Contents

0. Preliminaries

This and the following sections provide a pedagogical introduction to the basic features of SMP. Knowledge of these features suffices for many applications of SMP. Reference is made when appropriate to the "SMP Reference Manual", which gives a complete and systematic description of the facilities in SMP.

This primer assumes no prior experience with computer systems. A glossary of some technical terms used is given as an appendix.

As with all computer systems, SMP is most effectively learned through use. The reader is therefore strongly encouraged to experiment on an actual SMP system as he reads this primer.

Many superficial features of SMP differ from one installation to another. The details pertinent to a particular installation should be given in the "Implementation Notes".

First find a suitable terminal, connect it to the computer, and log in (see the Implementation Notes). A video terminal will probably be much more convenient than a printing one; SMP always keeps a record of your work. On video terminals there is usually a "cursor", often a square or an underscore, which indicates the position at which the next character will appear. Under normal circumstances, any character typed on the terminal should appear at this position*. (If this fails to happen, check that the terminal is connected to the computer, and that it is not on "local". Terminals sometimes become "locked": switching power off momentarily may "unlock" them. If each character is printed twice, switch the terminal or telephone modem from "half duplex" to "full duplex".) Next identify the characters on the terminal referred to in the Implementation Notes. "Control" characters are typed in direct analogy with upper-case "shifted" characters: hold down the key marked "CTRL" and press the required character.

On most systems, no typed input is processed until a ⊂*newline*⊃ is entered. Except for very complicated operations, the computer should respond to any input within a few seconds; if an expected response is not forthcoming, the computer is probably waiting for ⊂*newline*⊃. (An extra ⊂*newline*⊃ is never detrimental.) Before the ⊂*newline*⊃ is entered, text on a line may be deleted and retyped. ⊂*character delete*⊃ causes the last character typed to be discarded; typing two ⊂*character delete*⊃ ´s discards the last two characters, and so on. When a character is discarded, the cursor usually backspaces over its position (however, on some systems, the discarded character is reprinted). A replacement character may then be typed. ⊂*line delete*⊃ discards all characters typed so far on the present line.

Characters in examples given below may be different for particular systems. Any such differences are given in the Implementation Notes.

Notice that upper and lower case letters are distinguished in SMP. (If your terminal does not allow lower case letters, the Implementation Notes will give the necessary instructions.)

Now, using the instructions in the Implementation Notes, start an SMP job.

---

* Except when a "password" is being entered.

1. Fundamentals

When SMP is called, it begins by printing*

```
SMP 1
#I[1]::
```

placing the terminal cursor at the position marked □. With this prompt, the SMP job is ready to receive its first input. **#I[1]** is the name to be assigned to this first input line. As a first example of an input expression, type **2+5** followed by ⊂*newline*⊃.

```
#I[1]::  2+5
#O[1]:   7
#I[2]::
```

SMP read the input expression **2+5**, simplified it, and printed the result **7** as the first output expression **#O[1]**. It is now ready to receive a second line of input.

SMP simplifies any input line to which it can assign a unique meaning. If an input line is ambiguous or meaningless, SMP enters "edit mode" [1.7], allowing the line to be modified. (Throughout this primer, references to sections of the Reference Manual and Summary are given in square brackets; references to sections of the Primer are given as "sect. *n*".) Typing two ⊂*newline*⊃´s in succession exits the editor, discards the original line and causes the input prompt to be reissued. (If additional editor commands have unwittingly been entered, it may be necessary to type **\q** to exit the editor.)

```
#I[1]::  2++5
+ unexpected
         2++5
<edit>
         2++5
<edit>
#I[1]::
```

In this example, the first □ marks the position of the cursor on entry to edit mode, and the second □ its position after exit from edit mode achieved by input of two ⊂*newline*⊃´s.

SMP does assign a unique meaning to many unintended input lines; in some cases the input may be output again with little or no modification, in others, surprising results may be obtained. In most such cases, the intended input may simply be entered as the next input line.

```
#I[1]::  1..5+3.2
#O[1]:   [1,2,3,4,5,6,7,8]
#I[2]::  1.5+3.2
#O[2]:   4.7
```

At all stages in an SMP job, a ⊂*quit interrupt*⊃ terminates the computation or output, and issues a new input prompt.

---

* On some systems, there may be a wait of several seconds between the request for SMP and this response.

SMP is usually very taciturn: it prints essentially no messages. Information on a particular object or topic may be obtained by typing **?***name*.

The appendix to this primer lists a variety of common difficulties, and should be consulted if unexpected behaviour occurs.

An SMP job is usually terminated by typing ⊂*input termination character*⊃ after an input prompt*. All input and output in an SMP job is saved in an external file for possible later use, as discussed in sect. 4.

Most mathematical operations and functions are represented in SMP in close analogy with their written forms. The standard arithmetic operations are typed as:

*x***+***y*	*x* plus *y*
*x***-***y*	*x* minus *y*
*x****** *y* or *x* ⊂*space*⊃ *y*	*x* multiplied by *y*
*x***/***y*	*x* divided by *y*
*x***^***y*	*x* raised to the power *y*.

Here *x* and *y* stand for numbers or other SMP expressions. Additional ⊂*space*⊃´s may always be typed on either side of arithmetic operators. (Note that *x*******y* represents the "outer product" of *x* and *y*, not *x* raised to the power *y*.)

Arithmetic operations are performed in the conventional order: parenthesized expressions are evaluated first, followed by **^**,**/**,*****,**-**,**+**. ⊂*space*⊃´s or parentheses usually suffice to indicate multiplication: no explicit ***** need be typed.

Divisions group to the left (so that **4/3/2** means **(4/3)/2**) but powers group to the right (**4^3^2** means **4^(3^2)**) [2.10].

```
#I[1]:: 1+2+7

#O[1]:  10

#I[2]:: 1 +7-15

#O[2]:  -7

#I[3]:: 2*3*4

#O[3]:  24

#I[4]:: 2 3 4

#O[4]:  24

#I[5]:: -23 4

#O[5]:  -92

#I[6]:: 1/2+7/3

#O[6]:  17/6

#I[7]:: 242/12

#O[7]:  121/6
```

---

* A computation may usually be terminated at an intermediate stage by typing ⊂*quit interrupt*⊃ (without an input prompt); the job may be terminated by ⊂*break interrupt*⊃ followed by **Exit[]** (after the prompt **%I[1]::**).

```
#I[8]::  4^6

#O[8]:   4096

#I[9]::  (2+3)^2

#O[9]:   25

#I[10]::  2+3^2

#O[10]:   11

#I[11]::  2+3*4

#O[11]:   14

#I[12]::  3*4+2

#O[12]:   14

#I[13]::  3 4+ 2

#O[13]:   14

#I[14]::  3*(4+2)

#O[14]:   18

#I[15]::  3(4+2)

#O[15]:   18

#I[16]::  (1+2)(3+4) (5-6+7)

#O[16]:   126

#I[17]::  2/3 4

#O[17]:   8/3

#I[18]::  2/(3 4)

#O[18]:   1/6

#I[19]::  4^3^2

#O[19]:   262144

#I[20]::  (4^3)^2

#O[20]:   4096

#I[21]::  27^(1/3)

#O[21]:   3

#I[22]::  27^(1/2)

           1/2
#O[22]:   27
```

In lines 6 and 7 above, answers were given as rational numbers. On line 22, the answer could not be given as a simple rational number, and so was left in a symbolic form. In such cases, a decimal number result may be obtained using the SMP "projection" **N**. For any expression *expr*, **N[***expr***]** yields a real or complex numerical result if this is possible. Note the use of square brackets around *expr*, to be distinguished from the parentheses used to indicate grouping in expressions.

```
#I[1]::  1/2+7/3
#O[1]:   17/6
#I[2]::  N[1/2+7/3]
#O[2]:   2.83333
#I[3]::  27^(1/2)
           1/2
#O[3]:   27
#I[4]::  N[27^(1/2)]
#O[4]:   5.19615
#I[5]::  N[27^0.5]
#O[5]:   5.19615
```

SMP treats a large number of mathematical functions, including all common special functions of mathematical physics*. A list of the functions is given in [8]. Any of the functions listed may be evaluated numerically† by use of **N**. Note that all "system-defined" or "built-in" functions in SMP have names which begin with a capital letter. Their "arguments" are enclosed in square brackets (just as for **N**), and separated by commas.

```
#I[1]::  Exp[2]
#O[1]:   Exp[2]
#I[2]::  N[Exp[2]]
#O[2]:   7.38906
#I[3]::  N[BesK[2,4.56]]
#O[3]:   -0.816524
#I[4]::  N[BesJ[0.2,1.5] Sin[0.2^3]]
#O[4]:   0.00483492
```

In line 3, for example, **BesK[***n*,*z***]** represents the modified Bessel function $K_n(z)$.

At all stages in an SMP job, the symbol **%** stands for the latest output line generated.

```
#I[1]::  1/2+7/3
#O[1]:   17/6
#I[2]::  %
#O[2]:   17/6
#I[3]::  %^2
#O[3]:   289/36
```

---

* Sect. 8 describes the procedure for defining values and characteristics of further functions.

† When branch cuts are present, all functions are evaluated on their principal Riemann sheets.

```
#I[4]::  N[%]

#O[4]:   8.02778

#I[5]::  %+%^3

                                3
#O[5]:   289/36 + 289/36
```

SMP manipulates not only numbers, but also expressions containing symbolic parameters or "symbols". Symbols may be used to represent quantities whose numerical value is undetermined. Mathematical simplifications performed on expressions hold for any possible values of the symbols which appear in them.

Any sequence of letters and numbers (not starting with a number) may be used to denote a symbol. As throughout SMP, upper and lower case letters are distinguished. System-defined symbols such as **N**, **Pi** (see below), and **Exp** always begin with a capital letter. To avoid confusion, symbols introduced by the user should therefore begin with a lower case letter.

```
#I[1]::  x

#O[1]:   x

#I[2]::  x+x

#O[2]:   2x

#I[3]::  %-3x

#O[3]:   -x

#I[4]::  x1+x2-3x1/4

#O[4]:   x1/4 + x2

#I[5]::  %^2-4%

                                      2
#O[5]:   -x1 - 4x2 + (x1/4 + x2)

#I[6]::  (x+a)(x+b)-(x+y)^(a+b)/(2a+b)

                  a + b
         - (x + y)
#O[6]:   -------------- + (a + x) (b + x)
             2a + b

#I[7]::  (Exp[x^2]-1)Sin[x phi/4]

                     2        phi x
#O[7]:   (-1 + Exp[x ]) Sin[-----]
                              4
```

Notice that in, for example, line 3, **3x** denotes **3*x**. However, **x2** in line 4 is a single symbol.

One use of symbols is to represent mathematical constants, for which numerical values are defined [8.4].

```
#I[1]::  Pi^2-9

                 2
#O[1]:   -9 + Pi

#I[2]::  N[%]

#O[2]:   0.869604
```

```
#I[3]::  Sin[Pi/2]

#O[3]:   1

#I[4]::  Sin[45Deg]

#O[4]:   Sin[45Deg]

#I[5]::  N[%]

#O[5]:   0.707107

#I[6]::  Exp[Euler]-1

#O[6]:   -1 + Exp[Euler]

#I[7]::  N[%]

#O[7]:   0.781072
```

The argument of the trigonometric function **Sin** in line 4 is in radians; **Deg** is a constant with value **Pi/180**. **Euler** is the Euler-Mascheroni constant γ=0.5772....

Some mathematical operations which may be performed on symbolic expressions are described in [9]. Examples are

Ex[*expr***]**	"Expand" *expr* by using distributive rules for various functions.
Fac[*expr***]**	Factor *expr*.
D[*expr***,***var***]**	Form the partial derivative of *expr* with respect to *var*.
Int[*expr***,***var***]**	Form the integral of *expr* with respect to *var*.

```
#I[1]::  (x+1)(x+a)(x+b)

#O[1]:   (1 + x) (a + x) (b + x)

#I[2]::  Ex[%]

                          2              2              2    3
#O[2]:   a b + a x + a x  + b x + b x  + a b x + x  + x

#I[3]::  Fac[%]

#O[3]:   (1 + x) (a + x) (b + x)

#I[4]::  Sin[x]/(1+x+x^2)

           Sin[x]
#O[4]:   ----------
                 2
         1 + x + x

#I[5]::  D[%,x]

                  (1 + 2x) Sin[x]
         Cos[x] - ---------------
                             2
                    1 + x + x
#O[5]:   -----------------------
                          2
                 1 + x + x
```

```
#I[6]:  Ex[%]

          Cos[x]              2x Sin[x]
#O[6]:  ---------- - ------------------------
                2                 2      3    4
        1 + x + x    1 + 2x + 3 x  + 2 x  + x

                        Sin[x]
              - -------------------------
                           2      3    4
                  1 + 2x + 3 x  + 2 x  + x

#I[7]:  x/(1-4x+x^2)

             x
#O[7]:  -----------
                2
        1 - 4x + x

#I[8]:  Int[%,x]

                          1/2                          1/2
             -4 + 2x - 12          -4 + 2#1 - 12
        2Log[---------------]  2Log[---------------]
                          1/2                          1/2                    2
             -4 + 2x + 12          -4 + 2#1 + 12       Log[1 - 4x + x ]
#O[8]:  --------------------- - --------------------- + ----------------
                  1/2                   1/2                    2
                12                    12

                                 2
                Log[1 - 4#1 + #1 ]
              - ------------------
                         2
```

In line 8, the symbol **#1** was introduced as a constant of integration.

A symbol may be assigned an expression as a value. *symb*:*expr* assigns the expression *expr* to be the value of the symbol *symb* [3.2]. After this assignment, the symbol *symb* becomes essentially just a short notation for its value *expr*. Whenever *symb* appears, it is replaced by *expr*.

```
#I[1]:  x

#O[1]:  x

#I[2]:  x:a+b

#O[2]:  a + b

#I[3]:  x

#O[3]:  a + b

#I[4]:  x^2 + x

                      2
#O[4]:  a + b + (a + b)

#I[5]:  y:x+1

#O[5]:  1 + a + b

#I[6]:  x-y

#O[6]:  -1
```

```
#I[7]:: y
#O[7]:  1 + a + b
#I[8]:: x
#O[8]:  a + b
```

Note that *symb*`:`*expr* performs the specified assignment, and then yields the result *expr*. Thus the output from the assignment on line 2 was the assigned value `a+b`.

`a:b:c` assigns the value `c` to both `a` and `b`.

If an assignment is made for a symbol which already carries a value, the newly assigned value replaces the old one.

```
#I[1]:: x:2
#O[1]:  2
#I[2]:: x^2
#O[2]:  4
#I[3]:: x:3
#O[3]:  3
#I[4]:: x^2
#O[4]:  9
```

Once an assignment for a symbol, say `x`, has been made, it continues to be used throughout the SMP job. A common source of unexpected results is the use of symbols for which a value has been assigned much earlier in the job. *symb*`:` removes any value assigned to *symb*.

```
#I[1]:: x:2
#O[1]:  2
#I[2]:: x^2+x
#O[2]:  6
#I[3]:: x:
#I[4]:: x^2+x
                2
#O[4]:  x + x
```

Assignment of a value for a symbol causes the symbol to be replaced by that value whenever it appears. Substitutions for a symbol in a particular expression may also be made. `S[`*expr*`,`*symb*`->`*subst*`]` substitutes the value *subst* for the symbol *symb* in the expression *expr*. The "arrow" is typed as `-` followed by `>`.

```
#I[1]:: x^2+x
                2
#O[1]:  x + x
```

```
#I[2]::  S[%,x->a+b]

                          2
#O[2]:   a + b + (a + b)

#I[3]::  x

#O[3]:   x

#I[4]::  x:3+c

#O[4]:   3 + c

#I[5]::  x^2+x

                          2
#O[5]:   3 + c + (3 + c)

#I[6]::  S[%,c->4]

#O[6]:   56
```

At the beginning of this section we described the procedure for discarding ambiguous or erroneous SMP input lines. Often simple changes in such lines give them a definite meaning. When the editor is entered, the cursor is placed under the position in the line where modification is required. Any characters typed at this stage are used to replace the characters appearing above them in the original input line. Spaces may be used to position the cursor. **#** deletes the character appearing above it. **^**⊂*text*⊃⊂*space*⊃ inserts the sequence of characters ⊂*text*⊃ in the line immediately before the character above the **^**. The text to be inserted is terminated by a space: insert ***** to denote multiplication. The editing commands are performed when a ⊂*newline*⊃ is entered, and the edited line is then printed. If no editing commands are given before a ⊂*newline*⊃ the editing is assumed complete, and the edited line is used as input for SMP.

```
#I[1]::  2++5
+ unexpected
         2++5
<edit>    #
         2+5
<edit>

#O[1]:   7

#I[2]::  t:f]x]
] unexpected
         t:f]x]
<edit>       [
         t:f[x]
<edit>

#O[2]:   f[x]

#I[3]::  (c+d)(a++b+c
+ unexpected
         (c+d)(a++b+c
<edit>           ^x  +1)
         (c+d)(a+x+b+c+1)
<edit>

#O[3]:   (c + d) (1 + a + b + c + x)
```

Notice that **<edit>** is the prompt for input of editor commands.

Further examples

Example 1
Find the factors of $1-x^{12}$.

```
#I[1]:  Fac[1-x^12]

                                  2              2              2
#O[1]:  -(-1 + x) (1 + x) (1 + x ) (1 - x + x ) (1 + x + x )

                        2     4
              * (1 - x  + x )

#I[2]:  Ex[%]

                12
#O[2]:  1 - x
```

Example 2
Find the numerical value of $\Gamma(5.2)$.

```
#I[1]:  N[Gamma[5.2]]

#O[1]:  32.5781
```

Example 3
Verify that $x=1$ is a root of the polynomial $x^3+9x^2+11x-21$.

```
#I[1]:  x^3+9x^2+11x-21

                          2     3
#O[1]:  -21 + 11x + 9 x  + x

#I[2]:  S[%,x->1]

#O[2]:  0
```

Example 4
Find the value of the first derivative of $e^{x^a/(1+x)}$ at the point $x \to 1$.

```
#I[1]:  Exp[x^a/(1+x)]

                 a
                x
#O[1]:  Exp[-----]
               1 + x

#I[2]:  D[%,x]

                 a          a
                x        - x          -1 + a
               Exp[-----] (----- + a x       )
                   1 + x    1 + x
#O[2]:  -------------------------------------
                        1 + x
```

```
#I[3]::  S[%,x->1]

                Exp[1/2] (-1/2 + a)
#O[3]:   -------------------
                      2

#I[4]::   N[%]

#O[4]:   0.824361(-0.5 + a)

#I[5]::  N[D[Exp[x^a/(1+x)],{x,1,1}]]

#O[5]:   0.824361(-0.5 + a)
```

Example 5

Verify that the roots of a quadratic equation are given by the usual formulae.

```
#I[1]::  r1:(-b+Sqrt[b^2-4a c])/(2a)

                          2 1/2
                -b + (-4a c + b )
#O[1]:   ---------------------
                      2a

#I[2]::  r2:(-b-Sqrt[b^2-4a c])/(2a)

                          2 1/2
                -b - (-4a c + b )
#O[2]:   ---------------------
                      2a

#I[3]::  a(x-r1)(x-r2)

                                2 1/2                          2 1/2
                    -b - (-4a c + b )              -b + (-4a c + b )
#O[3]:   a (x - ---------------------) (x - ---------------------)
                           2a                             2a

#I[4]::  Ex[%]

                   2
#O[4]:   c + a x  + b x
```

2. Lists, projections and parts

SMP manipulates symbolic expressions. Symbols and numbers form the fundamental units. They are combined into more complex expressions through projections and lists. Symbols were discussed in the previous section. In this section, we introduce lists and projections. A thorough understanding of these constructs is crucial for all but the most superficial use of SMP.

