored out in the output. Symbols, vectors, dot products, vector components are legal. If a vector name is mentioned, all dot products and vector components referring to that vector are factored out. `B Excl,...` option for a list of the *only* symbols and functions to be factored out.
`D Name(n,...)=`xpr_1`,`xpr_2`,...`	Data expressions. Each expression must be a single term.
`F f1,f2=`x`,...`	Function list. Reality properties are specified by x, which may be `i`,`c`,`r` or `u`. The default is `r`.

`I i1,i2=`*d*`,...`	Index list. The dimension *d* may be a number or a single character algebraic variable, previously defined in an `A` list.
`Sum i1,i2,...`	Summation index list. These are not related to the summation indices of the `DS` function, but to indices that appear twice according to the Einstein summation convention. Indices in this list must have been defined before.
`T Name(n,...)=sy1,sy2,...`	Table.
`T Name:  "X=1:1,...`	Character table.
`V p1,p2=z,...`	Vector list. Names must be 1 or 2 characters. The optional `=z` implies that `p2` is to be set to zero as soon as it occurs, whether as a vector component or in a dot product.
`X Name(arg1,...)=`*xpr*	`X` expression.
`Z Name(arg1,...)=`*xpr*	`Z` expression. If `arg1` is a number or simple numerical expression, `Name` is a subscripted file. The number must be in the range 0 to 32,767. Here a simple numerical expression may contain numbers, or block or do-loop arguments that become numbers, separated by `+`, `-`, `*` or `/`.

5. Statements

Blocks:

`BDELETE name1,name2,...`	Delete blocks `name1,...` .

Do-Loops:

`Dostop on`	Sets the `Dostop` flag on, to force exit from a do-loop at a `*` line. `Dostop off` clears the flag to its default off condition. See the `Dostop` substitution command, and Chapter III, *Input Facilities*. Tricky.

Numbers:

`Digits #`	Set the number of digits to be printed for floating point numbers.

`Overflow off`	Suppress short number overflow error trapping. Reactivate with `Overflow on`.
`Precision #`	Set the precision to `#` digits for comparison and cancellation of numbers in various situations. Default: 22 digits.
`Rationalize #`	Set the precision to which a number and its attempted rationalization must agree to `#` digits. The default is to attempt rationialization with agreement to 22 digits. To turn off rationalization, use 0 for `#`.
`Round on`	Set conversion of numerical expressions in function arguments from the default truncation mode to round to the nearest integer. `Round off` reinstates the truncation default.

Printing:

`P` *options* The first 2 characters are significant in the following options:

```
brackets
heads
input        ninput
lists        nlists
output       noutput
statistics   nstatistics
```

`Lprinter`	Print output lines up to 130 characters.
`Screen`	Print output lines up to 80 characters.

Z file manipulation:

In the name lists for these statements, the name of an indexed file may appear without an index. In such a case the statement applies to all files having that name.

`Common name1,name2(5),name3,...`	Declare common files.
`Delete name1,name2(5),name3,...`	Delete files.
`Keep name1,name2(5),name3,...`	Keep local files over a `*next`.
`Names name1,name2(5),name3,...`	Declare the name lists for quantities in common files, which have been read previously from external disk files by the `Enter` statement.
`Nprint name1,name2(5),name3,...`	Do not print `Z` files.
`Print name1,name2(5),name3,...`	Print `Z` files.

Common file input/output:

`Enter cfile`	Enter common files contained in the disk file `cfile`. Must be in the first section, terminated by `*fix`. There may be several `Enter` statements in that section.
`Write cfile`	Write common files to the disk file `cfile`. Must be isolated in its own section, not a `*yep` section.

External file reading:

`Read fname`	Start reading from external disk file `fname`. No `Read` statements are allowed in the file `fname`.
`End`	Switch back to normal input and close the external file. A subsequent read of the same file starts again at the beginning.

Fortran output:

`Fortran #,Brackets, Compression,.,A`	Set options for the output of Fortran statements. See Chapter XVIII, *Statements*, for details.
`Punch name1,name2(5),...`	Write to `tape8` in Fortran compatible form.

Large problem management:

`Directory` $\#_1$`,pre1,...`	Allocate subfile disk storage across file system directories and devices. If present, must come first in a Schoonschip program in a `*fix` section.
`Dryrun #`	Run the program and print a maximum of `#` terms per result, but otherwise produce no output files. Print an estimate of the time and disk space required.
`Freeze cfile`	For the common file `cfile`, work on only the factors mentioned in a `B` list; and refer to the remaining factors through the `DF` function. See the command `Expand` for unfreezing.
`Large #`	Specify that `#` (between 1 and 5) disk files are to be used for output sorting and merging.
`Maximum #`	Specify the maximum subfile size, in bytes, for sorting and merging. Default: 110,000.
`Nshow`	Turn off `Showlength`.

`Outlimit #`	Change output limit in bytes for total disk space. Default: 500,000.
`Showlength`	Print the size and name of the last subfile used in sorting, as it is completed.
`Traditional`	Use the sorting method from earlier versions of Schoonschip, which does not utilize fractured files.

Miscellaneous statements:

`Beep` *`options`*	Issue beeps at the terminal. All beeps cleared if no option. Options: `#,t` Issue `#` beeps at termination. `#,*` Issue `#` beeps after a section. `#,e` Issue `#` beeps in case of error.
`External fname`	Load user supplied machine executable file into memory. See command `Extern`.
`M xxx = ...`	Compare the expression `xxx = ...` to a preceding line of `R` input. Followed by `*yep`.
`Oldnew a1=b1, a2=...`	Change name `a1` into `b1`, etc.
`Progress`	Print dots to the terminal to show progress in processing terms. Same as `Progress on`. Switched off by `Progress off`.
`R xxx = ...`	Transform text output `xxx = ...` from a Schoonschip problem directly into input for `*yep`. Useful for long expressions that would normally overflow the input space.
`Silence`	Turn off a mode on certain PC's that waits for user response.
`Synonym a1=b1`	Every occurrence of `a1` between single quotes is replaced by `b1`. The end of the range of validity of a set of synonyms is defined by a `Synonym` statement with no arguments. Up to 16 synonyms allowed.

6. Substitutions

The general structure of a substitution involves several elements from various groups, separated by commas. Substitutions are often used with the conditionals `IF`, `AND`, `OR` and their `NOT` versions, which we summarize in Section 7.

<u>`Id` and `Al`</u>:

The first group is `Id` or `Al` itself (possibly with `;` instead of `Al,`), followed by an optional number:

```
Id,   Al,   ;   Id,#,   Al,#,   ;#,
```

The number specifies on how many levels the substitution or command is to be applied.

`Id` Substitution or command on the next free Level.

`Al` Substitution or command on the same level. On a line starting with `Id`, subsequent `;`'s are read as `Al,`.

<u>Keywords</u>

The next group may contain one item, namely a keyword. Keywords are only for substitutions involving patterns, not commands. Valid keywords are:

```
Multi   Funct   Dotpr   Ainbe   Adiso   Always   Once
```

This group may be empty.

<u>Patterns and Commands</u>:

The next group either defines a pattern followed by an `=` sign, or is a command followed by arguments separated by commas (if any). Patterns are products of factors, with dummies designated by a `~` following the symbol. Not all factors can be dummies. See Chapter X, *Patterns*.

The available commands and their formats are:

`Absol` Force coefficient positive.

`Addfa` Add factor.

`Id,Addfa,`*expression*

`All` Collect vectors `p` into function `F`. Indices that are created may be assigned a dimension.

First syntax:
`Id,All,p,N,F`
possibly followed by the character arguments `"F_`, `"D_`, and/or `"V_`, which inhibit function argument, dot product or single vector inspection (example: `Id,All,p,N,F,"F_`).

Second syntax:
`Id,All,F1,N,F`

Collect all vector arguments of function `F1` into `F`. Substitute created indices as arguments of `F1`, and also as additional arguments of `F`.

See Chapter XII, *Substitution Commands*, to avoid a problem with renamed, “split” repeated indices and `Freeze`.