A list is an ordered (and indexed) collection of expressions. In its simplest form, a list consists of a set of expressions separated by commas and enclosed in brace brackets. An example is **{a+b,c+d,3,1}**. Such lists are called "contiguous".

It is often necessary to perform the same operation on several expressions. This may be achieved conveniently by collecting the expressions into a list, and then performing the operation on the complete list. This yields a list containing the results of performing the operation on each element of the original list.

```
#I[1]::  t:{a+b,c+d,3,1}
#O[1]:   {a + b,c + d,3,1}
#I[2]::  5t
#O[2]:   {5(a + b),5(c + d),15,5}
#I[3]::  Exp[t]
#O[3]:   {Exp[a + b],Exp[c + d],Exp[3],E}
#I[4]::  x+t^2
                                2          2
#O[4]:   {x + (a + b) ,x + (c + d) ,9 + x,1 + x}
#I[5]::  t+%
                                    2          2
#O[5]:   {a + b + x + (a + b) ,c + d + x + (c + d) ,12 + x,2 + x}
```

Notice that in line 5 two lists of the same length were added; the result was a list of the sums of their corresponding elements.

Sets of expressions in several lists may be combined by concatenating the lists using **Cat**.

```
#I[1]::  t:{a+b,x+1,c}
#O[1]:   {a + b,1 + x,c}
#I[2]::  Cat[t,t,{2,3}]
#O[2]:   {a + b,1 + x,c,a + b,1 + x,c,2,3}
#I[3]::  Rev[%]
#O[3]:   {3,2,c,1 + x,a + b,c,1 + x,a + b}
```

Rev[*list***]** reverses the order of elements in *list*.

A second important use of contiguous lists is to represent vectors. The dot product of two vectors *list1* and *list2* is given by *list1* **.** *list2* [8.2].

```
#I[1]::  {a,b,c}.{x,y,z}
#O[1]:   a x + b y + c z
```

The entries in a list may themselves be lists. A list of equal length lists may be considered as a matrix, with each list corresponding to a row of the matrix. Higher-rank tensors are represented as lists of matrices, lists of lists of matrices, and so on. Multiplication of two matrices or of a matrix by a vector is then obtained simply as a dot product of the corresponding lists [8.2].

```
#I[1]:: m:{{a,b},{c,d}}
#O[1]:  {{a,b},{c,d}}
#I[2]:: v:{p,q}
#O[2]:  {p,q}
#I[3]:: m.v
#O[3]:  {a p + b q,c p + d q}
#I[4]:: v.m
#O[4]:  {a p + c q,b p + d q}
#I[5]:: v.m.v
#O[5]:  p (a p + c q) + q (b p + d q)
#I[6]:: n:{{w,x},{y,z}}
#O[6]:  {{w,x},{y,z}}
#I[7]:: 3n
#O[7]:  {{3w,3x},{3y,3z}}
#I[8]:: n^2+2m
                  2       2       2       2
#O[8]:  {{2a + w ,2b + x },{2c + y ,2d + z }}
#I[9]:: m.n
#O[9]:  {{a w + b y,a x + b z},{c w + d y,c x + d z}}
#I[10]:: m.n.m
#O[10]:  {{a (a w + b y) + c (a x + b z),b (a w + b y) + d (a x + b z)},
          {a (c w + d y) + c (c x + d z),
            b (c w + d y) + d (c x + d z)}}
```

Note that in line 8, **n^2** gave the square of each element of the matrix **n**, not **n.n**.

Some mathematical matrix operations [9.6] are: **Det** (determinant), **Minv** (matrix inverse) and ****** (outer product).

```
#I[1]:: m:{{2,3},{-5,6}}
#O[1]:  {{2,3},{-5,6}}
#I[2]:: Det[m]
#O[2]:  27
#I[3]:: Minv[m]
#O[3]:  {{2/9,-1/9},{5/27,2/27}}
```

```
#I[4]:: n:{{w,x},{y,z}}

#O[4]:  {{w,x},{y,z}}

#I[5]:: m**n

#O[5]:  {{{{2w,2x},{2y,2z}},{{3w,3x},{3y,3z}}},
                {{{-5w,-5x},{-5y,-5z}},{{6w,6x},{6y,6z}}}}
```

Entries in a contiguous list are indexed by their positions. With **v:{a,b,c,d}**, the "projection" **v[3]** extracts the third component of **v**, and thus yields **c**.

```
#I[1]:: v:{a,x^2,3a/(b+c),1}

                 2  3a
#O[1]:  {a,x  ,------,1}
                  b + c

#I[2]:: v[2]

          2
#O[2]:  x

#I[3]:: v[2]+v[4-1]

           3a       2
#O[3]:  ----- + x
        b + c
```

A sequence of projections may be used to extract entries in lists of lists.

```
#I[1]:: w:{{a,b},{c,d}}

#O[1]:  {{a,b},{c,d}}

#I[2]:: w[2]

#O[2]:  {c,d}

#I[3]:: w[2][2]

#O[3]:  d

#I[4]:: w[1,1]

#O[4]:  a
```

Notice that **w[1][1]** is equivalent to **w[1,1]** (see, however, sect. 8).
Projections which specify an entry not present in a list are left unevaluated.

```
#I[1]:: v:{a,b,c}

#O[1]:  {a,b,c}

#I[2]:: v[7]

#O[2]:  v[7]

#I[3]:: v[3]+v[-2]

#O[3]:  c + v[-2]
```

Entries in a list may be specified by assignments. Existing entries may be changed by such assignments, or additional entries may be introduced.

```
#I[1]::  v:{a,b,c}

#O[1]:   {a,b,c}

#I[2]::  v[2]:p+q

#O[2]:   p + q

#I[3]::  v

#O[3]:   {a,p + q,c}

#I[4]::  v[4]:d

#O[4]:   d

#I[5]::  v

#O[5]:   {a,p + q,c,d}

#I[6]::  v[6]:p^2+q^2

          2    2
#O[6]:   p  + q

#I[7]::  v

                 2    2
#O[7]:   {[6]: p  + q , [1]: a, [2]: p + q, [3]: c, [4]: d}
```

On line 6, a value is defined for **v[6]**, even though no value has been assigned to **v[5]**. The resulting list is no longer contiguous: the indices of its entries are not given by their positions in the list. Instead, the index associated with each entry is displayed explicitly in square brackets. The resulting form for **v** is given on line 7.

The components of a vector may be defined in any order. When values have been specified for a suitable set of components, the list representing the vector becomes contiguous.

```
#I[1]::  u[1]:a

#O[1]:   a

#I[2]::  u

#O[2]:   {a}

#I[3]::  u[3]:c

#O[3]:   c

#I[4]::  u

#O[4]:   {[3]: c, [1]: a}

#I[5]::  u[2]:b

#O[5]:   b

#I[6]::  u

#O[6]:   {a,b,c}
```

The index of an entry in a list need not be a number: it may be any expression.

```
#I[1]::  f[x]:a

#O[1]:   a

#I[2]::  f

#O[2]:   {[x]: a}

#I[3]::  f[1+y]:b

#O[3]:   b

#I[4]::  f

#O[4]:   {[1 + y]: b, [x]: a}

#I[5]::  f[x]^2+f[z]

                 2
#O[5]:   f[z] + a

#I[6]::  f[z]:3

#O[6]:   3

#I[7]::  f

#O[7]:   {[z]: 3, [1 + y]: b, [x]: a}

#I[8]::  f[x]^2+f[z]

              2
#O[8]:   3 + a
```

In line 1, the value **a** is assigned to the projection **f[x]** of **f** with "filter" **x**. The value of the symbol **f** given on line 2 is then a list with one entry. The index of this entry is **x** and its value is **a**. The projection **f[x]** specifies that "part" of **f** indexed by the filter **x**, and hence extracts the value **a**. The assignment for **f[1+y]** on line 3 adds an entry with index **1+y** to the list giving the value of **f**. The list for **f** given on line 4 thus specifies the values of "parts" of **f** indexed with filters **x** or **1+y**. Parts of **f** indexed by other filters have not been specified. Thus on line 5, the projection **f[z]** is left in a symbolic unevaluated form, since no value for **f[z]** has been assigned. On line 6, an assignment for **f[z]** is made, so that on line 8, the projection **f[z]** may be evaluated.

Lists have analogues in other computer languages. Contiguous lists may be used as "arrays". Lists of contiguous lists are analogous to multi-dimensional arrays. Lists such as line 7 above resemble "records" or "structures".

Just as for symbols, assignment of a new value to a projection replaces any previous value.

```
#I[1]::  f[x]:a

#O[1]:   a

#I[2]::  f[y]:b

#O[2]:   b

#I[3]::  f

#O[3]:   {[y]: b, [x]: a}
```

```
#I[4]:  f[x]:c^2

                2
#O[4]:  c

#I[5]:  f

                                 2
#O[5]:  {[y]: b, [x]: c }

#I[6]:  f[x]+f[y]

                     2
#O[6]:  b + c
```

Values assigned to projections such as **f[x]** may be removed by **f[x]:**.
f: removes all values assigned to projections of **f**.

```
#I[1]:  f[x]:a

#O[1]:  a

#I[2]:  f[y]:b

#O[2]:  b

#I[3]:  f

#O[3]:  {[y]: b, [x]: a}

#I[4]:  f[y]:

#I[5]:  f

#O[5]:  {[x]: a}

#I[6]:  f[y]+f[x]

#O[6]:  a + f[y]

#I[7]:  f:

#I[8]:  f

#O[8]:  f

#I[9]:  f[y]+f[x]

#O[9]:  f[x] + f[y]
```

Projections may be used to represent "functions". **f[2]** may be considered as the "function" **f** with "argument" **2**. Assignments for projections then define values for "functions" at particular "points".

"System-defined" functions, such as **Log[2]**, are also projections. When no system-defined value exists for a projection, a value may be assigned to it.

```
#I[1]:  Log[x]

#O[1]:  Log[x]

#I[2]:  Log[x]:a

#O[2]:  a
```

```
#I[3]::  Log

#O[3]:   {[[x]]: a}

#I[4]::  Log[x]+Log[y]+Log[0]

#O[4]:   a + Log[0] + Log[y]

#I[5]::  Log[0]:Inf

#O[5]:   Inf

#I[6]::  Log[0]^2+Log[x^2]+Log[x]^2

                2      2      2
#O[6]:   Log[x ] + a  + Inf

#I[7]::  Log[1]

#O[7]:   0
```

In line 4, **Log[0]** was entered. This has no system-defined value; a value was assigned to it on line 5.

Notice that entry of **Log[0]** did not result in the printing of any warning message. Like other symbolic expressions for which no value has been defined, **Log[0]** was simply left unevaluated.

In line 7, **Log[1]** was automatically simplified to **0**. Such automatic evaluations of mathematical functions are performed only when the results are very simple. Automatic simplifications are performed before explicitly assigned values are used: **Log[1]** may thus not be assigned another value.

Some system-defined projections may be input in special forms. A complete table of such forms is given in [2.10]. For example, the factorial function **Fctl[***x***]** may be input as ***x*****!**. In all cases, ***x*****!** is equivalent to **Fctl[***x***]**.

```
#I[1]::  5!

#O[1]:   120

#I[2]::  Fctl[5]

#O[2]:   120

#I[3]::  x!:a

#O[3]:   a

#I[4]::  Fctl

#O[4]:   {[[x]]: a}

#I[5]::  x!^2

           2
#O[5]:   a

#I[6]::  (1/2)!:Sqrt[Pi]

             1/2
#O[6]:   Pi

#I[7]::  Fctl

                      1/2
#O[7]:   {[[1/2]]: Pi    , [[x]]: a}
```

We discussed above the extraction of an element in a matrix by two successive projections. Elements in a matrix may be introduced by assignments for projections with two filters.

```
#I[1]::   m[2,2]:d
#O[1]:    d
#I[2]::   m
#O[2]:    {[2]: {[2]: d}}
#I[3]::   m[1,1]:a
#O[3]:    a
#I[4]::   m
#O[4]:    {{a},{[2]: d}}
#I[5]::   m[2][1]:c
#O[5]:    c
#I[6]::   m
#O[6]:    {{a},{c,d}}
#I[7]::   m[1,2]:b
#O[7]:    b
#I[8]::   m
#O[8]:    {{a,b},{c,d}}
```

Projections with several symbolic filters may also be assigned values.

```
#I[1]::  a^2:3
#O[1]:   3
#I[2]::  Pow
#O[2]:   {[[a,2]]: 3}
#I[3]::  a^2+a^3
                 3
#O[3]:   3 + a
#I[4]::  0^0
           0
#O[4]:   0
#I[5]::  0^0:1
#O[5]:   1
#I[6]::  Pow
#O[6]:   {[[0,0]]: 1, [[a,2]]: 3}
#I[7]::  (1-1)^0
#O[7]:   1
```

In the assignment and evaluation of, for example, **Plus** projections, the commutative and associative properties of the **Plus** operation are used. (The specification of such properties will be discussed in sect. 8.)

```
#I[1]::  a+b:e^2

                2
#O[1]:   e

#I[2]::  Plus

                        2
#O[2]:   {[[a,b]]: e }

#I[3]::  a+b+c+d

                    2
#O[3]:   c + d + e

#I[4]::  a+c:f^2

                2
#O[4]:   f

#I[5]::  Plus

                      2                 2
#O[5]:   {[[a,c]]: f , [[a,b]]: e }

#I[6]::  a+b+c+d

                    2
#O[6]:   b + d + f
```

Not only mathematical functions but all SMP operations are specified by projections. Thus, for example, **a:b** is equivalent to the explicit projection **Set[a,b]**.

Just as parts of a list may be obtained by projections, so also parts of an arbitrary expression may be specified by suitable projections. In a function such as **f[a,b,c]** the filter **a** is specified by index **1**, **b** by **2** and **c** by **3**. The "projector" **f** is specified by index **0**.

```
#I[1]::  t:f[a,b,c]

#O[1]:   f[a,b,c]

#I[2]::  t[1]

#O[2]:   a

#I[3]::  t[3]

#O[3]:   c

#I[4]::  t[0]

#O[4]:   ´ f

#I[5]::  t[5]

#O[5]:   t[5]

#I[6]::  t[3]:d

#O[6]:   d

#I[7]::  t

#O[7]:   f[a,b,d]
```

In line 6, the part **t[3]** was re-assigned to have value **d**. The resulting complete projection **f[a,b,d]** was given on line 7.

The quote (´) on line 4 will be discussed in sect. 9.

Parts in functions input in special form are indexed according to their positions in the corresponding explicit projections.

```
#I[1]::  t:a^b

          b
#O[1]:   a

#I[2]::  t[1]

#O[2]:   a

#I[3]::  t[0]

#O[3]:   ´ Pow

#I[4]::  u:a+b+c+d

#O[4]:   a + b + c + d

#I[5]::  u[3]

#O[5]:   c
```

Some system-defined projections are output in forms for which the labelling of parts is not manifest. Such cases are indicated by a ***** at the beginning of the output. **Lpr[***expr***]** prints *expr* in a simple linear form in which the labelling of parts is manifest.

```
#I[1]::  a+b I+c

#O[1]:*  (a + c) + b I

#I[2]::  Lpr[%]

Cx[a + c,b]
```

Note that **I** is the complex unit (square root of -1).

Elements in lists of lists may be extracted by several successive projections (or equivalently a single projection with several filters). Similarly, parts in compound expressions may be extracted by suitable projections with several filters.

```
#I[1]::  t:(a+b)^2

                2
#O[1]:   (a + b)

#I[2]::  t[1]

#O[2]:   a + b

#I[3]::  t[1][2]

#O[3]:   b

#I[4]::  t[1,1]

#O[4]:   a

#I[5]::  t[2]

#O[5]:   2
```

```
#I[6]:  t[2,1]

#O[6]:  t[2,1]
```

Notice that since no part **t[2,1]** is present, the projection on line 6 is left unchanged.

Symbols and projections may carry numerical coefficients without explicit **Mult** projections. Numerical coefficients are thus irrelevant in defining positions of parts, even when they are rational numbers.

```
#I[1]:  t:2/3(a+2b)

          2(a + 2b)
#O[1]:    ---------
              3

#I[2]:  t[1]

#O[2]:  a

#I[3]:  t[2]

#O[3]:  2b

#I[4]:  t[-1]

#O[4]:  2/3
```

In line 4, projection with **-1** was used to extract the numerical coefficient.

The system-defined projection **Pos[***form***,***expr***]** gives a list of "positions" at which the subexpression *form* appears in the expression *expr*. Each position is specified by a list of the filters necessary to extract it by a projection.

```
#I[1]:  t:(a+b)^a+c^b

           b           a
#O[1]:  c  + (a + b)

#I[2]:  Pos[a,t]

#O[2]:  {{2,1,1},{2,2}}

#I[3]:  t[2,1,1]

#O[3]:  a

#I[4]:  Proj[t,{2,2}]

#O[4]:  a

#I[5]:  Pos[c^b,t]

#O[5]:  {{1}}
```

Notice that in line 1 parts were rearranged using the commutativity of **Plus**. The positions of parts are always determined by their location in the simplified form of an expression.

The projection **Proj[***expr***,{***x1***,***x2***,...}]** used on line 4 is equivalent to *expr***[***x1***,***x2***,...]**.

The specification of parts is important in performing operations on expressions. In many system-defined projections, a "domain" may be used to define a set of parts in an expression on which an operation is to be performed. Domains are described in [2.5].

In rearranging or simplifying expressions, it is often convenient to modify a particular part, while leaving other parts unchanged.

```
#I[1]::  t:x^2+3+(x^2-1)^2

                    2           2 2
#O[1]:  3 + x  + (-1 + x )

#I[2]::  t[3]:S[t[3],x->a]

                  2 2
#O[2]:  (-1 + a )

#I[3]::  t

                    2           2 2
#O[3]:  3 + x  + (-1 + a )
```

Further examples

Example 1
Find the angle between the vectors {1,-2,a} and {2a,-3,5} for the cases a=0 and a=1/2.

```
#I[1]:  v1:{1,-2,a}

#O[1]:  {1,-2,a}

#I[2]:  v2:{2a,-3,5}

#O[2]:  {2a,-3,5}

#I[3]:  ang:Acos[v1.v2/Sqrt[v1.v1 v2.v2]]

                          6 + 7a
#O[3]:  Acos[--------------------------]
                 2 1/2           2 1/2
       (5 + a )     (34 + 4 a )

#I[4]:  S[ang,a->0]

                6
#O[4]:  Acos[----------]
              1/2   1/2
       5     34

#I[5]:  N[%]

#O[5]:  1.0926

#I[6]:  S[ang,a->1/2]

                   19
#O[6]:  Acos[---------------]
                  1/2   1/2
       2 21/4     35

#I[7]:  N[%]

#O[7]:  0.794242
```

Example 2
Find the characteristic polynomial for the matrix

$$\begin{pmatrix} 1 & 2 & -1 \\ 4 & 3 & -1 \\ 4 & 2 & 1 \end{pmatrix}$$

```
#I[1]:  {{1,2,-1},{4,3,-1},{4,2,1}}

#O[1]:  {{1,2,-1},{4,3,-1},{4,2,1}}

#I[2]:  %-lam{{1,0,0},{0,1,0},{0,0,1}}

#O[2]:  {{1 - lam,2,-1},{4,3 - lam,-1},{4,2,1 - lam}}

#I[3]:  Det[%]

#O[3]:  -12 + 4lam + (1 - lam) (2 + (1 - lam) (3 - lam))
```

```
#I[4]::  Ex[%]

                                  2       3
#O[4]:   -7 - 5lam + 5 lam  - lam
```

Example 3

Find the real part of $e^{\frac{2}{5}\pi i}$.

```
#I[1]::  Exp[2I Pi/5]

#O[1]:*  Exp[2Pi/5 I]

#I[2]::   N[%]

#O[2]:*  0.309017 + 0.951057I

#I[3]::  Re[%]

#O[3]:   0.309017
```

Example 4

Form the 3×3 matrix M such that M_{ij} is ij.

```
#I[1]::  v:{1,2,3}

#O[1]:   {1,2,3}

#I[2]::  v**v

#O[2]:   {{1,2,3},{2,4,6},{3,6,9}}
```

Example 5

Change the last sin(x) in the expression **t** to $\sqrt{1-\cos^2(x)}$.

```
#I[1]::  t:4Sin[x] +3z^3(1+z+Sin[x])Sin[x]^2/(4(1+Sin[x]))

                 3                          2
         3 z  (1 + z + Sin[x]) Sin[x]
#O[1]:   ---------------------------- + 4Sin[x]
                4(1 + Sin[x])

#I[2]::  Pos[Sin,t]

#O[2]:   {{1,1,2,3,0},{1,1,3,1,0},{1,2,2,0},{2,0}}

#I[3]::  t[1,2,2]:Sqrt[1-Cos[x]^2]

                    2 1/2
#O[3]:   (1 - Cos[x] )

#I[4]::  t

                 3                          2
         3 z  (1 + z + Sin[x]) Sin[x]
#O[4]:   ---------------------------- + 4Sin[x]
                           2 1/2
            4(1 + (1 - Cos[x] )   )
```

3. Patterns

The assignment **f[x]:x^2** defines a projection of **f** with the specific filter **x** to be **x^2**. This assignment does not define any value for **f[y]**. To define a mathematical function **g** which forms the square of any expression which appears as its "argument", one performs the assignment **g[$x]:$x^2**. Here the "generic symbol" **$x** stands for any expression. Any symbol which begins with **$** is taken as a generic symbol, and may stand for any expression.

```
#I[1]::  f[x]:x^2

          2
#O[1]:   x

#I[2]::  f[y]+f[x]

                2
#O[2]:   f[y] + x

#I[3]::  g[$x]:$x^2

          2
#O[3]:   $x

#I[4]::  g

                  2
#O[4]:   {[$x]: $x }

#I[5]::  g[2]+g[x]+g[a^2]

              4     2
#O[5]:   4 + a  + x

#I[6]::  g[0]:c

#O[6]:   c

#I[7]::  g

                             2
#O[7]:   {[0]: c, [$x]: $x }

#I[8]::  g[0]+g[2]

#O[8]:   4 + c
```

The assignment for **g[0]** in line 6 defined a special value for the function **g**. This is used in preference to the more general assignment for **g[$x]** given in line 3.

When values for projections are assigned, those for more specific cases are placed before those for more general cases. The most restricted applicable assignment is therefore used in the evaluation of a particular projection.

```
#I[1]::  f[0]:a

#O[1]:   a

#I[2]::  f

#O[2]:   {[0]: a}
```

```
#I[3]::  f[$x]:1/$x

          1
#O[3]:    --
          $x

#I[4]::  f

                              1
#O[4]:   {[0]: a, [$x]:       --}
                              $x

#I[5]::  f[1]:b

#O[5]:   b

#I[6]::  f

                                    1
#O[6]:   {[1]: b, [0]: a, [$x]:     --}
                                    $x

#I[7]::  f[z]+f[0]^2

          1     2
#O[7]:    - + a
          z

#I[8]::  g[$n]:$n g[$n-1]

#O[8]:   $n g[-1 + $n]

#I[9]::  g

#O[9]:   {[$n]: $n g[-1 + $n]}

#I[10]::  g[5]

#O[10]:   0

#I[11]::  g[1]:1

#O[11]:   1

#I[12]::  g

#O[12]:   {[1]: 1, [$n]: $n g[-1 + $n]}

#I[13]::  g[5]

#O[13]:   120
```

In line 8, a "recursive" definition for the factorial function **g** was given. With no end condition specified, the result for **g[5]** in line 10 is **0**. With the specification on line 11, evaluation of **g[5]** uses the recursion relation on line 8 until the end point of line 11 is reached.

Assignments may be made not only for projections such as **f[$x]** but also for more complicated "patterns" such as **f[$x^2,$y]**.

```
#I[1]::  Exp[1+Log[$x]]:E $x

#O[1]:   E $x

#I[2]::  Exp

#O[2]:   {[[1 + Log[$x]]]: E $x}
```

```
#I[3]::  Exp[1+Log[a+b]]

#O[3]:   E (a + b)

#I[4]::  Exp[2+Log[a+b]]

#O[4]:   Exp[2 + Log[a + b]]

#I[5]::  Exp[1+$x]:h[$x]

#O[5]:   h[$x]

#I[6]::  Exp

#O[6]:   {[[1 + Log[$x]]]: E $x, [[1 + $x]]: h[$x]}

#I[7]::  Exp[1+a^2]+Exp[1+Log[b]]

                 2
#O[7]:   h[a ] + b E
```

All occurrences of a particular generic symbol in a pattern must stand for the same expression.

```
#I[1]::  $x^$x:q[$x]

#O[1]:   q[$x]

#I[2]::  Pow

#O[2]:   {[[$x,$x]]: q[$x]}

#I[3]::  x^2+y^y

                   2
#O[3]:   q[y] + x

#I[4]::  $x^($x+$y):p[$x,$y]

#O[4]:   p[$x,$y]

#I[5]::  x^(x+a)

#O[5]:   p[x,a]

#I[6]::  Log[a+b]^(Log[a+b]+c^2)

                          2
#O[6]:   p[Log[a + b],c ]
```

Patterns match expressions whenever replacements for generic symbols may be deduced without solving explicit equations. For example, while **x^$n** matches **x^2**, **x^$n** does not match **x** alone, since in the latter case, determination of the necessary replacement for **$n** would require solution of the equation **x^$n=x**. Nevertheless, **f[$n,x^$n]** does match **f[1,x]** since the replacement for **$n** may be deduced from a direct comparison between the first filters, without solving an equation. Similarly, **f[1-$x,$x]** matches **f[-2,3]**, while **f[1-$x,1+$x]** matches **f[1-2a,1+2a]**, but does not match **f[-2,4]**.

```
#I[1]::  f[2/$x,$x]:h[$x]

#O[1]:   h[$x]

#I[2]::  f[1,2]

#O[2]:   h[2]
```

```
#I[3]::  f[2,1]

#O[3]:   h[1]

#I[4]::  f[2a,1/a]

                1
#O[4]:   h[-]
                a

#I[5]::  g[2/$x,1/$x]:i[$x]

#O[5]:   i[$x]

#I[6]::  g[2/x,1/x]

#O[6]:   i[x]

#I[7]::  g[2,1]

#O[7]:   g[2,1]

#I[8]::  gp[2$xi,$xi]:i[1/$xi]

                  1
#O[8]:   i[-----]
                $xi

#I[9]::  gp[2,1]

#O[9]:   i[1]

#I[10]::  j[2$x]:jp[$x]

#O[10]:   jp[$x]

#I[11]::  j[x]

#O[11]:   jp[x/2]

#I[12]::  j[1]

#O[12]:   jp[1/2]
```

Notice that in line 10, **2$x** is treated as a single generic symbol matching any expression, regardless of its numerical coefficient.

Pattern matching takes account of commutativity and associativity properties of functions.

```
#I[1]::  b+$x:u[$x]

#O[1]:   u[$x]

#I[2]::  a+b

#O[2]:   u[a]

#I[3]::  a+b+c

#O[3]:   c + u[a]

#I[4]::  a $x^2:v[$x]

#O[4]:   v[$x]
```

```
#I[5]::  a b c^2 d^2

                2
#O[5]:   b d   v[c]

#I[6]::  f[$x+$y]:g[$x]g[$y]

#O[6]:   g[$x] g[$y]

#I[7]::  f[x+y]+f[x+y+z]

#O[7]:   f[x + y + z] + g[x] g[y]
```

In line 7, **$x+$y** matches **x+y** but not **x+y+z**. Multi-generic symbols such as **$$x** are used to represent a sequence of arguments to a function. **$$x+$y** would match **x+y+z** with **$y** replaced by **x** and **$$x** replaced by **y+z**.

```
#I[1]::  f[$$x+$y]:g[$$x]g[$y]

#O[1]:   g[$y] g[$$x]

#I[2]::  f[x+y+z]

#O[2]:   g[x] g[y + z]

#I[3]::  f[x]+f[x+y]+f[x+y+z+w]

#O[3]:   f[x] + g[w] g[x + y + z] + g[x] g[y]

#I[4]::  mu.$$x.mu:h[$$x]

#O[4]:   h[$$x]

#I[5]::  p.mu.q.r.mu.p

#O[5]:   p.h[q.r].p

#I[6]::  a.mu.a1.b1.mu.c1.mu.d1.e1.mu.b.c

#O[6]:   a.h[a1.b1].c1.h[d1.e1].b.c

#I[7]::  a.mu.a1.b1.mu.c1.mu.b

#O[7]:   a.h[a1.b1].c1.mu.b

#I[8]::  Log[a b c d]

#O[8]:   Log[a b c d]

#I[9]::  Log[$x $$x]:Log[$x]+Log[$$x]

#O[9]:   Log[$x] + Log[$$x]

#I[10]::  Log[a b c d]

#O[10]:   Log[a] + Log[b] + Log[c] + Log[d]
```

The class of expressions matched by a particular pattern may be restricted by imposing conditions on the generic symbols in it. Whereas **$x** stands for any expression, **$x_=$x>0** stands only for expressions such that **$x>0** is determined to be true.

```
#I[1]::  g[$n_=$n>0]:h[$n]

#O[1]:   h[$n]
```

```
#I[2]:  g[2]+g[-3]+g[x]

#O[2]:  g[-3] + g[x] + h[2]

#I[3]:  gamma[$x_=Natp[$x]]:($x-1)!

#O[3]:  (-1 + $x)!

#I[4]:  gamma[4]+gamma[-2]

#O[4]:  6 + gamma[-2]
```

In line 3, **Natp[***x***]** is a projection which yields **1** (representing "true") if *x* is determined to be a natural number (positive integer), and **0** (representing "false") otherwise. Similarly, **Intp** tests for an integer, **Numbp** for a number and **Listp** for a list.

x=*y* yields **1** if *x* and *y* are equal, and **0** if they are determined to be unequal. If no definite truth value is obtained, the expression *x*=*y* is left unchanged. Other relational operators are ~= (unequal), > (greater than), < (less than), >= and <=.

Finally, ~*x* means "not *x*", *x*&*y* means "*x* and *y*" and *x*|*y* means "*x* or *y*".

```
#I[1]:  3=2

#O[1]:  0

#I[2]:  x=2

#O[2]:  x = 2

#I[3]:  3>2>=1 & x~=y

#O[3]:  x ~= y

#I[4]:  f[$x,$y_=$x>$y & 0<$y<1]:g[$x,$y]

#O[4]:  g[$x,$y]

#I[5]:  f[2,1/2]

#O[5]:  g[2,1/2]

#I[6]:  f[x,1/2]

#O[6]:  f[x,1/2]
```

Without giving a definite numerical value for a symbol **x**, one may assert that **x** is positive by the assignment **x>0:1**. Similarly, **x** may be defined as an integer by **Intp[x]:1**.

Further examples

Example 1
Find the set of all subparts of an expression.

```
#I[1]::  t:6(1+4x)(2+3x+x z)
#O[1]:   6(1 + 4x) (2 + 3x + x z)
#I[2]::  Pos[$x,t]
#O[2]:   {{1,1},{1,2},{1},{2,1},{2,2},{2,3,1},{2,3,2},{2,3},{2},{0}}
```

Example 2
Define rules to simplify $e^{i\pi\frac{n}{2}}$ where n is an integer.

```
#I[1]::  Exp[I Pi ($x_=Intp[2$x])]:I^Mod[2$x,4]
          Mod[2$x,4]
#O[1]:*  I
#I[2]::  Exp[3I Pi]
#O[2]:   -1
#I[3]::  Exp[3/2 I Pi]
#O[3]:*  -I
#I[4]::  Exp[I Pi]
#O[4]:*  Exp[Pi I]
```

Example 3
Find the 5^{th} and 10^{th} terms in the series defined by

$$f(x) = f(x-1) + a\ f(x-2)$$
$$f(1) = f(2) = 1$$

```
#I[1]::  f[$x]:f[$x-1]+a f[$x-2]
#O[1]:   f[-1 + $x] + a f[-2 + $x]
#I[2]::  f[1]:f[2]:1
#O[2]:   1
#I[3]::  f[5]
#O[3]:   1 + 2a + a (1 + a)
```

```
#I[4]::  f[10]
#O[4]:   1 + 2a + a (1 + a) + a (1 + 2a) + a (1 + 2a + a (1 + a))
           + a (1 + 2a + a (1 + a) + a (1 + 2a))
           + a (1 + 2a + a (1 + a) + a (1 + 2a) + a (1 + 2a + a (1 + a)))
           + a (1 + 2a + a (1 + a) + a (1 + 2a) + a (1 + 2a + a (1 + a))
                  + a (1 + 2a + a (1 + a) + a (1 + 2a)))
#I[5]::  Ex[%]
                        2       3       4
#O[5]:   1 + 8a + 21 a  + 20 a  + 5 a
```

Example 4

Define a function of three variables which vanishes if the difference between any arguments is greater than 1 and is equal to one otherwise.

```
#I[1]::  f[$1,$2,$3]:~(Abs[$1-$2]>1 | Abs[$2-$3]>1 | Abs[$3-$1]>1)
#O[1]:   ~ (Abs[-$1 + $3] > 1 | Abs[$1 - $2] > 1 | Abs[$2 - $3] > 1)
#I[2]::  f[0,0,0]
#O[2]:   1
#I[3]::  f[2,0,0]
#O[3]:   0
#I[4]::  f[1,2,3]
#O[4]:   0
#I[5]::  f[0,.5,.7]
#O[5]:   1
```

4. Building up calculations

One important feature of SMP is the possibility of constructing a system of rules and definitions, and then using this system repeatedly for many different cases. SMP remembers and organizes each definition or rule given.

Everything typed during an SMP job is kept in a "record file" which may be re-read or printed either during the job or after the job has finished. Details of the "record file" are given in the Implementation Notes; in most implementations, it is named **smp.out**. At most installations, each SMP job run by a particular user generates the same record file: if the record is to be saved permanently, it must be moved to another file before the next SMP job is started. The record file may also be printed on paper to provide a permanent record. The necessary procedures are given in the Implementation Notes. Note that only input and output SMP expressions are included in the record file: operations such as editing are omitted.

The input and output lines in an SMP job may be manipulated just like other expressions. The *i*th output expression is given by **@***i*. **@{***i*,*j*,...**}** gives a list of output expressions *i*, *j*, ... Each output line is assigned as the value of a projection of **#O**, as indicated by the **#O[***i***]:** at the beginning of each output line; **@***i* is a short notation for **#O[***i***]**. **#O** is a list of all output expressions.

```
#I[1]:  a:x^2+1

                2
#O[1]:  1 + x

#I[2]:  a+a^2

                2          2 2
#O[2]:  1 + x  + (1 + x )

#I[3]:  @1

                2
#O[3]:  1 + x

#I[4]:  @{2,1}

                2          2 2       2
#O[4]:  {1 + x  + (1 + x ) ,1 + x }

#I[5]:  #O

                      2          2 2       2              2
#O[5]:  {[4]: {1 + x  + (1 + x ) ,1 + x }, [3]: 1 + x ,

                           2          2 2             2
              [2]: 1 + x  + (1 + x ) , [1]: 1 + x ,

              [0]: {"/home/cmc/mk/smp.init"}}

#I[6]:   {1,5..9}

#O[6]:  {1,5,6,7,8,9}

#I[7]:  @{2..4,6}

                2          2 2       2        2          2 2       2
#O[7]:  {1 + x  + (1 + x ) ,1 + x ,{1 + x  + (1 + x ) ,1 + x },

              {1,5,6,7,8,9}}
```

In line 5, **#O[0]** is a list of "initialization files" read in when an SMP job is initiated, and containing SMP input which is simplified at the beginning of the job.

In line 6, *i*..*j* gives *i*,*i*+**1**,...,*j*.

Input expressions are stored in an unevaluated form in the list **#I**. The **#I[***i***]::** which begins each input line indicates that the unsimplified input expression is assigned as a "delayed value" for the projection

#I[*i***]**. An important distinction exists between the "immediate assignment" *a***:***b* used, for example, for the output lines **#O**, and the "delayed assignment" *a***::***b* used for the input lines **#I**. In immediate assignment, *b* is simplified, and the result is assigned as the value of *a*. In delayed assignment, the unsimplified form of *b* is maintained as the value of *a*, and is simplified afresh each time *a* is used. Thus subsequent re-definitions relevant to the simplification of *b* are used with delayed assignment but not with immediate assignment.

```
#I[1]::  a:x+1

#O[1]:   1 + x

#I[2]::  a^2+a

                          2
#O[2]:   1 + x + (1 + x)

#I[3]::  a:x-2

#O[3]:   -2 + x

#I[4]::  @2

                          2
#O[4]:   1 + x + (1 + x)

#I[5]::  #I[2]

                             2
#O[5]:   -2 + x + (-2 + x)

#I[6]::  t::b+2

#O[6]:   ´ b + 2

#I[7]::  u:b+2

#O[7]:   2 + b

#I[8]::  b:3

#O[8]:   3

#I[9]::  t

#O[9]:   5

#I[10]::  u

#O[10]:   5

#I[11]::  b:4

#O[11]:   4

#I[12]::  t

#O[12]:   6

#I[13]::  u

#O[13]:   5
```

Both **t** and **u** as defined in lines 6 and 7 depend on **b**. After the value of **b** is reassigned in line 11, the value obtained for **t** uses the new value of **b** while that for **u** does not.

The "apostrophe", "acute accent" or "quote mark" (´) in line 6 indicates that the expression **b+2** is unsimplified.

As illustrated in the example, an input expression or "command" may be "re-executed" in the current environment or with current definitions simply by asking for its value. This operation may be considered as a simple

case of a "program" executed many times under different conditions or using different data.

```
#I[1]::  x^2-1

                  2
#O[1]:   -1 + x

#I[2]::  S[%,x->3]

#O[2]:   8

#I[3]::  x^3+4x-1

                        3
#O[3]:   -1 + 4x + x

#I[4]::  #I[2]

#O[4]:   38

#I[5]::  (x+y+4)^2

                       2
#O[5]:   (4 + x + y)

#I[6]::  #I[2]

                  2
#O[6]:   (7 + y)

#I[7]::  Edh[#I[2]]

         S[%,x -> 3]
<edit>     #^@3   5
         S[@3,x -> 5]
<edit>
#O[7]:   144

#I[8]::  f[$a]::S[$a,x->3]

#O[8]:   ' S[$a,x -> 3]

#I[9]::  f[x^2-1]

#O[9]:   8

#I[10]::  f[(x+1)(x-y)^2]

                      2
#O[10]:   4 (3 - y)

#I[11]::  f

#O[11]:   {[$a]:: S[$a,x -> 3]}
```

On line 7, **Edh** was used to change the unevaluated input line **#I[2]** using the editor.