`Anti`	Check function arguments for characters, and remove any trailing underscore (`_`) present if the character is mentioned in the antiparticle character table.

```
Id,Anti,TA
```

`Asymm`	Rearrange the arguments of an antisymmetric function.

```
Id,Asymm,f1,2,3,4,f2,f3,4,5,...
Id,Asymm,F1:2,3,4:7,8,9:12,13,14:...
Id,Asymm,F1:1-3:7-9:12-14:...
```

`Beep`	Generate a beep on the terminal.

```
Id,Beep
```

`Coef`	Inspect numerical coefficient.

`IF Coef,`*option*

Options:

`pl`	plus
`ng`	negative
`eq,#`	equal
`lt,#`	less than
`gt,#`	greater than
`ib,#,#`	in-between

`Commu`	Rearrange commuting functions.

```
Id,Commu,f1,f2,f3,...
```

`Compo`	Compose names.

`Id,Compo,<`*options*`>,f1,f2,<`*options*`>,f3,f4,...`

Options: `AFIVX` with `AFV` possibly twice. `"S_` suppresses symmetrization of the name. A character table may be specified, to be used if characters with an underscore appear.

`Count`	Count certain occurrences with certain weights.

`Id,Count,xxx,arg1,`$\#_1$`,arg2,`$\#_2$`,...`

If `xxx` is a number and the count is `#` then the term is kept if `#` $\geq$ `xxx`.

If `xxx` is a symbol or vector component or dot product then `xxx^#` is made.

If `xxx` is a function or `X` or `D` expression then `xxx(#)` is made. In that case a multiple count can also be done:

`Id,Count,Fx,arg1,#,...,# : arg2,#,...,# : arg3,#,...`

Now `Fx(`$\#_1$`,`$\#_2$`,`$\#_3$`,...)` is made with $\#_1$ determined by the arguments up to the first colon, $\#_2$ by the arguments between the first and second colon, etc.

`Cyclic` Function is invariant under cyclic permutation of its arguments.

`Id,Cyclic,f1,2,3,4,f2,f3,4,5,...`

`Dostop` Set or clear `Dostop` flag. Tricky, see Chapter III, *Input Facilities*.

```
Id,Dostop,on
Id,Dostop,off
```

Test `Dostop` flag:

```
IF Dostop
Id,Dostop
```

`Epfred` Reduce products of `Epf` functions.

`Id,Epfred`

`Even` Function is even in the signs of its arguments.

`Id,Even,f1,2,3,4,f2,f3,4,5,...`

`Expand` Expand frozen subfiles (appearing as `DF(fname,#)`).

`Id,Expand,fname`

`Extern` Jump to user defined routine.

`Id,Extern,arg1,arg2,...`

`Iall` Inverse of `All`.

```
Id,Iall,p,F
Id,Iall,F
```

`Numer` Insert numerical values.

`Id,Numer,arg1,`$\#_1$`,arg2,`$\#_2$`,...`

`Odd` Function is odd in the signs of its arguments.

`Id,Odd,f1,2,3,4,f2,f3,4,5,...`

`Order` Order functions on the basis of their first two arguments.

`Id,Order,F1,F2,...`

`Print` Print message a maximum of `#` times.

`Id,Print,#,`*message*

`Ratio`	Rationalize. `Id,Ratio,xpa,xpb,bma`
`Stats`	Count passage in one of four counters (0-3). The result is printed at the end. `Id,Stats,#`

`Symme`	Rearrange arguments of symmetric function.
	`Id,Symme,f1,2,3,4,f2,f3,4,5,...`
	`Id,Symme,F1:2,3,4:7,8,9:12,13,14:...`
	`Id,Symme,F1:1-3:7-9:12-14:...`

Particle Physics Commands:

Three commands are specialized for problems of relativistic quantum mechanics. See Chapter XV, *Particle Physics*, and Chapter XVII, *Gamma Algebra.*

`Gammas` Collect, reduce and/or take the trace of products of `G`'s.