After the definition on line 8, **f** acts as a "function" which substitutes **3** for **x** in any expression. Notice that if **f** had been defined by an immediate (**:**) rather than a delayed (**::**) assignment, the substitution would have been performed on the symbol **$a** at the time of assignment, rather than on each separate occasion when a projection of **f** is used, with the result that **f[$a]** would have been set to **$a**.

When a sequence of commands is typed on successive input lines, the results of each command are printed on the corresponding output line. A sequence of commands whose intermediate results need not be displayed may be entered on a single input line, separated by **;**. A semicolon at the end of an input line causes the input expression or expressions to be simplified without printing the result.

```
#I[1]::  a:3;a^2-a

#O[1]:   6

#I[2]::  t:a^a-a^2;

#I[3]::  t

#O[3]:   18
```

A set of definitions which will be used in several SMP jobs may be saved in an external file, and read into each SMP job as required. An external file consists of SMP input lines which are simplified in order when the file is read, just as if they were typed as explicit input. An external file with name *file* is read into an SMP job by **<***file*. Many external files containing functions with specialized applications are provided with SMP, as described in the SMP Library. An example is the file **XLCM** (note that file names may differ between installations, as specified in the Implementation Notes):

```
                /**  Lowest common multiple  **/

/*: LCM[$n1,$n2,...]
        yields the lowest common multiple of $n1,$n2,... . */

        LCM :Flat
        LCM[$n1,$n2] :: ($n1 $n2)/Gcd[$n1,$n2]
```

The text enclosed between **/*** and ***/** is treated as "commentary" and is not processed.

```
#I[1]::  <XLCM

#I[2]::  LCM[12,21,7]

#O[2]:   84

#I[3]::  LCM[2,6,8]

#O[3]:   24

#I[4]::  <Xlcm

#O[4]:   <Xlcm
```

In line 4, an attempt was made to read the non-existent file **Xlcm**: return of the unevaluated command signifies failure of the attempt. Files other than the standard SMP ones may be specified by complicated names, such as **/u1/swolf/smp/fun**; such file names must be enclosed in quotes: **<"/u1/swolf/smp/fun"**.

Many external files contain tables of relations and transformations for mathematical functions, given in the form of "replacements" to be applied when required using **S** (see sect. 1). For example, the file **XTr41** gives replacements for hyperbolic functions:

```
                /**  Elementary transcendental functions - 4.1  **/

        /** 4.1  Hyperbolic functions - Relations to other functions **/

STr_:Ldist

STr[4,1,1]:   Sinh[$x] -> (Exp[$x]-Exp[-$x])/2

STr[4,1,2]:   Sinh[$x] -> -I Sin[I $x]

STr[4,1,3]:   Cosh[$x] -> (Exp[$x]+Exp[-$x])/2
```

```
STr[4,1,4]:   Cosh[$x] -> Cos[I $x]

STr[4,1,5]:   Tanh[$x] -> (Exp[$x]-Exp[-$x])/(Exp[$x]+Exp[-$x])

STr[4,1,6]:   Tanh[$x] -> -I Tan[I $x]


#I[1]::  <XTr41

#I[2]::  t:Sinh[x+1]+Sinh[x-1]+Tanh[2x]

#O[2]:   Sinh[-1 + x] + Sinh[1 + x] + Tanh[2x]

#I[3]::  S[t,STr[4,1,1]]

#O[3]:   -Exp[-(-1 + x)]   Exp[-1 + x]   Exp[-(1 + x)]   Exp[1 + x]
        ---------------- + ----------- - ------------- + ----------
               2                2              2             2

                + Tanh[2x]

#I[4]::  S[t,STr[4,1,3]]

#O[4]:   Sinh[-1 + x] + Sinh[1 + x] + Tanh[2x]

#I[5]::  S[t,STr[4,1,{4..6}]]

#O[5]:   -Exp[-2x] + Exp[2x]
        -------------------- + Sinh[-1 + x] + Sinh[1 + x]
        Exp[-2x] + Exp[2x]

#I[6]::  S[t,STr[4,1,1],STr[4,1,6]]

#O[6]:*  (-Exp[-(-1 + x)]   Exp[-1 + x]   Exp[-(1 + x)]   Exp[1 + x]
         ---------------- + ----------- - ------------- + ----------)
                2                2              2             2

                + -Tan[2x I] I

#I[7]::  S[t,STr[4,1]]

#O[7]:   -Exp[-2x] + Exp[2x]   Exp[-(-1 + x)]   Exp[-1 + x]
        -------------------- - ---------------- + -----------
        Exp[-2x] + Exp[2x]           2                2

                  Exp[-(1 + x)]   Exp[1 + x]
                - ------------- + ----------
                        2             2
```

In line 7, **STr[4,1]** is the list of all relations defined in **XTr41**. **S** applies the relations in the order given: in this case, the first three relations transform **t** so that the last replacements have no effect.

The relations in **XTr41** were given as replacements and used selectively with **S**. **Arep[STr[4,1]]** would have converted the list of replacements to assignments and thus defined the transformations to be performed automatically whenever applicable.

The SMP Library contains many external files. Some simply define additional relations for mathematical functions. Others introduce entirely new mathematical objects. Still others define projections to be used in manipulating expressions. These external files may be used without understanding their construction. They may also be studied as examples of more advanced uses of SMP.

The external file **XLdEq** defines the function **LdEq** used in manipulating lists of equations:

```
                /** Equation list distribution **/

/*: LdEq[$expr]
        distributes /* Eq */ over lists in $expr, thus converting equations
        involving lists into lists of equations. */

        LdEq[$expr] :: Ldist[$expr,'Eq]


#I[1]:: <XLdEq

#I[2]:: {x,y}={a,b}

#O[2]:  {x,y} = {a,b}

#I[3]:: LdEq[%]

#O[3]:  {x = a,y = b}
```

New external files may be prepared outside an SMP job (usually using an editor) according to the procedure outlined in the Implementation Notes. Each input line is terminated by a ⊂*newline*⊃. When an input expression extends over several lines, each intermediate line must end with a backslash (\). Descriptive comments (whose value should not be underestimated) are enclosed between **/*** and ***/**, and may extend over several lines.

External files may also be prepared using definition or results in an SMP job. **Put[***x1***,***x2***,...,***file***]** places assignments for *x1*, *x2*, ... or their projections in *file* in a form suitable for subsequent input.

```
#I[1]:: Log[$x/$y]:Log[$x]-Log[$y]

#O[1]:  Log[$x] - Log[$y]

#I[2]:: Log[$x $$x]:Log[$x]+Log[$$x]

#O[2]:  Log[$x] + Log[$$x]

#I[3]:: Log[$x^$y]:$y Log[$x]

#O[3]:  $y Log[$x]

#I[4]:: Log

                 $y
#O[4]:  {[[$x    ]]: $y Log[$x], [[$x $$x]]: Log[$x] + Log[$$x],

                   $x
                 [[--]]: Log[$x] - Log[$y]}
                   $y

#I[5]:: t:Log[x^2 y^3/(s+1)^3]

#O[5]:  2Log[x] + 3Log[y] - 3Log[1 + s]

#I[6]:: Put[Log,t,"log.ex"]

#O[6]:  2Log[x] + 3Log[y] - 3Log[1 + s]
```

The file **log.ex** is then

```
Log[$x/$y] : Log[$x] + -Log[$y]
Log[$x*$$x] : Log[$x] + Log[$$x]
Log[$x^$y] : $y*Log[$x]
t : 2Log[x] + 3Log[y] + -3Log[1 + s]
```

This file may be used as input in another SMP job, or may be edited and manipulated outside of SMP.

```
#I[1]::  Log[x y^2]

                 2
#O[1]:   Log[x y ]

#I[2]::  <"log.ex"

#O[2]:   2Log[x] + 3Log[y] - 3Log[1 + s]

#I[3]::  @1

#O[3]:   Log[x] + 2Log[y]

#I[4]::  t-%

#O[4]:   Log[x] + Log[y] - 3Log[1 + s]

#I[5]::  Put[#O,resO]

#O[5]:   {[4]: Log[x] + Log[y] - 3Log[1 + s], [3]: Log[x] + 2Log[y],
              [2]: 2Log[x] + 3Log[y] - 3Log[1 + s],
              [1]: Log[x] + 2Log[y], [0]: {"/home/cmc/mk/smp.init"}}

#I[6]::  Put[#I,resI]

#O[6]:   {[5]:: Put[#O,resO], [4]:: t - %, [3]:: #O[1],
                                                 2
              [2]:: <"log.ex", [1]:: Log[x y ], [0]:: Init}
```

On lines 5 and 6 the complete lists of output and input expressions were saved in the files **resO**

```
#O[0,1] : "/home/cmc/mk/smp.init"
#O[1] : Log[x] + 2Log[y]
#O[2] : 2Log[x] + 3Log[y] + -3Log[1 + s]
#O[3] : Log[x] + 2Log[y]
#O[4] : Log[x] + Log[y] + -3Log[1 + s]
```

and **resI**

```
#I[0] :: Init
#I[1] :: Log[x*y^2]
#I[2] :: <"log.ex"
#I[3] :: #O[1]
#I[4] :: t + -%
#I[5] :: Put[#O,resO]
```

In most implementations, printed hard copies of expressions may be generated from within SMP by **Hard[***expr***]**. When possible, hard copies of graphics output (see sect. 7) may be generated in this way.

During an SMP job, it is sometimes necessary to manipulate external files or perform other operations outside SMP. The remainder of an input line whose first character is **!** is passed as a command to the monitor (shell or command line interpreter), and is ignored by SMP. The details of some possible commands are given in the Implementation Notes.

```
#I[1]::  t:2

#O[1]:   2
```

```
#I[2]::  !p log.ex
Log[$x/$y] : Log[$x] + -Log[$y]
Log[$$x*$x] : Log[$$x] + Log[$x]
Log[$x^$y] : $y*Log[$x]
t : Log[x^2*y^3/(s + 1)^3]

#I[2]::  t^2

#O[2]:   4
```

The monitor command **p log.ex** used to print the file **log.ex** is specific to the implementation used for the example.

Further examples

Example 1
Define a function to convert sin(x) and cos(x) in an expression to complex exponential form. Save the function in the file **csrep**.

```
#I[1]::  r1:Cos[$x]->(Exp[I $x]+Exp[-I $x])/2

                      Exp[-$x I] + Exp[$x I]
#O[1]:*  Cos[$x] -> ----------------------
                             2

#I[2]::  r2:Sin[$x]->-I(Exp[I $x]-Exp[-I $x])/2

                     -(-Exp[-$x I] + Exp[$x I])
#O[2]:*  Sin[$x] -> -------------------------- I
                             2

#I[3]::  ef[$e]::S[$e,r1,r2]

#O[3]:   ´ S[$e,r1,r2]

#I[4]::  Sin[3.2]+(1+Cos[Pi/4]^2)^(1/2)

                          2 0.5
#O[4]:   (1 + Cos[0.25Pi] )    + Sin[3.2]

#I[5]::  ef[%]

                                                 2
           (Exp[-Pi/4 I] + Exp[Pi/4 I])    1/2
#O[5]:* (1 + ----------------------------)
                         4

                    Exp[-16I/5]   Exp[16I/5]
               + (------------ - ----------) I
                         2             2

#I[6]::  N[%]

#O[6]:   1.16637

#I[7]::  Put[r1,r2,ef,csrep]

#O[7]:   {[$e]:: S[$e,r1,r2]}
```

Example 2
Define a function to evaluate the norm of a vector.

```
#I[1]::  norm[$l]::N[Sqrt[$l.$l]]

#O[1]:   ´ N[Sqrt[$l.$l]]

#I[2]::  v:{.23,-1.1,2.7}

#O[2]:   {0.23,-1.1,2.7}

#I[3]::  norm[v]

#O[3]:   2.92453
```

Example 3
Simplify $\Gamma(\frac{13}{4})\,\Gamma(\frac{15}{4})$ using the transformations given in the external file **XGamma**.

```
#I[1]:: <XGammaS
#O[1]: <XGamma
#I[2]:: SGamma[8]
#O[2]: SGamma[8]
#I[3]:: Gamma[3.75] S[Gamma[3.25],SGamma[8]]
#O[3]: Gamma[3.25] Gamma[3.75]
#I[4]:: SGamma[6]
#O[4]: SGamma[6]
#I[5]:: S[@3,SGamma[6]]
#O[5]: Gamma[13/4] Gamma[15/4]
#I[6]:: N[%]
#O[6]: 11.2753
#I[7]:: N[Gamma[3.25]Gamma[3.75]]
#O[7]: 11.2753
```

5. Manipulating expressions

The two expressions **b+c+a** and **a+c+b**, while superficially different, are mathematically equal. This equality is made manifest by casting both expressions into the canonical form **a+b+c** in which the arguments of the commutative function **Plus** have been ordered. Simple transformations such as this are performed automatically by SMP.

In the more complicated case of the expressions **x+x^2** and **x(1+x)**, a canonical form may be obtained by expansion (using **Ex**). However, no algorithm exists for placing an arbitrarily complicated expression in a canonical form. The beginning of this section concentrates on the variety of functions in SMP for transforming rational expressions. The external files discussed in sect. 4 contain many transformations for transcendental and special functions.

```
#I[1]:: a:(1+x^2)/((1-x)(2-x)(3-x))

                        2
                1 + x
#O[1]:  ------------------------
        (1 - x) (2 - x) (3 - x)

#I[2]:: Int[%,x]

#O[2]:  -5Log[-3 + x] + 5Log[-3 + #1] + 5Log[-2 + x] - 5Log[-2 + #1]

                - Log[-1 + x] + Log[-1 + #1]

#I[3]:: D[%,x]

          -5        5         1
#O[3]:  ------ + ------ - ------
        -3 + x   -2 + x   -1 + x

#I[4]:: b:%;

#I[5]:: Neq[a,b]

#O[5]:  1

#I[6]:: Rat[b]

        5 - 5x - (-3 + x) (-2 + x)
#O[6]:  --------------------------
        (-3 + x) (-2 + x) (-1 + x)

#I[7]:: Ex[a]

                                      2
                1                    x
#O[7]:  ------------------ + ------------------
                   2    3                  2    3
        6 - 11x + 6 x  - x    6 - 11x + 6 x  - x

#I[8]:: Col[%]

                  2
          1 + x
#O[8]:  ------------------
                   2    3
        6 - 11x + 6 x  - x

#I[9]:: Fac[%]

                       2
                -(1 + x )
#O[9]:  --------------------------
        (-3 + x) (-2 + x) (-1 + x)
```

```
#I[10]::  Pf[a,x]

             -5        5         1
#O[10]:  ------ + ------ - ------
         -3 + x   -2 + x   -1 + x
```

Since **b** is the result of differentiating the integral of **a**, **a** and **b** must be mathematically equal. They are, however, written in different forms. The projection **Neq[***expr1***,***expr2***]** used on line 5 tests for the numerical equality of *expr1* and *expr2* by substituting sets of random numbers for symbols which appear in them*.

The projection **Rat[***expr***]** used on line 6 introduces a common denominator for all terms in *expr*, multiplying the numerators by the necessary factors, and in this case casts **b** into the same form as **a**.

Lines 7 through 10 give further forms for **a**. On line 7, **Ex** was used to perform all the possible distributions, leaving an expression containing no parentheses. In line 8, terms with the same denominator were collected using **Col**. The result was then factorized in line 9. Finally, in line 10, the expression **a** was placed in a partial fraction form with respect to **x** using **Pf**.

```
#I[1]::  (1-x^2)(x+a)(x-3)+(2-a)(x^2+1)(1+x)^6

                          2                      6       2
#O[1]:  (-3 + x) (1 - x ) (a + x) + (1 + x)  (1 + x ) (2 - a)

#I[2]::  Ex[%]

                                    2         3         4         5
#O[2]:  2 - 4a + 9x - 5a x - 13a x  - 27a x  - 30a x  - 26a x

              6         7         8       2       3       4       5
        - 16a x  - 6a x  - a x  + 33 x  + 55 x  + 59 x  + 52 x

              6       7       8
        + 32 x  + 12 x  + 2 x

#I[3]::  Fac[%]

                                                 2         3         4
#O[3]:  -(1 + x) (-2 + 4a - 7x + a x + 12a x  + 15a x  + 15a x

                          5       6       7       2       3       4
                  + 11a x  + 5a x  + a x  - 26 x  - 29 x  - 30 x

                          5       6       7
                  - 22 x  - 10 x  - 2 x )

#I[4]::  Cb[@2,a]

                                    2         3         4         5         6
#O[4]:  2 + 9x + a (-4 - 5x - 13 x  - 27 x  - 30 x  - 26 x  - 16 x

                          7     8         2       3       4       5
                    - 6 x  - x ) + 33 x  + 55 x  + 59 x  + 52 x

              6       7       8
        + 32 x  + 12 x  + 2 x

#I[5]::  Coef[a,@2]

                        2         3         4         5         6       7     8
#O[5]:  -4 - 5x - 13 x  - 27 x  - 30 x  - 26 x  - 16 x  - 6 x  - x
```

* This test is clearly probabilistic in nature; however, the chance that it will fail is infinitesimal.

```
#I[6]:: Cb[@2,x^$i]

                                  2                    3
#O[6]:  2 - 4a + 9x - 5a x + x  (33 - 13a) + x  (55 - 27a)

                 4                    5                    6
          + x  (59 - 30a) + x  (52 - 26a) + x  (32 - 16a)

                 7                    8
          + x  (12 - 6a) + x  (2 - a)

#I[7]:: Cb[@2,{x^$i,x}]

                                    2                    3
#O[7]:  2 - 4a + x (9 - 5a) + x  (33 - 13a) + x  (55 - 27a)

                 4                    5                    6
          + x  (59 - 30a) + x  (52 - 26a) + x  (32 - 16a)

                 7                    8
          + x  (12 - 6a) + x  (2 - a)
```

Application of **Ex** on line 2 gave a comparatively large expression: to avoid unintentional generation of huge expressions, **Ex** should be used with forethought. In line 3, **Fac** could not entirely reverse the effect of **Ex**. **Cb[***expr***,***form***]** combines coefficients of *form* in *expr*, or of patterns matching *form*. Thus in line 6, **Cb[@2,x^$i]** combines coefficients of each power of **x**. In line 7, the list **{x^$i,x}** causes collection of terms proportional to **x** as well as to powers of **x**. On line 5, **Coef** was used to extract the total coefficient of **a**.

The various projections illustrated in the examples above provide tools for manipulating expressions into "simpler" forms. The optimal procedure in particular cases is usually obtained through trial and error.

It is often convenient to apply the same operation or "simplification" to a set of expressions. **Map[***temp***,***expr***]** applies the "template" *temp* to parts in *expr* lying on the "first level".

```
#I[1]:: r:{a,b,c}

#O[1]:  {a,b,c}

#I[2]:: Map[f,r]

#O[2]:  {f[a],f[b],f[c]}

#I[3]:: s:a^2+b+c(c+1)

                          2
#O[3]:  b + c (1 + c) + a

#I[4]:: Map[f,s]

                                      2
#O[4]:  f[b] + f[c (1 + c)] + f[a ]

#I[5]:: s^2

                             2 2
#O[5]:  (b + c (1 + c) + a )

#I[6]:: Map[$x^2,s]

          2          2    4    2
#O[6]:  c  (1 + c)  + a  + b

#I[7]:: Map[f,s,Inf]

                                                    f[2]
#O[7]:  f[b] + f[f[c] f[f[1] + f[c]]] + f[f[a]      ]
```

When the "template" is a single symbol, such as **f** in lines 2 and 4, the application of the template to *exp* by **Map** consists in forming **f[***exp***]**. In application of a template containing a generic symbol, as on line 6, the expression *exp* is inserted in place of the generic symbol.