`Id,Gammas,`*options*`,L1,L2-L3:`*options*`,`
`    Trace,L3,L4-L5:`*options*`, ...`

Options are designated by characters:

`"C`	do collection and normalization only
`"N`	normalize
`"A`	anomalous trace
`"S`	do sum splitting
`"U`	unify strings
`"I`	do index pair reduction
`"i`	do index pairs in `"_`-type strings
`"P`	do `P` tricks
`"W`	do final work

Each option except `"C` may be followed by an underscore, which switches it off. The `"C` option when not combined with anything else gives normalization, but with no specialization to four dimensions (essential if there are vectors that are not four-dimensional).

Default: `"N`, `"A_`, `"S`, `"U_`, `"I`, `"i`, `"P`, `"W`

`Spin`	Replace pairs of `Ug`, `Ubg` by spin summation expression. The spinors may be defined by loop indices or vectors. `Id,Spin,L1,p,...`
`Ndotpr`	Create special dot products. Closely related to `Gammas` command. `Id,Ndotpr,F1`

7. Conditional Substitutions

Conditionals are used with substitution patterns and commands according to the following structure (square brackets denote optional elements and are not included in the syntax):

> `IF` [`NOT`] *pattern* [`=` *expression*] *or command*
> [`AND`] [`NOT`] *pattern* [`=` *expression*] *or command*
> [`OR`] [`NOT`] *pattern* [`=` *expression*] *or command*
> *substitutions*
> [`ELSE`
> *substitutions*]
> `ENDIF`

The above lines all begin in column 1, with the exception of lines with a dot (`.`) in column 1, which can be used for indentation to show nesting. Conditional structures may be nested up to a depth of 8.

Substitutions are not allowed between `IF`, `AND` and `OR` statements when there is an `ELSE`; furthermore `AND` and `OR` statements are not allowed after an `ELSE`, except as part of a new, nested structure. See Chapter XI, *Conditional Substitutions*, for the filtering rules for successive `AND`/`OR` statements.

8. Built-In Functions

General purpose:

`D` Delta function.

`DD` Internal purposes (used to represent `X` and `D` expressions and files).

`DS` Summation function.

Format:
`DS(J,`j_1`,`j_2`,(`xpr_1`),(`xpr_2`))`

Here xpr_2 is numeric, optional. The sum over the given range is constructed. The numerical coefficient is found by recursion, using xpr_2. The first coefficient is 1.

Example:
`e^x = DS(J,0,20,(x^J),(1/J)) + O(x^21).`

Caution: summation symbols such as `J` may be used only once in a given expression.

Character summation:
`DS(C1;C2;...;Sym;C3;C4;TX,...,2:4,7,(`*xpr*`))`

`Epf` Antisymmetric Weyl tensor. May have any number of arguments.

Numerical functions:

These are used in expressions of numerical type, such as expressions occurring as exponents, including negative exponents; but they may not used in simple numerical expressions, such as `Z` expression indices:

`DB(`n`) = `$n!$ `DB(`n`,`m`) = `$n!/\{m!(n-m)!\}$	`DB` with two arguments is word-bound, fails if $n = 19$, $m = 8, 9, \ldots$
`DT(`n_1`,`n_2`,...)`	1 if all arguments positive or 0, else 0.
`DK(`n_1`,`n_2`)`	1 if $n_1 = n_2$, else 0.
`DP(`n_1`,`n_2`,...)`	1 if permutation to ascending order is even, else 0. Normal order: $-127, \ldots, 0, \ldots, +127$. Vanishes if two arguments are equal.

`DF(xfile,#)`
`DF(xfile,#`$_1$`,#`$_2$`)`
	The latter case is equivalent to the former with `# = 100*#`$_1$` + #`$_2$.

`DC(`n_1`,`n_2`,...)`
`DC("X,...)`
	1 if sum of all arguments is 0, else 0.
	`"X` is an option character which may be `"C`, `"F`, `"T`, with or without a trailing underscore.