On line 4, **Map** applied the template **f** to each subexpression in the "first level" of **s**. The *n*th "level" in an expression is in general defined to be the set of all parts which may be extracted by projection with *n* filters. Thus the "first level" in **s** consists of the arguments of **Plus**. By default, **Map** treats only the first level in an expression. In line 7, however, **Map** was explicitly specified to treat an infinite number of levels, so that **f** was applied to all subparts of **s**.

In general, it is possible to treat a "domain" containing an arbitrary set of subparts of an expression. The parts to be included in a domain are determined by their levels and by a "criterion" which they must satisfy, as described in [2.5].

Domains may also be specified directly for projections such as **Fac** and **Cb**, as described in the Reference Manual.

It is often convenient to apply some operation or simplification automatically to each output expression. Any value assigned to the global object **Post** is applied to every output expression before it is printed. To obtain all expressions in expanded form **Post::Ex**. To obtain a numerical value for each output expression **Post::N**.

```
#I[1]::  (1+x)(1-x)

#O[1]:   (1 - x) (1 + x)

#I[2]::  Post::Ex

#O[2]:   ´ Ex

#I[3]::  @1

                 2
#O[3]:   1 - x

#I[4]::  a(a+1)^3

                   2       3     4
#O[4]:   a + 3 a  + 3 a  + a

#I[5]::  Post::N

#O[5]:   ´ N

#I[6]::  Sin[1]+Gamma[4.5]

#O[6]:   12.4732

#I[7]::  Post:

#I[8]::  Sin[1]+Gamma[4.5]

#O[8]:   Gamma[4.5] + Sin[1]
```

Further examples

Example 1

Express $(2-a)\,(a+\sin(bx))^2\,(1+a^2+3\sin(bx))$ as a polynomial in $\sin(bx)$.

```
#I[1]::  (2-a)(a+Sin[b x])^2(1+a^2+3Sin[b x])

                                  2        2
#O[1]:   (2 - a) (a + Sin[b x])  (1 + a  + 3Sin[b x])

#I[2]::  Ex[%]

                    2              3                  2          2
#O[2]:   11a Sin[b x]  - 3a Sin[b x]  + 4a Sin[b x] - 4 a  Sin[b x]

                  2              3          2    3
         + 4 a  Sin[b x] - a  Sin[b x]  + a  Sin[b x]

                4              2    3      4    5            2
         - 2 a  Sin[b x] + 2 a  - a  + 2 a  - a  + 2 Sin[b x]

                    3
         + 6 Sin[b x]

#I[3]::  Cb[%,Sin[b x]^$n]

                                    3
#O[3]:   4a Sin[b x] + (6 - 3a) Sin[b x]

                      2    3          2      2
         + (2 + 11a - 4 a  - a ) Sin[b x]  + 4 a  Sin[b x]

            3              4              2    3      4    5
         + a  Sin[b x] - 2 a  Sin[b x] + 2 a  - a  + 2 a  - a

#I[4]::  Cb[%,Sin[b x]]

                    3                  2    3          2
#O[4]:   (6 - 3a) Sin[b x]  + (2 + 11a - 4 a  - a ) Sin[b x]

                      2    3      4                2    3      4    5
         + (4a + 4 a  + a  - 2 a ) Sin[b x] + 2 a  - a  + 2 a  - a
```

Example 2

Find the residues at the poles of $\dfrac{1+ax+3x^2}{x(x+1)(x-2)}$.

```
#I[1]::  (1+a x+3x^2)/(x(x+1)(x-2))

                      2
           1 + a x + 3 x
#O[1]:   ------------------
         x (-2 + x) (1 + x)

#I[2]::  Pf[%,x]

         -1    4/3 - a/3    13/6 + a/3
#O[2]:   -- + --------- + ----------
         2x       1 + x        -2 + x
```

Example 3

Find the coefficient of x^2 in the second derivative of $(1+x^2)\,(1-ax)^3$.

```
#I[1]:: D[(1+x^2)(1-a x)^3,x,x]

                                  2          2                2          3
#O[1]:  -12a x (1 - a x)  + 6 a  (1 - a x) (1 + x ) + 2 (1 - a x)

#I[2]:: Ex[%]

                          2  2       3          3  3       2
#O[2]:  2 - 18a x + 36 a  x  - 6 a  x - 20 a  x  + 6 a

#I[3]:: Coef[x^2,%]

                2
#O[3]:  36 a
```

Example 4

Find the 5^{th} iterant of the map $x \to (1-\lambda x)^2$ at $x \to 1$ and $\lambda \to \frac{1}{2}$.

```
#I[1]:: r:$s->(1-lam $s)^2

                          2
#O[1]:  $s -> (1 - lam $s)

#I[2]:: S[x,r,5]

                                                          2 2 2 2 2
#O[2]:  (1 - lam (1 - lam (1 - lam (1 - lam (1 - lam x) ) ) ) )

#I[3]:: S[%,x->1,lam->1/2]

                          2
                 26527/32768  2
#O[3]:  (1 - ------------)
                      2

#I[4]:: N[%]

#O[4]:  0.452018
```

6. Mathematical operations

In written mathematics, it is commonplace to use shortened notation and to infer the omitted details from convention or context of usage. For example, the indefinite integral $\int x\ dx$ is usually taken as $\frac{x^2}{2} = \int_0^x y\ dy$, while $\int \frac{dx}{x}$ is usually taken as $\log(x) = \int_1^x \frac{dy}{y}$. Similarly, the notation $f'(x^2)$ may mean either

$\left.\frac{df(y)}{dy}\right|_{y=x^2}$ or $\left.\frac{df(y^2)}{dy}\right|_{y=x}$. Without further information, SMP could not resolve these ambiguities. Mathematical operations in SMP must be defined precisely. General definitions may be used to introduce particular short notations.

```
#I[1]::  Int[x,x]

                2      2
               x      #1
#O[1]:         -- - ----
               2       2

#I[2]::  Int[1/x,x]

#O[2]:   Log[x] - Log[#2]

#I[3]::  Int[x,{x,a,b}]

                  2     2
               - a     b
#O[3]:         ---- + --
                2     2

#I[4]::  Int[f[x],{x,0,1}]

#O[4]:   Int[f[#3],{#3,0,1}]

#I[5]::  Int[f[x],x]

#O[5]:   Int[f[#5],{#5,#4,x}]
```

In the "indefinite" integrals on lines 1 and 2, SMP introduced the internally-generated symbols **#1** and **#2** to represent the lower limit of integration; the upper limit was taken as **x**. On line 3, the upper and lower limits were specified explicitly as **b** and **a**.

In line 4, the definite integral of the undefined function **f** was requested. The result was simply a canonical form of the integral, with **x** replaced by the "dummy" variable **#3**. On line 5, the canonical form for the indefinite integral involved both a dummy variable **#5** and a lower limit of integration **#4**.

The definite integral of **f** required in line 4 may be defined to be **c** by the direct assignment **Int[f[$x],{$x,0,1}]:c**.

```
#I[1]::  Int[f[$x],{$x,0,1}]:c

#O[1]:   c

#I[2]::  Int[f[x]+x,{x,0,1}]

#O[2]:   1/2 + c

#I[3]::  Int[f[$x],{$x,$1,$2}]:cf[$2-$1]

#O[3]:   cf[-$1 + $2]
```

```
#I[4]::   Int[f[x],x]

#O[4]:    cf[x - #2]
```

On line 3, the indefinite integral of **f** was defined. As usual, the patterns are organized so that a definite integral of **f[x]** from **0** to **1** would use the specific assignment on line 1 rather than the more general one of line 3.

Int yields explicit results for the majority of rational, logarithmic, exponential and trigonometric expressions whose indefinite integrals lie in the same class of expressions*. To avoid generation of potentially huge expressions, **Int** does not automatically expand the integrand. Explicit use of **Ex** may therefore be required to cast the integrand into a suitable form for integration.

```
#I[1]::   Int[(1+x^2)/(1+x^3),x]

                               1/2                                1/2
                -1 + 2x - (-3)                    -1 + 2#1 - (-3)
          5Log[-------------------]     5Log[--------------------]
                               1/2                                1/2
                -1 + 2x + (-3)                    -1 + 2#1 + (-3)        2Log[1 + x]
#O[1]:    ------------------------  -  -------------------------  +  -----------
                      1/2                               1/2                3
                 6 (-3)                            6 (-3)

                                                 2                        2
                 2Log[1 + #1]     Log[1 - x + x ]     Log[1 - #1 + #1 ]
               - ------------  +  ---------------  -  -----------------
                       3                 6                    6

#I[2]::   Int[x(1+x),x]

#O[2]:    Int[#3 (1 + #3),{#3,#2,x}]

#I[3]::   Ex[%]

           2     3     2      3
          x     x    #2     #2
#O[3]:    -- +  -- - --- -  ---
          2     3     2      3

#I[4]::   Int[1/(1+x^2+x^3)^2,{x,0,1}]

                         20/31 + 2#4/31
#O[4]:    7/93 + Int[---------------,{#4,0,1}]
                              2      3
                         1 + #4  + #4
```

The integral in line 1, while explicit, is rather complicated. The integral on line 4 was not completely evaluated by the built-in integrator; the part not done is however given in its simplest canonical form.

Definite integrals for which an explicit result is not found may be evaluated numerically using **N** if their value is a single number.

```
#I[1]::   Int[Gamma[x],{x,1,4}]

#O[1]:    Int[Gamma[#1],{#1,1,4}]

#I[2]::   N[%]

#O[2]:    5.85236
```

---

* All such integrals could be performed explicitly if it were possible to give an explicit solution for an arbitrary polynomial equation.

We next discuss the slightly more involved case of differentiation.

```
#I[1]::  D[x^a,x]

                -1 + a
#O[1]:   a x
#I[2]::  D[f[x],x]
#O[2]:   D[f[#1],{#1,1,x}]
#I[3]::  D[%,x]
#O[3]:   D[f[#1],{#1,2,x}]
#I[4]::  D[f[g[x]],x]
#O[4]:   D[f[#1],{#1,1,g[x]}] D[g[#1],{#1,1,x}]
#I[5]::  D[f[x,x],x]
#O[5]:   D[f[x,#2],{#2,1,x}] + D[f[#1,x],{#1,1,x}]
```

The expression on line 2 represents $\left.\frac{d^1 f(\#1)}{d\#1^1}\right|_{\#1=x}$. **#1** is a dummy variable of differentiation; the complete result is a function only of the point **x** at which the derivative is evaluated. The form in line 3 represents the second derivative. In line 4, the chain rule is used to cast the derivative into a canonical form. The "function" $f(z_1,z_2)$ depends on two independent "variables" z_1 and z_2. The derivative of this function with respect to x along the curve $z_1=z_2=x$ is evaluated on line 5, yielding the result $\left.\frac{\partial f(z_1,z_2=x)}{\partial z_1}\right|_{z_1=x} + \left.\frac{\partial f(z_1=x,z_2)}{\partial z_2}\right|_{z_2=x}$.

Derivatives need not always be taken with respect to symbols: any expression containing only one distinct symbol may be used.

```
#I[1]::  D[f[x],g[x]]

         D[f[#1],{#1,1,x}]
#O[1]:   -----------------
         D[g[#1],{#1,1,x}]
#I[2]::  D[f[a x^2],Log[x]]

             2                    2
#O[2]:   2a x  D[f[#1],{#1,1,a x }]
#I[3]::  S[%,f->Sin]

             2        2
#O[3]:   2a x  Cos[a x ]
#I[4]::  D[Sin[a x^2],Log[x]]

             2        2
#O[4]:   2a x  Cos[a x ]
#I[5]::  D[%,x,x,x]

                2          2        3  3        2       4  5        2
#O[5]:   2(-24 a  x Sin[a x ] - 36 a  x  Cos[a x ] + 8 a  x  Sin[a x ])
#I[6]::  D[@4,{x,3}]

                2          2        3  3        2       4  5        2
#O[6]:   2(-24 a  x Sin[a x ] - 36 a  x  Cos[a x ] + 8 a  x  Sin[a x ])
```

The arbitrary function **f** in the derivative on line 2 was specified to be **Sin** on line 3: the result is the same as that obtained on line 4 where **f** was replaced by **Sin** before the derivative was taken. Lines 5 and 6 show two equivalent forms which yield the third derivative of **@4**.

Derivatives may be defined by direct assignments, just like integrals.

```
#I[1]::  D[f[$x],{$x,1,$y}]:g[$y]

#O[1]:   g[$y]

#I[2]::  D[f[x],x]

#O[2]:   g[x]

#I[3]::  D[f[x^2] x,x]

                 2          2     2
#O[3]:   f[x ] + 2 x  g[x ]

#I[4]::  D[%,x]

                   2          3                    2
#O[4]:   6x g[x ] + 4 x  D[g[#1],{#1,1,x }]

#I[5]::  D[h[$x,$y],{$x,1,1}]:c[$y]

#O[5]:   c[$y]

#I[6]::  D[h[x^2,y-1](x+3)^2,{x,1,1}]

#O[6]:   32c[-1 + y] + 8h[1,-1 + y]
```

Line 1 defined the "function" **g** to be the derivative of the function **f**. In line 4, the second derivative of **f** was required, and was written as a derivative of **g**. On line 5,

$\left.\frac{\partial h(x,y)}{\partial x}\right|_{x=1}$ was defined to be $c(y)$.

In the partial derivatives represented by **D**, all distinct symbols are assumed independent. **Dt** represents a total derivative, in which all symbols are assumed interdependent, unless they are defined to be constants (by *symb_***:Const** [4]), or are explicitly defined to have zero total derivatives.

```
#I[1]::  t:Pi x^a

              a
#O[1]:   x  Pi

#I[2]::  D[t,x]

                -1 + a
#O[2]:   a x         Pi

#I[3]::  Dt[t,x]

                    -1 + a     a
#O[3]:   Pi (a x          + x  Dt[a,x] Log[x])

#I[4]::  Dt[t]

                    -1 + a           a
#O[4]:   Pi (a x          Dt[x] + x  Dt[a] Log[x])
```

On line 4, the total differential of **t** was given.

Ps[*expr***,***x***,***x0***,***n***]** yields the power (Taylor-Laurent) series of *expr* with respect to *x* about the point *x=x0* to *n* terms.

```
#I[1]::  t:Ps[x Exp[x],x,0,4]

                      3    4    5
             2       x    x    x
#O[1]:*  x + x  + 1 + -- + -- + --
                     2    6    24

#I[2]::  t^2+1/t

                                2       3
          -1                 5 x     49 x
#O[2]:*  x   - 1 + x/2 + 1 + ---- + -----
                            6       24

#I[3]::  Ax[@2]

                              2       3
                     1     5 x     49 x
#O[3]:   -1 + x/2 + - + 1 + ---- + -----
                    x      6       24

#I[4]::  @2^2

                                      2
          -2        -1                8 x
#O[4]:*  x   - 2 x    + 2 + 2x/3 + 1 + ----
                                      3

#I[5]::  @3^2

                              2       3
                     1     5 x     49 x   2
#O[5]:   (-1 + x/2 + - + 1 + ---- + -----)
                    x      6       24
```

As mentioned in sect. 3 above, the star at the beginning of the output on line 1 indicates that the result is represented* and treated in a special manner: in manipulations such as line 2, the power series is truncated as necessary. The projection **Ax** used on line 3 converts the power series into an ordinary polynomial, whose square in line 5 is left unexpanded and untruncated.

The assignment of power series expansions for functions is described in [9.5]; if no expansion has been assigned, information on derivatives is used if appropriate.

Essential singularities are factored out of power series when they are encountered.

The specification of multi-variate power series is described in [9.5].

In addition to ordinary power series, [9.5] describes the projections **Ra** and **Cf** used to obtain rational (Pade) and continued fraction approximations.

Lim[*expr***,***x***,***x0***]** yields the limit of *expr* when *x* tends to *x0*.

```
#I[1]::  Lim[Sin[x]/x,x,0]

#O[1]:   1

#I[2]::  Lim[Cos[x]/x,x,0]

          -1
#O[2]:*  x   - x/2
```

As revealed on line 2, **Lim** in general yields a power series.

---

* The parts in the representation are not manifest from the output form: the projection **Lpr[***expr***]** prints *expr* in an explicit form.

Sum[*expr***,{***i***,***imin***,***imax***}]** yields $\sum_{i=imin}^{imax} expr$; **Prod** yields the corresponding product.

```
#I[1]::  Prod[f[i],{i,3,10}]

#O[1]:   f[3] f[4] f[5] f[6] f[7] f[8] f[9] f[10]

#I[2]::  Sum[i^2,{i,1,20}]

#O[2]:   2870
```

An equation *expr1*=*expr2* is automatically simplified by linear elimination of parameters. If *expr1* and *expr2* are determined to be identical, it yields **1**; if they are found unequal, it gives **0**. If possible, an explicit solution or solutions for a parameter *x* appearing in an equation *eqn* is found by **Sol[***eqn***,***x***]**.

```
#I[1]::  2x+3a-5=6x-4a+3

#O[1]:   7a = 8 + 4x

#I[2]::  Sol[%,x]

                 -8 + 7a
#O[2]:   {x -> -------}
                    4

#I[3]::  Sol[x^2-3x+7=0,x]

                        1/2                     1/2
                3 - (-19)           3 + (-19)
#O[3]:   {x -> -------------,x -> -------------}
                      2                   2

#I[4]::  N[%]

#O[4]:*  {x -> 1.5 - 2.17945I,x -> 1.5 + 2.17945I}

#I[5]::  Sol[Gamma[x]=x,x]

#O[5]:   {Gamma[x] = x}

#I[6]::  Sol[{x+2y=3a,x-5y=a},{x,y}]

#O[6]:   {{x -> 17a/7,y -> 2a/7}}
```

Linear elimination was automatically performed on the equation in line 1: an explicit solution for **x** was obtained on line 2, and given as a replacement. Similar replacements were obtained for the two solutions to the quadratic equation on line 3; complex numerical values of these roots were obtained on line 4. The transcendental equation on line 5 could not be solved or simplified, and the result was an equation, rather than a replacement representing an explicit solution. Line 6 is an example of solution to a list of simultaneous equations.

It is often convenient to summarize a set of mathematical operations in an "operator". The templates mentioned in sect. 5 may be used to represent operators. A generic symbol indicates the positions in the template at which the operand of the operator is to be inserted. Hence **D[$1,x]+$1** represents the operator $\partial_x + 1$. **Ap[***temp***,{***expr***}]** applies the operator *temp* to the expression *expr*.

Further examples

Example 1
Find the value of $\int_0^1 e^{x^3} dx$.

```
#I[1]::   Int[Exp[x^3],{x,0,1}]

                      3
#O[1]:    Int[Exp[#1 ],{#1,0,1}]

#I[2]::   N[%]

#O[2]:    1.3419
```

Example 2
Define a function to obtain Legendre polynomials using Rodrigues's formula:

$$P_l(x) = \frac{1}{2^l l!} \frac{d^l}{dx^l} (x^2-1)^l$$

```
#I[1]::   p[$l,$x]::Ex[1/(2^$l $l!)D[(z^2-1)^$l,{z,$l,$x}]]

                 1          2      $l
#O[1]:    ´ Ex[-------- D[(z  - 1)   ,{z,$l,$x}]]
               $l
             2   $l!

#I[2]::   p[0,x]

#O[2]:    1

#I[3]::   p[2,x]

                      2
                   3 x
#O[3]:    -1/2 + ----
                   2

#I[4]::   p[10,x]

                          2           4           6           8           10
                    3465 x    15015 x    45045 x    109395 x    46189 x
#O[4]:    -63/256 + ------- - -------- + -------- - --------- + ---------
                      256        128         128         256         256
```

Example 3
Find the gradient of the function $f(x)=2x^2+3x+4xy-5y^2+7$, and deduce the positions of any saddle points.

```
#I[1]::   2x^2+3x+7+4x y-5y^2

                                2      2
#O[1]:    7 + 3x + 4x y + 2 x  - 5 y

#I[2]::   {D[%,x],D[%,y]}

#O[2]:    {3 + 4x + 4y,4x - 10y}

#I[3]::   Sol[%,{x,y}]

#O[3]:    {{x -> 3/4,y -> -3/10}}
```

Example 4
Find the power series expansion of $r\ e^{r^2}$ where $r=1+x+x^2$ about the point $x=0$ to fifth order.

```
#I[1]::  r:1+x+x^2

                   2
#O[1]:   1 + x + x

#I[2]::  Ps[r Exp[r^2],x,0,5]

                                    3          4          5
                        2      49 x   E    61 x   E   1523 x   E
#O[2]:*  E + 3x E + 8 x   E + ------- + ------- + ----------
                                  3          2          30
```

Example 5
Calculate the Bernoulli numbers up to B_{10} using the generating function:

$$\frac{t}{e^t-1} = \sum_{n=0}^{\infty} B_n \frac{t^n}{n!}$$

```
#I[1]::  Ps[t/(Exp[t]-1),t,0,11]

                      2    4       6                          8                    10
                     t    t       t                          t                    t
#O[1]:*  1 - t/2 + -- - --- + ----- - 8.26720*^-7 t  + 2.08768*^-8 t
                     12   720   30240

#I[2]::  Ax[%]

                      2    4       6                          8                    10
                     t    t       t                          t                    t
#O[2]:   1 - t/2 + -- - --- + ----- - 8.26720*^-7 t  + 2.08768*^-8 t
                     12   720   30240

#I[3]::  S[%,t^$n->t^$n $n!]

                      2    4    6    8       10
                     t    t    t    t     5 t
#O[3]:   1 - t/2 + -- - -- + -- - -- + -----
                     6    30   42   30    66

#I[4]::  S[%,Plus->List,t->1,Inf]

#O[4]:   {1,-1/2,1/6,-1/30,1/42,-1/30,5/66}
```

Example 6
Calculate the exponential constant e from fourth order power series and continued fraction approximations to e^x.

```
#I[1]::  Ps[Exp[x],x,0,4]

                     2    3    4
                    x    x    x
#O[1]:*  1 + x + -- + -- + --
                    2    6    24

#I[2]::  S[Ax[%],x->1]

#O[2]:   65/24
```

```
#I[3]:: N[%]

#O[3]:  2.70833

#I[4]:: Cf[Exp[x],x,0,4]

                1    -1 x 1/2 x -1/6 x 1/6 x
#O[4]:*  1 + ---- + ---- + ----- + ------ + -----
             1 +    1 +    1 +     1 +    1 +

#I[5]:: #I[2]

#O[5]:  19/7

#I[6]:: N[%]

#O[6]:  2.71429

#I[7]:: N[E]

#O[7]:  2.71828
```

Example 7

Define the derivatives of the Γ function using the relations:

$$\frac{d}{dx}\Gamma(x) = \psi^{(0)}(x)\Gamma(x)$$

$$\frac{d}{dx}\psi^{(n)}(x) = \psi^{(n+1)}(x)$$

```
#I[1]:: D[Gamma[$x],{$x,1,$y}]:Psi[1,$y] Gamma[$y]

#O[1]:  Gamma[$y] Psi[1,$y]

#I[2]:: D[Gamma[Exp[x^2]],x]

                    2                  2                   2
#O[2]:  2x Exp[x ] Gamma[Exp[x ]] Psi[1,Exp[x ]]

#I[3]:: D[Psi[$n,$x],{$x,1,$y}]:Psi[$n+1,$y]

#O[3]:  Psi[1 + $n,$y]

#I[4]:: D[Gamma[x],x,x,x]

                                  3
#O[4]:  Gamma[x] Psi[1,x] + Gamma[x] Psi[3,x]
              + 3Gamma[x] Psi[1,x] Psi[2,x]
```

7. Presentation of results

Many results are best presented in the form of tables or graphs. SMP allows tables and graphs not only to be generated, but also to be manipulated as symbolic expressions in their own right.

Lists may be considered as tables. For example, values of a function at a set of points may be tabulated in a list whose indices correspond to the points. The external file **XPrtable** formats tables for printing.

Ar[*n***,***temp***]** yields a contiguous list whose entries have indices **1**, ..., *n* and values obtained by application of the "template" *temp* (see sect. 5) to these indices.

```
#I[1]::  Ar[5,f]
#O[1]:   {f[1],f[2],f[3],f[4],f[5]}
#I[2]::  Ar[5,Log]
#O[2]:   {0,Log[2],Log[3],Log[4],Log[5]}
#I[3]::  N[%]
#O[3]:   {0,0.693147,1.09861,1.38629,1.60944}
#I[4]::  Ar[5,$x^2]
#O[4]:   {1,4,9,16,25}
#I[5]::  Ar[{{0,1.2,0.1}},f]
#O[5]:   {[0]: f[0], [0.1]: f[0.1], [0.2]: f[0.2], [0.3]: f[0.3],
               [0.4]: f[0.4], [0.5]: f[0.5], [0.6]: f[0.6],
               [0.7]: f[0.7], [0.8]: f[0.8], [0.9]: f[0.9], [1]: f[1],
               [1.1]: f[1.1], [1.2]: f[1.2]}
#I[6]::  Ar[{{0,1.2,0.1}},$x+$x^2]
#O[6]:   {[0]: 0, [0.1]: 0.11, [0.2]: 0.24, [0.3]: 0.39, [0.4]: 0.56,
               [0.5]: 0.75, [0.6]: 0.96, [0.7]: 1.19, [0.8]: 1.44,
               [0.9]: 1.71, [1]: 2, [1.1]: 2.31, [1.2]: 2.64}
#I[7]::  %^2+0.6
#O[7]:   {[0]: 0.6, [0.1]: 0.6121, [0.2]: 0.6576, [0.3]: 0.7521,
               [0.4]: 0.9136, [0.5]: 1.1625, [0.6]: 1.5216,
               [0.7]: 2.0161, [0.8]: 2.6736, [0.9]: 3.5241, [1]: 4.6,
               [1.1]: 5.9361, [1.2]: 7.5696}
```

In line 5, the indices in the list were specified to run from **0** to **1.2** in steps of **0.1**. Notice that in line 7, the specified operation was performed on each element of the list, and a new transformed table was generated.

The tables in this example are analogous to vectors. **Ar** may also be used to generate tables in which each element carries a set of indices, as in a matrix or tensor.

```
#I[1]::  Ar[{2,3},f]
#O[1]:   {{f[1,1],f[1,2],f[1,3]},{f[2,1],f[2,2],f[2,3]}}
```

```
#I[2]::  Ar[{2,3},f[$i,$j^2]]

#O[2]:   {{f[1,1],f[1,4],f[1,9]},{f[2,1],f[2,4],f[2,9]}}

#I[3]::  Ar[{2,3},1/($i+$i^2+$j)]

#O[3]:   {{1/3,1/4,1/5},{1/7,1/8,1/9}}

#I[4]::  Ar[{{0,1,0.3},{4,5}},f]

#O[4]:   {[0]: {[4]: f[0,4], [5]: f[0,5]},
                [0.3]: {[4]: f[0.3,4], [5]: f[0.3,5]},
                [0.6]: {[4]: f[0.6,4], [5]: f[0.6,5]},
                [0.9]: {[4]: f[0.9,4], [5]: f[0.9,5]}}

#I[5]::  Ar[{{0,1,0.3},{4,5}},N[Log[$x+$y^2]]]

#O[5]:   {[0]: {[4]: 2.77259, [5]: 3.21888},
                [0.3]: {[4]: 2.79117, [5]: 3.2308},
                [0.6]: {[4]: 2.8094, [5]: 3.24259},
                [0.9]: {[4]: 2.82731, [5]: 3.25424}}

#I[6]::  %^2

#O[6]:   {[0]: {[4]: 2.77259^2, [5]: 3.21888^2},
                [3/10]: {[4]: 2.79117^2, [5]: 3.2308^2},
                [3/5]: {[4]: 2.8094^2, [5]: 3.24259^2},
                [9/10]: {[4]: 2.82731^2, [5]: 3.25424^2}}

#I[7]::  Ar[{2,2,2},f]

#O[7]:   {{{f[1,1,1],f[1,1,2]},{f[1,2,1],f[1,2,2]}},
                {{f[2,1,1],f[2,1,2]},{f[2,2,1],f[2,2,2]}}}
```

In line 1, **Ar** was used to generate a 2×3 matrix, each of whose entries was given by applying **f** to its two indices. On line 2, a 2×3 matrix whose i,jth element is $f(i,j^2)$ was obtained by applying the template **f[$i,$j^2]** to the indices for each element. When a template containing several generic symbols is applied to a set of expressions, the generic symbol which appears first in lexical ordering is associated with the first expression, the second with the second expression, and so on.

In line 7 above, a rank-3 2×2×2 tensor whose i,j,kth element is $f(i,j,k)$ was generated.

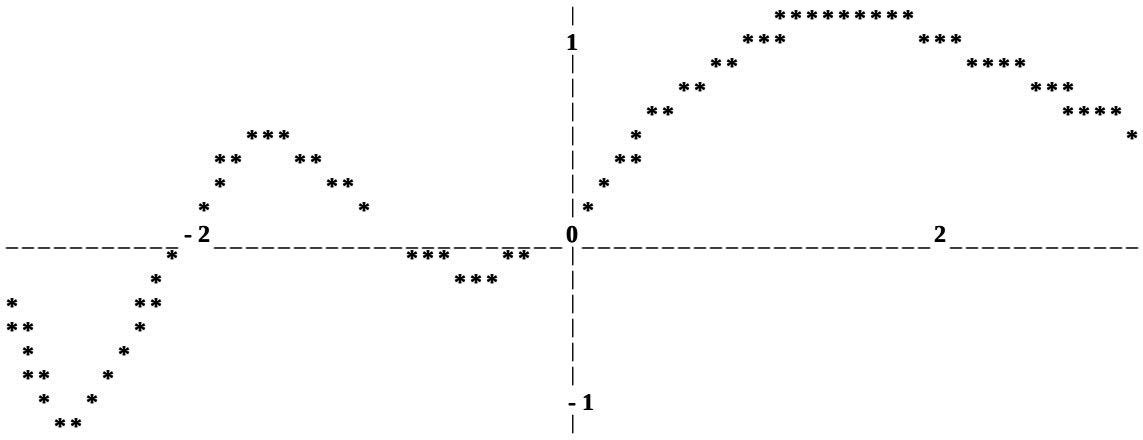
In some cases, it suffices to generate and print a table once, without retaining a permanent record of its elements in the form of a list. **Do[***i***,***imin***,***imax***,***istep***,***expr***]** evaluates *expr* with *i* equal to *imin*, *imin*+*istep*, ..., *imax*. If *istep* is omitted, it is taken as **1**, and if in addition, *imin* is omitted, it also takes the value **1**. **Pr[***expr1***,***expr2***,...]** prints a sequence of expressions.

```
#I[1]:  Do[i,5,Pr[i,i!,i!!,i^i]]

1       1       1       1
2       2       2       4
3       6       3       27
4       24      8       256
5       120     15      3125

#O[1]:  3125
```

```
#I[2]:  Do[u,x+1,x+5,Pr[u,Ex[u^2]]]

                    2
1 + x   1 + 2x + x
                    2
2 + x   4 + 4x + x
                    2
3 + x   9 + 6x + x
                     2
4 + x   16 + 8x + x
                      2
5 + x   25 + 10x + x

                       2
#O[2]:  25 + 10x + x
```

Numerical results may be presented as graphs. **Graph[***expr***,***x***,***xmin***,***xmax***]** plots a graph of the numerical value of *expr* with *x* between *xmin* and *xmax*.

```
#I[1]:  Graph[1/Gamma[x],x,-3,3]

#O[1]:
```



At some installations, graphics output devices may be available for which graphs are automatically generated with continuous high-resolution lines.

```
#I[1]:=  Graph[1/Gamma[x],x,-3,3]
```

Graph[{{*expr1***},{***expr2***},...},***x***,***xmin***,***xmax***]** plots the curves for *expr1*, *expr2*, ... on a single set of axes.

```
#I[1]:: Graph[{{Besj[2,x]},{Besj[5,x]}},x,0,20]
```

Graph[{*x***,***y***},***u***,***umin***,***umax***]** yields a parametric plot of *x,y* with the parameter *u* in the range *umin* to *umax*.

```
#I[1]:: Graph[{Sin[t]/Sqrt[t],Cos[t]/Sqrt[t]},t,0,30]
```

Graph[*z***,{***x***,***y***},{***xmin***,***ymin***},{***xmax***,***ymax***}]** gives a contour plot of *z* as a function of *x* and *y*.

```
#I[1]:: Graph[(4x+y)/5 Cos[x/5] Cos[y/5],{x,y},{0,0},{25,25}]
```

```
#I[1]:  Graph[(4x+y)/5 Cos[x/5] Cos[y/5],{x,y},
            {0,0},{25,25},,{-10,-20,0}]
```

Plots in SMP are symbolic expressions consisting of a set of point, line and surface specifications with special printing characteristics. Two, three and higher-dimensional graphs may thus be manipulated extensively, as indicated in the Reference Manual.

Further examples

Example 1

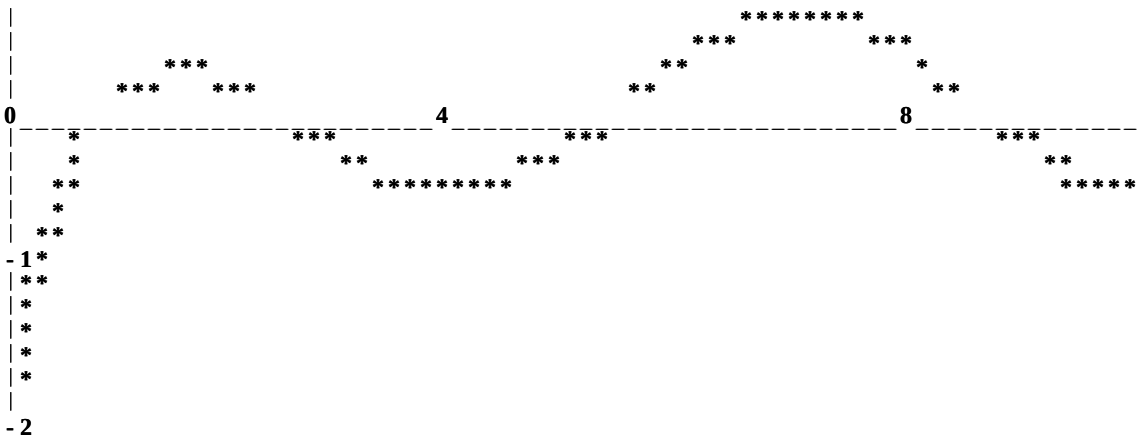
Form the 4×4 matrix $M_{ij}=\frac{1}{i+j+1}$. Find the inverse of M.

```
#I[1]:: Ar[{4,4},1/($i+$j+1)]
#O[1]:  {{1/3,1/4,1/5,1/6},{1/4,1/5,1/6,1/7},{1/5,1/6,1/7,1/8},
                {1/6,1/7,1/8,1/9}}
#I[2]:: Minv[%]
#O[2]:  {{1200,-6300,10080,-5040},{-6300,35280,-58800,30240},
                {10080,-58800,100800,-52920},{-5040,30240,-52920,28224}}
```

Example 2

Find graphically the approximate position of the second zero of $\log(x)\,J_0(x)$. Make tables of the values of the function to locate the zero more accurately.

```
#I[1]:: Graph[Log[x]BesJ[0,x],x,0,10]
#O[1]:
```




```
#I[2]:: t[$x]::N[Log[$x]BesJ[0,$x]]
#O[2]:  ´ N[Log[$x] BesJ[0,$x]]
```

```
#I[3]::  Ar[{{2.4,2.8,.025}},t]
#O[3]:   {[2.4]: 0.0021954, [2.425]: -0.00923844, [2.45]: -0.0208132,
              [2.475]: -0.0325159, [2.5]: -0.0443336,
              [2.525]: -0.0562534, [2.55]: -0.0682626,
              [2.575]: -0.0803484, [2.6]: -0.0924982, [2.625]: -0.1047,
              [2.65]: -0.11694, [2.675]: -0.129207, [2.7]: -0.141488,
              [2.725]: -0.153772, [2.75]: -0.166046,
              [2.775]: -0.178298, [2.8]: -0.190517}
#I[4]::  Ar[{{2.4,2.41,.001}},t]
#O[4]:   {[2.4]: 0.0021954, [2.401]: 0.00174091, [2.402]: 0.00128618,
              [2.403]: 8.31210*^-4, [2.404]: 3.75992*^-4,
              [2.405]: -7.94692*^-5, [2.406]: -5.35173*^-4,
              [2.407]: -9.91117*^-4, [2.408]: -0.0014473,
              [2.409]: -0.00190373, [2.41]: -0.00236039}
```

Example 3
Generate the totally antisymmetric (Levi-Civita) tensor in three dimensions.

```
#I[1]::  Ar[{3,3,3},Sig]
#O[1]:   {{{0,0,0},{0,0,1},{0,-1,0}},{{0,0,-1},{0,0,0},{1,0,0}},
              {{0,1,0},{-1,0,0},{0,0,0}}}
#I[2]::   Pos[1,%]
#O[2]:   {{1,2,3},{2,3,1},{3,1,2}}
```

Example 4
Form the unit lower-triangular 4×4 matrix.

```
#I[1]::  Ar[{4,4},Gt]
#O[1]:   {{0,0,0,0},{1,0,0,0},{1,1,0,0},{1,1,1,0}}
```

Example 5
Obtain a picture of an involution of the circle:

$$x = \cos(\phi) + \phi \sin(\phi)$$

$$y = \sin(\phi) - \phi \cos(\phi)$$

8. Defining new mathematical constructs

One of the most powerful features of SMP is the simplicity with which mathematical constructs may be introduced and manipulated. The SMP definition of a new construct usually coincides very closely with its precise mathematical definition.