Particle physics:

Dirac matrices and spinors (use `-m` for antiparticle spinors):

```
G(L,mu)                      Ug(L,m,p)
Gi(L)                        Ug(L,mu,m,p)
G5(L)                        Ubg(L,m,p)
G6(L) = Gi(L) + G5(L)        Ubg(L,mu,m,p)
G7(L) = Gi(L) - G5(L)
```

New notation:

```
G(i1,i2,mu,nu,...,G6,...)
Gi(i1,i2)   G5(i1,i2)   G6(i1,i2)   G7(i1,i2)
```

Internally generated `G`-strings:

`G(L1,"s,"4,mu,nu,...,G5,...)`	General 4- and `N`-dim'l mixture.
`G(L1,"t,"4,mu,nu,...,G5,...)`	General 4- and `N`-dim'l mixture.
`G(L1,"S,"X,mu,nu,...)`	4-dim'l. `"X` may be `"4`, `"5`, `"6` or `"7`.
`G(L1,"T,"X,mu,nu,...)`	4-dim'l. `"X` may be `"4`, `"5`, `"6` or `"7`.
`G(L1,"s,"_,mu,nu,...)`	`N`-4 dim'l.
`G(L1,"t,"_,mu,nu,...)`	`N`-4 dim'l.

9. Conditional Input

The conditional input commands implement the selective reading of Schoonschip program text input, in the spirit of conditional assembler directives. They are used with the 10 conditional input parameters, `Sw0`, ..., `Sw9`. See Chapter XIII, *Conditional Input* for more details.

`Goto` ***xpr***	Skip unconditionally until a program line with `@` in column 1 is found, followed by the number which is the value of *xpr*, or an `@` in column 1 of an otherwise empty line. The value of *xpr* must be `0`, ..., `99`.
`Gotoif` ***xpr***`,` xpr_1 *?* xpr_2	Skip as in `Goto` if the condition xpr_1 *?* xpr_2 evaluates to true, where *?* may be `>`, `<` or `=`, otherwise continue with the next program line. The values of xpr_1 and xpr_2 must be signed, 16-bit numbers.
`Gotoifn` ***xpr***`,` xpr_1 *?* xpr_2	The same as `Gotoif`, except that skipping occurs when xpr_1 *?* xpr_2 evaluates to false.
`Set _SwX=`***xpr***	Set `_SwX` to the value of the byte-sized, signed numerical expression *xpr*, where `X` = `0`, ..., `9`.

Appendix A

Command Line Options

The standard usage is:

```
Schip file1 file2 options
```

Schoonschip input is read from `file1`, and the output is written to `file2`. Any Fortran compatible output (`Punch` statement) is written to `tape8`. If `file2` is absent, then the output is written to the standard output, usually the screen. If in addition `file1` is absent, then the input is read from the standard input, usually the keyboard. When using the keyboard for input, remember that `*end` terminates a run. Errors also terminate a run. Schoonschip normally echos input lines back to the screen.

The input, output and punch files can also be designated with the notation `i=`, `o=` or `f=` (the `f` stands for *Fortran* compatible); upper case is accepted as well:

```
Schip i=file1 o=file2 f=file3
```

This notation cannot be combined with the first method.

There are a number of options, any of which may be absent. Certain of these are specified by a leading `-` sign. They are (upper case may also be used):

`s` For versions of Schoonschip running on a PC with bitmapped screen. The "`Ready`" line and the keyboard input request are suppressed if `s` (for *silence*) is specified.

`b` Certain Fortran compilers do not accept negative exponents without brackets (for example, `A**-3`). If `b` is specified, brackets are written. The default is no brackets.

`p`	Show progress by printing a dot (`.`) to the standard output every 128 terms.
`.`	Place a `.` after every integer in the output. Some Fortran compilers insist on this.

Example:

```
Schip file1 file2 -b.p
```

Options `b`, `.` and `p` apply.

These options may also be selected from within a Schoonschip program by means of the statements

```
Silence
Fortran options
Progress on
Progress off
```

These statements are discussed in Chapter XVIII, *Statements*.

Another option causes noise (CTRL G) to be sent to the terminal upon completion of the program. This can be specified by `b=#`, where `#` is the number of beeps the terminal is to receive.

The option `S=#`, where `#` is a number in the range −128 to 127 inclusive, sets the initial value of the parameter `_Sw0` to the value `#`.

Example:

```
Schip file1 file2 b=10 s=19
```

Upon completion 10 beeps will be sent to the terminal. The initial value of `_Sw0` is 19.