As a first example, consider the introduction of a new differentiation projection **newdiff** which obeys the standard rules:

$$\frac{d}{dx}\,x = 1$$

$$\frac{d}{dx}\,y = 0 \qquad (y \neq x)$$

$$\frac{d}{dx}\,(u+v) = \frac{du}{dx} + \frac{dv}{dx}$$

$$\frac{d}{dx}\,(u\ v) = v\ \frac{du}{dx} + u\ \frac{dv}{dx}$$

where x and y are symbols.

```
#I[1]::  newdiff[$x,$x]:1
#O[1]:   1
#I[2]::  newdiff[$y_=Numbp[$y],$x]:0
#O[2]:   0
#I[3]::  newdiff[$y_=Symbp[$y],$x_=(~P[$x=$y])]:0
#O[3]:   0
#I[4]::  newdiff[$u+$$v,$x]::newdiff[$u,$x]+newdiff[$$v,$x]
#O[4]:   ´ newdiff[$u,$x] + newdiff[$$v,$x]
#I[5]::  newdiff[$u $$v,$x]::$$v newdiff[$u,$x]+$u newdiff[$$v,$x]
#O[5]:   ´ $$v newdiff[$u,$x] + $u newdiff[$$v,$x]
#I[6]::  newdiff
#O[6]:   {[$u $$v]:: {[$x]:: $$v newdiff[$u,$x] + $u newdiff[$$v,$x]},
                [$u + $$v]:: {[$x]:: newdiff[$u,$x] + newdiff[$$v,$x]},
                [$y _= Symbp[$y]]: {[$x _= ~ P[$x = $y]]: 0},
                [$y _= Numbp[$y]]: {[$x]: 0}, [$x]: {[$x]: 1}}
#I[7]::  newdiff[2x+3,x]
#O[7]:   2
#I[8]::  t:(x+a+b)(3/2x+2b)(4x+3)+x+a x
#O[8]:   x + a x + (3 + 4x) (2b + 3x/2) (a + b + x)
#I[9]::  newdiff[t,x]
#O[9]:   1 + a + (3 + 4x) (3a/2 + 7b/2 + 3x) + 4(2b + 3x/2) (a + b + x)
```

```
#I[10]::  Ex[%]

                                                                  2        2
#O[10]:   1 + 11a/2 + 21b/2 + 9x + 8a b + 12a x + 28b x + 8 b  + 18 x

#I[11]::  Ex[D[t,x]]

                                                                  2        2
#O[11]:   1 + 11a/2 + 21b/2 + 9x + 8a b + 12a x + 28b x + 8 b  + 18 x

#I[12]::  Put[newdiff,defdiff];
```

The rules for differentiation stated in mathematical form above were given as assignments for **newdiff** in lines 1 through 5. The resulting form for **newdiff** was displayed in line 6. Examples of **newdiff** were given on lines 7 and 9, the latter being compared with the result obtained with the built-in differentiation projection **D** on line 11. On line 12, the definitions for **newdiff** were saved in the file **defdiff**:

```
newdiff[$x] : {[$x]: 1}
newdiff[$y _= Numbp[$y]] : {[$x]: 0}
newdiff[$y _= Symbp[$y]] : {[$x _= ~ P[$x = $y]]: 0}
newdiff[$u + $$v,$x] :: newdiff[$u,$x] + newdiff[$$v,$x]
newdiff[$u*$$v,$x] :: $$v*newdiff[$u,$x] + $u*newdiff[$$v,$x]
```

On lines 2 and 3, **Numbp[***y***]** tests whether *y* is a number, and **Symbp[***y***]** tests whether it is a symbol. In line 3, **P[***expr***]** yields **1** if *expr* is **1** ("true") and **0** otherwise. Its use in this case implements the assumption that distinct symbols represent independent variables. The generic symbol **$u** in line 4 may represent either a composite expression or a single symbol and thus an independent variable. The alternative realized in a particular case depends on the explicit form of **$u** provided. To avoid evaluation before this form is specified, delayed assignments are used in lines 4 and 5.

On lines 4 and 5, **$$v** represents a sequence of arguments to **Plus** and **Mult**, and, through recursion, applies to derivatives of sums and products with any number of terms. As discussed in sect. 3, use of **$v** would give rules only for sums or products with exactly two terms.

Notice that in all assignments for **newdiff**, there was no explicit requirement that the variable of differentiation **$x** be a symbol. Such a requirement may be introduced by use of **$x_=Symbp[$x]** rather than **$x** on the left-hand side in each case.

In contrast to the rather direct definition for **newdiff** given above, conventional programming languages would require a definition such as:

```
#I[1]::   newdiff[$y,$x]::Sel[Symbp[$y],If[P[$y=$x],1,0],\
            Numbp[$y],0,$y[0]=Plus,newdiff[$y[1],$x]+newdiff[$y[2],$x],\
            $y[0]=Mult,$y[2]newdiff[$y[1],$x]+$y[1]newdiff[$y[2],$x]]

#O[1]:    ´ Sel[Symbp[$y],If[P[$y = $x],1,0],Numbp[$y],0,$y[0] = Plus,

                newdiff[$y[1],$x] + newdiff[$y[2],$x],$y[0] = Mult,

                $y[2] newdiff[$y[1],$x] + $y[1] newdiff[$y[2],$x]]
```

Notice that this assignment cannot treat sums and products containing more than two terms.

The "select" projection **Sel[***pred1***,***expr1***,***pred2***,***expr2***,...]** tests each of the "predicate expressions" *predi* in turn, yielding the *expri* associated with the first one found to be "true". **If[***pred***,***expr1***,***expr2***]** tests *pred*, yielding *expr1* if it is found to be "true" and *expr2* if it is found "false".

In the second example the function **newdiff** accepts any expression **$y** as a first argument, and then uses **Sel** and **If** to test the possible forms of **$y** and select an appropriate case. In the first example, however, individual assignment are given directly for each possible form, so that explicit **Sel** or **If** tests are unnecessary. The **$y[0]=Plus** case in the **Sel** of the second example is covered by an assignment for expressions of the form **$u+$$v** on line 4 of the first example. The terms in the sum are identified in the first example as **$u** and **$$v**. In the second example, they must be identified and extracted by the explicit

projections **$y[1]** and **$y[2]**.

In addition to its simplicity and directness, the approach used in the first example has the virtue that further definitions may be given as additional assignments. In the second example, the entire **Sel** must be rebuilt to accommodate the additional case.

For the next example, we consider a function $f(x)$ obeying the relations:

$$(a) \qquad f(n\pi) = -1^n \qquad (n \text{ integer})$$

$$(b) \qquad f(-x) = -f(x)$$

```
#I[1]:: f[($n_=Intp[$n]) Pi]:(-1)^$n

                $n
#O[1]:  (-1)

#I[2]:: f[3Pi]

#O[2]:  -1

#I[3]:: f[x]

#O[3]:  f[x]

#I[4]:: f[N[3Pi]]

#O[4]:  f[9.42478]

#I[5]:: f[$x_=Mod[$x,!N[Pi]]=0]:(-1)^($x/N[Pi])

                0.31831$x
#O[5]:  (-1)

#I[6]:: f[N[3Pi]]

#O[6]:  -1

#I[7]:: f[3]

#O[7]:  f[3]

#I[8]:: f[$x_=Nc[$x]<0]:-f[-$x]

#O[8]:  -f[-$x]

#I[9]:: f[-x]+f[y]

#O[9]:  -f[x] + f[y]
```

The assignment on line 1 implements relation (a) when **Pi** appears explicitly. The assignment on line 4 treats cases such as line 3 in which the argument of **f** is a real number divisible by **Pi**. The function **Mod[***x***,***y***]** yields the remainder from the division of *x* by *y*.

The definition on line 5 requires the numerical value of **Pi**. **N[Pi]** is evaluated immediately on the right-hand side of the assignment. However, in the condition on the left-hand side, **N[Pi]** would usually be re-evaluated whenever the condition is tested. **!***expr* causes *expr* to be simplified on input, and thus avoids re-evaluation in this case.

Line 8 uses the reflection formula (b) to produce a canonical form by removing any explicit overall minus sign in the argument of **f**. **Nc[***x***]** gives the overall numerical coefficient of the expression *x*. When no explicit overall minus sign is present, projections of **f** are left unchanged. If no condition had been given in the assignment on line 8, **f[x]** would have been replaced by **-f[-x]** which would in turn have been replaced by **f[x]** ad infinitum.

We now consider manipulation of permutations (see also the external file **XPerm**). Permutations are represented as usual by a list of integers giving the final positions of each element. The projection **comp[***perm1***,***perm2***]** yields the permutation obtained by applying first *perm1* then *perm2* (composition).

```
#I[1]:: comp[$p1,$p2]::Ar[Len[$p1],$p2[$p1[$1]]]
#O[1]:  ´ Ar[Len[$p1],$p2[$p1[$1]]]
#I[2]:: comp[{3,2,1},{1,3,2}]
#O[2]:  {2,3,1}
#I[3]:: comp[{2,3,1},{3,2,1},{1,3,2}]
#O[3]:  comp[{2,3,1},{3,2,1},{1,3,2}]
#I[4]:: comp[comp[{2,3,1},{3,2,1}],{1,3,2}]
#O[4]:  {3,1,2}
```

On line 1, **Len[***expr***]** gives the "length" of *expr*: the number of elements in a list, or the number of filters in a projection. On line 2, **$1** in the definition of **comp** runs over the values **1**, **2** and **3**. The inner projection in the template yields the position of each element after the first permutation; the outer projection then gives the final position after the element has been subjected to the second permutation.

comp is defined on line 1 as a function of two arguments only. The expression on line 3 is therefore not simplified. However, the mathematical operation of composition is associative, allowing composition of three permutations to be performed as on line 4. Notice that the arguments of the function **comp** on line 4 are simplified before the function itself.

Assignment of the property **Flat** causes **comp** to act as an associative *n*-ary function. This property may be used in evaluating the composition of three permutations.

```
#I[1]:: _comp[Flat]:1
#O[1]:  1
#I[2]:: comp[$p1,$p2]::Ar[Len[$p1],$p2[$p1[$1]]]
#O[2]:  ´ Ar[Len[$p1],$p2[$p1[$1]]]
#I[3]:: comp[{2,3,1},{3,2,1},{1,3,2}]
#O[3]:  {3,1,2}
#I[4]:: Put[comp,perms]
#O[4]:  {[[$p1,$p2]]:: Ar[Len[$p1],$p2[$p1[$1]]]}
#I[5]:: f[f[a,b],c]
#O[5]:  f[f[a,b],c]
#I[6]:: f_:Flat
#O[6]:  Flat
#I[7]:: f[f[a,b],c]
#O[7]:  f[a,b,c]
#I[8]:: _f
#O[8]:  {[Flat]: 1}
```

Line 6 defines **f** to have the property **Flat**. The nested projection on line 5 is then reduced to a "flattened" canonical form on line 7. The built-in functions **Plus**, **Mult** and **Dot** also carry the property **Flat**, so that **(a+b)+c** is reduced to **a+b+c**.

Any properties or attributes assigned to a symbol *s* are given in the list *_s*. When a symbol carries a property *prop*, a nonzero entry *_s*[*prop*] appears in the list *_s*. Line 8 shows that **f** carries the property **Flat**. The form *s_*:*prop* used on line 6 is a short notation for *_s*[*prop*]:1. Projections of *_s* may be assigned or removed just like projections of a symbol.

Properties are used to specify many characteristics of projections. Projections carrying the property **Comm** are treated as commutative functions, and their arguments are sorted into a canonical order. The property **Reor** may be used to specify more complicated reordering symmetries for the filters of a projection (as required for indicial tensors). For example, *_s*[**Reor**]:**Asym** defines *s* to be totally antisymmetric with respect to reordering of its arguments.

When a projection carrying the property **Ldist** has lists as its filters, the projection is applied separately to each element of the lists, yielding a list of the results. All common mathematical functions have this property, as illustrated in sect. 2.

a_b assigns the symbol *a* to have the "type" *b*, and to share any properties of *b*. Such assignments are often required to distinguish objects of different classes when using external files.

It is often convenient to represent several related functions as projections from the same object, distinguishing them by the number of arguments given. Thus both the ordinary gamma function $\Gamma(x)$ and the incomplete gamma function $\Gamma(x,a)$ are represented as projections of **Gamma**. Projections with fewer filters usually correspond to functions obtained when omitted arguments take on simple "default" values. The ordinary gamma function is obtained from $\Gamma(x,a)$ when $a=0$.

Assignment of the property **Tier** to a symbol *s* indicates that projections of *s* with different numbers of filters represent different functions. Without **Tier**, an additional filter in a projection represents extraction of a deeper subpart of the same function or object. **f[1,2]** is then equivalent to **f[1][2]**. When **f** has the property **Tier**, however, **f[1,2]** and **f[1][2]** are distinguished. Most built-in functions allow several optional arguments used to specify their action more precisely; they therefore carry the property **Tier**.

Composition may be considered as the analogue for permutations of the **Dot** operation. Lists used to represent permutations may be distinguished from other lists by enclosing them in a projection as **Perm**[*list*]. No value is assigned to **Perm**; it is merely used to identify permutation lists.

```
#I[1]::  Perm[$l1].Perm[$l2]::Perm[comp[$l1,$l2]]

#O[1]:   ´ Perm[comp[$l1,$l2]]

#I[2]::  Perm[{2,3,1}].Perm[{3,2,1}].Perm[{1,3,2}]

#O[2]:   Perm[comp[comp[{2,3,1},{3,2,1}],{1,3,2}]]

#I[3]::  <perms

#O[3]:   ´ Ar[Len[$p1],$p2[$p1[$1]]]

#I[4]::  @2

#O[4]:   Perm[{3,1,2}]

#I[5]::  _Perm[Pr][$x]::Fmt[{{0,0},{1,1},{1,-1},{2,0}},"[",$x,Ar[Len[$x]],"]"]

#O[5]:   ´ [ $x          ]
          [ Ar[Len[$x]] ]
```

```
#I[6]:: @4

          {3,1,2}
#O[6]:* [       ]
          {1,2,3}

#I[7]:: Lpr[%]

Perm[{3,1,2}]

#I[8]:: Perm[{2,1,3,4}].Perm[{1,4,2,3}]

          {4,1,2,3}
#O[8]:* [         ]
          {1,2,3,4}
```

Line 5 defines a special output form for **Perm**. When a suitable property is defined, projections from *f* are printed in the form assigned for the corresponding projections of **_*f*[Pr]**. **Fmt[***spec*,*expr1*,*expr2*,...**]** prints with the expressions *expr1*, *expr2*, ... in relative positions defined by the lists of horizontal and vertical offsets in *spec*. **Lpr** prints expressions without using special output forms, as on line 7.

If permutations are represented as ordinary lists, composition may be represented by a special input form.

```
#I[1]:: Sxset["<>",comp,4]

#O[1]:  comp

#I[2]:: {2,3,1}<>{3,2,1}<>{1,3,2}

#O[2]:  comp[{2,3,1},comp[{3,2,1},{1,3,2}]]

#I[3]:: <perms

#O[3]:  ´ Ar[Len[$p1],$p2[$p1[$1]]]

#I[4]:: @2

#O[4]:  {3,1,2}
```

Line 1 defines **<>** to be an operator of class 4 [2.11] representing **comp**.

External files in the SMP Library serve as further examples. See particularly the files **XLap** (Laplace transforms), **XGr** (graph theory) and **XFrac** (fractions).

9. Programming

Assignments may be considered to define programs. In most of the cases discussed above, the "programs" have involved only mathematical functions or single SMP commands. In some cases, it is necessary to use a sequence of commands to manipulate or transform an expression.

In the simplest case, the sequence is formed by nesting commands. The nested projection *f*`[`*g*`[`*x*`]]` performs the command *g* on the expression *x*, and then performs *f* on the result. This method is adequate when the output from each step is required only as the input to the next step.

A procedure consists of a sequence of independent commands. The commands are given in order, and separated by semicolons. Procedures are necessary for manipulations which require intermediate expressions to be used repeatedly. Such expressions may be assigned as the values of temporary "local" variables in a procedure. `Lcl[`*x1*`,`*x2*`,...]` defines *x1*, *x2*, ... to be local variables whose original values are restored upon completion of the procedure.

```
#I[1]::  Ex[(1+x+a)^3]

                                    2      2        2    3      2    3
#O[1]:   1 + 3a + 3x + 6a x + 3a x  + 3 a  x + 3 a  + a  + 3 x  + x

#I[2]::  Coef[x,%]

                      2
#O[2]:   3 + 6a + 3 a

#I[3]::  Fac[%]

                 2
#O[3]:   3 (1 + a)

#I[4]::  Fac[Coef[x,Ex[(1+x+a)^3]]]

                 2
#O[4]:   3 (1 + a)

#I[5]::  t:Ex[(1+x+a)^3];t:Coef[x,t];t:Fac[t]

                 2
#O[5]:   3 (1 + a)

#I[6]::  t

                 2
#O[6]:   3 (1 + a)

#I[7]::  t:Ex[(1+x+a)^3];t:Coef[x,t]+Coef[x^2,t];Fac[t]

#O[7]:   3(1 + a) (2 + a)

#I[8]::  t

                      2
#O[8]:   6 + 9a + 3 a

#I[9]::  Lcl[t];t:Ex[(1+x+a)^3];t:Coef[x^2,t]/Coef[x,t]

            3 + 3a
#O[9]:   -------------
                      2
         3 + 6a + 3 a

#I[10]::  t

                       2
#O[10]:   6 + 9a + 3 a
```

In lines 1, 2 and 3, a succession of commands were used on a single expression. On line 4, the commands were nested together. In line 5, the commands were performed sequentially in a procedure, using **t** to carry intermediate results. Note that even after the procedure has been completed, **t** retained the last intermediate value assigned to it in the procedure.

In the procedure on line 7, the intermediate result from **Ex** was used twice; to perform the same operation using nested commands would have required two separate invocations of **Ex**. On line 9, a similar procedure was executed, but with **t** defined as a local variable. The original value of **t** was then restored on exit from the procedure, as shown on line 10.

Variables used only in intermediate stages of a procedure should be defined as local to avoid potential conflicts with existing or future variables of the same name used outside the procedure. It is conventional for the names of local variables to start with **%**.

If a procedure contains conditional statements (**If** or **Sel**), the result of the procedure may be returned before the final expression in the procedure is executed. When **Ret[***expr***]** is encountered in a procedure, the procedure is terminated, and the result *expr* is returned.

A procedure may contain invocations of further procedures. In a set of nested procedures, a variable defined as local in an outer procedure is shared with all inner procedures in which it is not explicitly defined as local. **Ret[***expr***,***n***]** returns the value *expr* through *n* nested procedures.

It is often necessary to perform an operation repetitively a fixed number of times, or until a particular form is reached. Many SMP functions may be specified to apply an operation repeatedly. For example, **S[***expr***,***a***->***b***,***n***]** performs substitutions in *expr* using the replacement *a->b* successively until no further change occurs, or at most *n* times.

Loop[*cond***,***expr***]** simplifies *expr* repetitively until the condition *cond* ceases to be "true".

```
#I[1]::  S[x,$x->f[$x]]
#O[1]:   f[x]
#I[2]::  S[x,$x->f[$x],4]
#O[2]:   f[f[f[f[x]]]]
#I[3]::  t:x;n:0;Loop[n<4,t:f[t];Inc[n]];t
#O[3]:   f[f[f[f[x]]]]
#I[4]::  For[t:x;n:0,n<4,Inc[n],t:f[t]]
#O[4]:   f[f[f[f[x]]]]
#I[5]::  t:x;Do[i,4,t:f[t]]
#O[5]:   f[f[f[f[x]]]]
```

Inc[*x***]** increments the value of *x* by **1**.

A natural mathematical definition of the Fibonacci numbers is

$$f(n) = f(n-1) + f(n-2) \quad \text{(a)}$$

$$f(1) = f(2) = 1 \quad \text{(b)}$$

The evaluation of Fibonacci numbers using this definition requires nested "recursive" application of the function *f*. *f* is used repeatedly until the end conditions (b) are encountered.

```
#I[1]::  f[$n]:f[$n-1]+f[$n-2]
#O[1]:   f[-2 + $n] + f[-1 + $n]
```

```
#I[2]::  f[1]:f[2]:1
#O[2]:   1
#I[3]::  f[3]
#O[3]:   2
#I[4]::  f[10]
#O[4]:   55
#I[5]::  g[$n]::(g[$n]:g[$n-1]+g[$n-2])
#O[5]:   ´ g[$n] : g[$n - 1] + g[$n - 2]
#I[6]::  g[1]:g[2]:1
#O[6]:   1
#I[7]::  f
#O[7]:   {[1]: 1, [2]: 1, [$n]: f[-2 + $n] + f[-1 + $n]}
#I[8]::  g
#O[8]:   {[1]: 1, [2]: 1, [$n]:: g[$n] : g[$n - 1] + g[$n - 2]}
#I[9]::  g[5]
#O[9]:   5
#I[10]::  g
#O[10]:   {[5]: 5, [4]: 3, [3]: 2, [1]: 1, [2]: 1,
                [$n]:: g[$n] : g[$n - 1] + g[$n - 2]}
```

The function **g** on line 5 is defined to be a program which recursively computes values of **g** and then stores the result ("dynamic programming"). After the evaluation of **g[5]** the list **g** contains the explicit values for all the cases computed, as shown on line 10.

Many common definitions implicitly use recursion. For example, **Log[$x $$x]:Log[$x]+Log[$$x]** is applied recursively to **Log[a b c]** to yield first **Log[b c] + Log[a]** and then **Log[a] + Log[b] + Log[c]**.

Recursive assignments are used until end conditions are encountered. Use of the definition

$$f(n) = f(n+2) - f(n+1)$$

would miss the end conditions and cause the attempted evaluation of say *f(4)* to continue forever. **S** allows recursive replacements to be applied a fixed number of times, instead of continuing until end conditions are reached.

In investigating or verifying the operation of a definition or program, it is often convenient to assign *f*_**:Trace** so that each projection from *f* is printed before evaluation.

```
#I[1]::  Log[$x $$x]:Log[$x]+Log[$$x]
#O[1]:   Log[$x] + Log[$$x]
#I[2]::  Log_:Trace
#O[2]:   Trace
```

```
#I[3]:  Log[a b c d e]

Log[a b c d e]

Log[a]

Log[b c d e]

Log[b]

Log[c d e]

Log[c]

Log[d e]

Log[d]

Log[e]

#O[3]:  Log[a] + Log[b] + Log[c] + Log[d] + Log[e]
```

Many programs require arguments of particular types. For example, the Fibonacci program above allows only integer arguments. Such type restrictions are enforced by placing conditions on the input variables: **f[$n_=Natp[$n]]:f[$n-1]+f[$n-2]** defines the reduction of **f[$n]** only when **$n** is a positive integer.

Expressions are usually simplified whenever they appear. However, in assignments performed with **::**, the expression on the right hand side is not immediately simplified. It is simplified each time the assignment is used, after generic symbols have been replaced by the specific values required. Use of **::** in the definition of the Fibonacci function **g[$n]** above causes the assignment on the right-hand side to be performed separately for each value of **$n**. Whenever simplification of an expression or program on the right-hand side of an assignment followed by replacements for generic symbols would yield a different result from simplification after the replacement of particular values for generic symbols, a delayed assignment **::** should be used. This case is realized when structural operations are performed on the expressions represented by the generic symbols, or when an assignment occurs.

Only values of arguments are usually passed to functions. However, when projections or assignments of an argument are required, it is necessary to pass the argument "by name" in an unsimplified form. Some system-defined functions automatically maintain their arguments in unsimplified form. For example, in **a:1**, **a** is maintained unsimplified. In **!a:1**, **a** is simplified before assignment. If the value of **a** were **b**, then in this case, **b** rather than **a** would be assigned the value **1**. Similarly, functions such as **Plus** to be used as templates are passed in an unevaluated form.

The simplification of any expression may be prevented by explicit use of ´. ´*expr* is an expression equivalent to *expr*, but maintained in an unsimplified form. Structural operations may be performed on the "frozen" or "held" expression ´*expr* just as on *expr*, but the results are left unsimplified until explicitly "released" by **Rel[***expr***]**.

External files in the SMP Library serve as further examples. See particularly **XHorn** (Horner representation of a polynomial), **XLItp** (Lagrangian interpolation) and **XYoung** (Young graphs for group theory).

10. Epilogue

The reader of this primer should by now have a gained a basic working knowledge of SMP. Experience with an actual SMP system is crucial in consolidating and extending this knowledge.

Casual reading of the Reference Manual and the external files in the SMP Library will indicate further capabilities and facilities of SMP.

The keyword indices to the Reference Manual and to the Library may be used to locate a detailed description of facilities relevant to a specific calculation or application. In some cases, a single built-in function, or an existing library function, may perform the complete calculation required. In all but the most mundane cases, however, several functions will be required. The first time a complicated operation is performed, it may be convenient to carry out each step on a separate line, inspecting and checking the intermediate results. However, once the procedure for the operation is clear, a single function should be devised to implement it. It is almost always worthwhile to give the most general definition feasible, and to save it in an external file; later changes of parameters or interpretation may then be investigated without devising necessary definitions again. It is usually valuable to include a short commentary with each external file describing the operation or nature of the definitions given.

An operation or construct requires precise formulation to be defined in SMP. Mathematical definitions may very often be used directly in SMP. Definitions of procedures are often more difficult and lengthy to implement.

The appearance of global variables in definitions should usually be avoided: when the definition is used again, the values of the variables may have been changed.

Separate functions should be defined for each part of a complicated operation, and controlled by an overall dispatching function.

When no complicated mathematical formulae are involved, a rough rule is that individual SMP definitions should usually be less than two or three lines long. If they are longer, an alternative formulation is probably desirable. The shorter its definition, the easier a function is to test, verify and understand.

Once a definition has been devised, it should be checked carefully. Incorrect definitions may often give plausible results. Simple cases with known results should be tried to verify the defintion and its implementation.

It is common for a definition or function not to behave in the way intended or expected. To understand discrepancies, it is often valuable to trace the operations by hand in a simple case.

SMP does what it is defined or programmed to do exactly and pedantically. It can determine what the user intends only through the definitions given.

If an unexpected result is obtained, it is always possible that SMP may be in error. Such errors should be considered as a very last resort in tracing the source of a difficulty. The definitions given should first be examined and investigated in extreme detail.

When performing a calculation, needless complication should be avoided. A more complicated calculation will take longer to perform, and the result will often be lengthy and difficult to assimilate. There is nevertheless no limit in principle to the size of calculations and expressions handled by SMP. (Some installations may however impose limits.) SMP was in fact designed to perform calculations of extreme complexity.

SMP is a very concise language. Very complicated operations can be specified on a single input line, so that the processing of the line may require a substantial amount of time. The internal operation of SMP is nevertheless extremely efficient.

Most scientific investigations involve three phases: development of conceptual framework, mundane calculation and interpretation and exposition of results. It is in the second of these phases that SMP may be used to greatest advantage. It should enable calculations to be performed quickly and without error, and make accessible problems of immense complexity. Judging from the authors´ experience, it will prove an invaluable aid in almost any scientific investigation.

Appendix Some common difficulties

Some of the commoner difficulties encountered by SMP users are collected here.

1. A "system-defined" projection is returned unsimplified when simplification is expected.

   All system-defined projections begin with capital letters. **Ex[(x+1)(x-1)]** will work; **ex[(x+1)(x-1)]** will not.

   Check the spelling of names for system-defined objects.

   Filters for system-defined projections ("arguments" to "functions") are enclosed in square brackets, not parentheses.

2. An input expression is not printed as expected.

   1/2x means **(1/2)*x** not **1/(2x)**.

   x^2a means **(x^2)*a** not **x^(2a)**.

   a/b/c means **a/(b*c)** not **(a*c)/b**.

   Filters for projections are given in square brackets: **f(x)** means **f*x** not **f[x]**.

   ab is a single symbol, not the product of symbols **a b** or **a*b**.

3. SMP runs for a long time without printing a result.

   Try to stop processing using ⊂*quit interrupt*⊃ or ⊂*break interrupt*⊃.

   Estimate how complicated the calculation requested should have been. Try an analogous but simpler calculation.

   A non-terminating recursive assignment may have been performed. Investigate relevant assignments to see whether a special case of the left-hand side may appear on the right-hand side. If so, insert suitable conditions to prevent infinite recursion. If the assignment itself leads to infinite recursion, use **::** rather than **:**.

   A non-terminating sequence of substitutions may have been specified through **S** or **Si**. In **S** projections, the last filter may be a repetition count: a level specification must be given as second-to-last filter.

   A non-terminating **For** or **Do** loop may have been invoked; usually these yield a processing impasse after 1000 iterations. A non-terminating recursive projection definition may have been made and used.

4. A strange result involving unexpected objects is obtained.

   Some of the objects used in the current expression were probably assigned values earlier in the job. The value of *expr* is removed by *expr***:**.

   Subexpressions may be simplified when not intended: ´*expr* is equivalent to *expr* but remains in an unsimplified form.

   See also 5.

5. An input ambiguity is reported on an input line with apparently correct syntax.

   Syntax modifications may have been performed. These are removed by **Sxset[]**.

   Non-printing characters may have been entered. Re-type the input line. Multiplication must be input as an explicit ***** when ⊂*space*⊃ would be ambiguous: **a <b** means **Gt[b,a]** not **a*<b**.

6. An assignment is not used when expected, or a pattern does not match as expected.

   f[x]:x^2 assigns the value of the projection **f[x]** with the particular filter **x** to be **x^2**, and has no consequences for **f[y]**. **f[$x]:$x^2** assigns the projection of **f** with an arbitrary expression **$x** to be **$x^2**.

   x^$n:*expr* does not assign a value for **x**. Similarly, **$x+x:***expr*, **$x x:***expr* and **x/$x:***expr* do not assign values for **x**.

   f[$x+$y,$x-$y]:*expr* does not assign a value for **f[4,1]**.

f[$x+$y]:*expr* assigns a value for **f[x+y]** but not for **f[x+y+z]**; **f[$x+$$y]** assigns a value for any number of summands.

f[1-$x,$x] does not match **f[a,1-a]** since **1-$x** and **1-1+a** do not match, because **Plus** is a projection with property **Flat**; **f[1-$$x,$$x]** would match.

f[$$x_?(Len[$$x]>3)]:*expr* causes **f[a,b,c,d]** to test **Len[a,b,c,d]**; **f[$$x_?(Len[List[$$x]]>3)]** is a suitable test for projections of **f** with more than three filters.

Properties such as **Flat**, **Tier** and **Comm** should be assigned to projections before any values are assigned to the projections.

Indices in lists are not simplified or modified except when they are first assigned.

f[$x]:=$x^2 specifies a syntax modification, not an assignment.

7. The value obtained from a projection is not the one which was supposed to have been assigned, or which appears to be present.

   Positions of parts are determined by the simplified form of an expressions. Particularly in commutative functions, positions of parts may change when assignments for other parts are made.

   f[$x]:S[$x,x->1] makes the assignment **f[$x]:$x**; **f[$x]::S[$x,x->1]** is a delayed assignment which defines the **S** to be simplified again for each specific form of **$x** provided.

   If a projection unexpectedly yields a list, the projection requested probably had a different number of filters from the one assigned. Alternatively, the projection should have been assigned property **Tier.**

   a:b;c means **(a:b);c not a:(b;c)**.

   Expressions may be printed in a form which differs from their internal form. Such expressions are indicated by a ***** at the beginning of output. Direct forms in which labelling of parts are manifest is obtained by **Lpr**.

8. A replacement in an **S** projection did not behave as expected.

   f[$x]->S[$x,x->1] becomes **f[$x]->$x**; **f[$x]-->S[$x,x->1]** is a delayed replacement which causes the **S** projection to be simplified again each time the replacement is used.

   Unless the *rpt* filter is specified, **S** does not perform further replacements on expressions obtained as a result of replacements.

9. **Run** or **<** failed to find the specified files.

   All file names must be symbols: **smp.in** must therefore be written as **"smp.in"**.

10. A template was not used as expected.

    Generic symbols in a template are ordered lexically (with **$$x** before **$%x** before **$x**) and associated with expressions in that order. In a sequence of nested template applications, the outermost is usually performed first.

    Ap[f[x],{1}] yields **Proj[f[x],{1}]**; **Ap[`f[x],{1}]** yields **f[x]**.

    Templates are not simplified before use, except when given as **!***temp*.

Glossary

Terminology or usage specific to SMP is indicated by *.
References are to sections in the summary/reference manual.

array Example of a contiguous list.

ASCII "American Standard Code for Information Interchange"; a standard set of numerical codes for characters.

* ***assignment*** The association of an attribute (usually value) to an expression [3.2]; definition; assertion; assumption; declaration.

asynchronous Not occuring in a fixed time sequence.

binary file File containing direct binary machine code instructions.

* ***block*** Unit of computer memory (usually 16 bytes); space required to store one symbol.

break interrupt Real time interrupt used to suspend processing and enter an interactive subsidiary procedure.

buffer Temporary text storage region.

bug An error in a program or procedure.

bus error A class of program errors involving illegal memory fetches.

byte Usually 8 binary bits of computer memory.

C The programming language in which SMP was written.

* ***chameleonic expression*** An expression containing chameleonic symbols [2.8].

* ***chameleonic symbol*** Symbol whose name changes whenever it is evaluated [2.8].

character string See ***string***.

comment Input text used for descriptive purposes and not processed [2.9].

compiler Program for transforming code in a high-level language into machine code.

* ***contiguous list*** A list whose indices are successive integers starting at 1 [2.4].

control character Character typed at a terminal while "CTRL" key is depressed.

core Main computer memory.

CPU Central processing unit; processor.

* ***criterion*** Template [2.7] applied to expressions to determine whether an operation should be preformed on them.

data space Portion of computer memory used to store expressions manipulated in a job.

* ***deassignment*** The removal of an attribute (usually value) from an expression [3.2].

default Attribute assumed if not explicitly specified; "standard".

* ***delayed value*** A value maintained in an unsimplified form, and simplified afresh when used [3.2].

* ***depth*** The maximum number of filters necessary to specify any part of an expression [2.5].

* ***domain*** A set of parts in an expression [2.5].

* ***editor*** Interactive facility for performing textual modifications on input lines [1.7].

* ***entry*** Expression appearing in a list [2.4].

* ***evaluation*** Process of replacing symbols and projections by values assigned to them [3.7].

exception Processing error detected by CPU.

* ***expression*** Combination of symbols, projections and lists manipulated by SMP [2.5].

* ***extension*** See ***type extension***.

external file A file containing SMP expressions to be input when required [1.4].

external program A program to be run in the monitor taking input from SMP and passing input back to SMP.

file A named area (usually on a magnetic disk) used to provide long-term storage of text or data.

* ***filter*** An expression appearing in a projection [2.3] (***e.g.*** **x** or **y** in **f[x,y]**).

* ***full list*** List forming a "rectangular array" [7.6].

function Example of a projection.

garbage collection Process by which areas of computer memory not required for further processing are reclaimed.

* ***generic expression*** An expression containing generic symbols; pattern [2.6].

* ***generic symbol*** A symbol representing any one of a possibly infinite class of expressions [2.6]; dummy symbol; pattern variable.

global object An object whose value or attributes can affect expressions which do not directly contain it [1.3].

* ***held form*** A form of an expression in which simplification is prevented (***e.g.*** **´x**) [3.5].

* ***immediate value*** A value simplified when assigned, and maintained in a simplified form [3.2].

* ***impasse*** See ***processing impasse***.

* ***index*** An expression labelling an entry in a list [2.4].

* ***interactive procedure*** Procedure in which results from one segment are output before the next segment is input.

* ***interpreter*** Program which executes procedures in a high-level language interactively.

interrupt See ***real-time interrupt***.

iteration Successive simplification or evaluation of a set of expressions [6.2].

job One execution of a program, from initiation to termination; process; task.

* ***length*** The number of entries in a list or number of filters in a projection [7.4].

* ***level*** A set of parts of an expression [2.5].

* ***list*** Indexed and ordered set of expressions (***e.g.*** **{x,y}** or **{[a1]:x,[a2]:y}**) [2.4]; vector; structure or record; lambda function; rule.

local variables Symbols used within a procedure, and restored on exit from the procedure [6.3].

logging in Signing on; connecting to computer.

* ***match*** A pattern ***p2*** matches ***p1*** if it represents a superset of the expressions represented by ***p1*** [2.6].

memory fault A class of program errors involving use of illegal memory addresses.

modem A device for acoustic or electronic connection of computer equipment to a telephone.

monitor Shell; command line interpreter; operating system.

* ***multi-generic symbol*** Generic symbol representing a sequence of expressions [2.6].

newline Line termination character; carriage return; line feed; enter.

* ***null projection*** Special projection representing a sequence of expressions [2.3].

paging The ability to read into main memory areas of virtual memory only when they are requested by a running program.

parallel processing Independent and usually simultaneous execution of several operations.

parsing Process by which textual input is transformed into internal representation.

* ***pattern*** An expression containing generic symbols; generic expression [2.6].

postprocessor A procedure or operation performed before output on each expression after processing (**Post** [1.3]).

precedence Attribute determining the grouping of input forms [2.10].

preprocessor A procedure or operation performed on each input expression before further processing (**Pre** [1.3]).

precedence Ordering for treatment of input forms [2.10].

procedure A set of expressions to be simplified in order when required [6.3]; program.

process Independent simplification of a procedure; job.

* ***processing impasse*** A state achieved when complete processing of an input expression appears to require infinite time or memory space.

* ***projection*** Expression specifying part of a general structure (***e.g.*** **f[x,y]**) [2.3]; array subscripting; function or procedure call.

* ***projector*** The object from which a projection is taken [2.3] (***e.g.*** **f** in **f[x,y]**).

prompt Character or message printed by the computer to indicate that further input may be given.

* ***property*** Attribute of a symbol used to specify its treatment [4].

quit interrupt Real time interrupt used to terminate current processing [1.5].

real-time interrupt A command to be issued at any time (not necessarily after an input prompt) [1.5].

recursion Simplification of an expression involving further simplification of similar expressions [3.1].

* ***replacement*** A construct specifying a substitution (***e.g.*** **x->y**) [3.3].

segment An expression forming part of a procedure [6.3].

* ***simplification*** The process performed by SMP on input expressions [3.1].

string A sequence of textual characters.

* ***status interrupt*** Real time interrupt used to obtain information on the status of processing.

subscripted variable Example of a projection.

* ***subsidiary mode*** Input mode for entry of segments in interactive subsidiary procedures.

* ***symbol*** Fundamental symbolic object (***e.g.*** **x** and **Mult**) [2.2]; variable; parameter; unknown.

syntax Construction of expressions [2].

* ***system-defined*** Object or procedure with a built-in definition in SMP; primitive.

systems programming Computer programming intended as a service to many users rather than directed to a particular application.

tab ASCII HT; usually CTRL I; usually has fixed "stops" 8 spaces apart.

* ***template*** Expression specifying an action on a set of expressions [2.7].

terminal Computer input and output device; teletype; console; VDU.

termination character Character signifying termination of the current procedure and used to terminate a job [1.5].

text space Portion of computer memory in which the executable text of SMP resides.

* ***tick*** Unit of computing time in SMP [A.8].

* ***type extension*** Mechanism through which operations may be modified by the nature of their operands [4].

* ***user-defined*** Input by the "user" and not intrinsic to SMP.

user name Name by which a person or group is specified to a computer.

* ***value*** An expression specified to replace an expression or class of expressions on simplification [3.1].

virtual memory System by which effective computer memory may include not only physical main memory but also secondary storage media and thus be of practically arbitrary size.