It is important that one can now specify Schoonschip's memory usage. There are two large storage areas used during execution, called the input space and the output space. The input space contains expressions, substitutions, etc. The output space is where the final result is collected. Performance for large expressions is very sensitive to the size of the output store—the larger the better. Because of its heritage from the old CDC days, Schoonschip has always been very frugal in memory usage. The built-in sizes of the input and output spaces used to be 20 kilobytes and 50 kilobytes, respectively. Given that nowadays few machines have less than 1 megabyte of memory that seems far too restrictive.

Versions since January 1, 1989 have been more expansive. To begin with, about 80 kilobytes are used separately for buffers, etc. This used to be much less. Default sizes for the input and output spaces are machine dependent, but typically they are 100 kilobytes and 250 kilobytes. The total memory usage (excluding Schoonschip itself, which is about 100 kilobytes) comes to about 450 kilobytes.

The input and output sizes may now be specified on the command line. For that purpose one uses the notation `IS=#` and `OS=#` (lower case also accepted) where the `#`'s stand for numbers specifying the size in kilobytes. Thus the old situation is obtained with this command line:

```
Schip file1 file2 IS=20 OS=50
```

Finally a number `#` preceded by a `+` sign can be specified on the command line. Schoonschip then skips the first `#` problems in the input file. Every occurrence of a `*end` line counts as one problem. Example:

```
Schip +5 Varia Out
```

Problem 5 in file `Varia` is executed.

Some other examples:

```
Schip Infil Outfil -bp
```

The file `Infil` is taken to contain the input, output is written to `Outfil`. Dots are printed when starting a problem, and then after every 128 terms. Brackets are placed around negative exponents.

```
Schip +1 i=Varia o=Vario -p
```

Problem 1 of `Varia` is executed, the output is written to `Vario`, and meanwhile dots are written to the screen show progress.

Appendix B

Compatibility

Programs that worked with older M68000 versions of Schoonschip will still work provided square brackets `[ ]` are taken out and replaced, for example, by curly brackets `{ }`. In addition there is a slight change in `Integ` (evaluation of numerical expressions as function arguments). It now truncates, rather then rounding to the nearest integer. The statement `Round on` may be used to reinstate rounding (CDC versions of Schoonschip truncated).

The command `Trick` is now called `Gammas`, but is still accepted in its old form. The block delete command `DELETE` (all upper case) has been replaced by `BDELETE` (case insensitive, only the first four characters significant). The rarely used statement `Printer` has been replaced by `Lprint`.

Programs that executed on previous CDC versions of Schoonschip will also run with this version after minor, mainly cosmetic modifications. The major differences are in the use of lower case characters in this version, and different notations for block and do-loop arguments:

- Block arguments in the text of the block must be embedded in quotes. The same holds for do-loop arguments. The latter can now take negative values in all cases.

  The use of `{ }` is obligatory. Block arguments in a call may now have commas provided they are enclosed in `{ }`.

  The word `COPY` is no longer needed when calling a block.

  Block names may be at most 6 characters in length.
- The complex variable `I` is now written as `i`. The statement `Oldnew i=I` can be used to take care of this.
- The `S` list is now the `A` list. `S` is, however, still accepted.
- `ID` and `AL` are now `Id` and `Al`. Both are accepted.

- `GI` is now `Gi`.
- `CONJG` and `INTEG` are now `Conjg` and `Integ`.
- Various keywords and commands sometimes have slightly different spellings—`ODDXX` is now `Odd`.
- The `Print` options have a different format. For example, `PRINT INPUT` is now `P input`.
- Dummies are now characterized by a trailing `~` instead of a `+`.
- An exponent may now also be denoted by `^` instead of `**`.
- The special function `DX` is no longer implemented. It is used internally by the `Gammas` command.
- Characters as function arguments are denoted by `"X` instead of `=X`.

Index

Many Schoonschip program elements are listed in two or three different ways in this index. Generally there is a list of subentries by symbolic name, with page numbers only, under a general category such as “statements”. Each of these is also usually listed separately by symbolic name, and by text name when one exists, with more informative subentries.