

## SCMUTILS Reference Manual

This is a description of the Scmutils system, an integrated library of procedures, embedded in the programming language Scheme, and intended to support teaching and research in mathematical physics and electrical engineering. Scmutils and Scheme are particularly effective in work where the almost-functional nature of Scheme is advantageous, such as classical mechanics, where many of the procedures are most easily formulated as quite abstract manipulations of functions.

Many people contributed to the development of the Scmutils library over many years, so we may miss some of them. The principal contributors have been:

Gerald Jay Sussman, Harold Abelson, Jack Wisdom, Jacob Katzenelson, Hardy Mayer, Christopher P. Hanson, Matthew Halfant, Bill Siebert, Guillermo Juan Rozas, Panayotis Skordos, Kleantes Koniaris, Kevin Lin, Dan Zuras

## Scheme and Functional Programming

Scheme is a simple computer language in the Lisp family of languages, with important structural features derived from Algol-60. We will not attempt to document Scheme here, as it is adequately documented in the IEEE standard (IEEE-1178) and in numerous articles and books. The R<sup>4</sup>RS is a terse document describing Scheme in detail. It is included with this document. We assume that a reader of this document is familiar with Scheme and has read a book such as

Harold Abelson, Gerald Jay Sussman and Julie Sussman  
Structure and Interpretation of Computer Programs  
MIT Press and McGraw-Hill (1985, 1996)

As a reminder, Scheme is an expression-oriented language. Expressions have values. The value of an expression is constructed from the values of the constituent parts of the expression and the way the expression is constructed. Thus, the value of

```
(+ (* :pi (square r)) 1)
```

is constructed from the values of the symbols

```
+, *, :pi, square, r, 1
```

and by the parenthetical organization.

In any Lisp-based language, an expression is constructed from parts with parentheses. The first subexpression always denotes a procedure and the rest of the subexpressions denote arguments. So in the case of the expression above, "square" is a symbol whose value is a procedure that is applied to a thing (probably a number) denoted by "r". That value of the application of square to r is then combined with the number denoted by ":pi" using the procedure denoted by "\*" to make an object (again probably a number) that is combined with the number denoted by "1" using the procedure denoted by "+". Indeed, if the symbols have the values we expect from their (hopefully mnemonic) names,

```
+      = a means of addition
*      = a means of multiplication
:pi    = a number approximately equal to 3.14159265358979
square = a means of squaring
1      = the multiplicative identity in the reals
r      some number, say for example 4
```

then this expression would have the approximate value denoted by the numeral 51.26548245743669.

We can write expressions denoting procedures. For example the procedure for squaring can be written

```
(lambda (x) (* x x)) ,
```

which may be read

```
"the procedure of one argument x that multiplies x by x" .
```

We may bind a symbol to a value by definition

```
(define square
  (lambda (x) (* x x))) ,
```

or equivalently, by

```
(define (square x) (* x x)) .
```

The application of a defined procedure to operands will bind the symbols that are the formal parameters to the actual arguments that are the values of the operands:

```
(+ (square 3) (square 4)) => 25
```

This concludes the reminders about Scheme. You must consult an alternate source for more information.

One caveat: unlike the Scheme standard the Scmutils system is case sensitive.

## Generic Arithmetic

In the Scmutils library arithmetic operators are generic over a variety of mathematical datatypes. For example, addition makes sense when applied to such diverse data as numbers, vectors, matrices, and functions. Additionally, many operations can be given a meaning when applied to different datatypes. For example, multiplication makes sense when applied to a number and a vector.

The traditional operator symbols, such as "+" and "\*" are bound to procedures that implement the generic operations. The details of which operations are defined for which datatypes is described below, organized by the datatype.

In addition to the standard operations, every piece of mathematical data, *x*, can give answers to the following questions:

(type *x*)

Returns a symbol describing the type of *x*. For example,

```
(type 3.14)           => *number*
(type (vector 1 2 3)) => *vector*
```

(type-predicate *x*)

Returns a predicate that is true on objects that are the same type as *x*

(arity *p*)

Returns a description of the number of arguments that *p*, interpreted as a procedure, accepts, compatible with the MIT Scheme procedure-arity procedure, except that it is extended for datatypes that are not usually interpreted as procedures. A structured object, like a vector, may be applied as a vector of procedures, and its arity is the intersection of the arities of the components.

An arity is a newly allocated pair whose car field is the minimum number of arguments, and whose cdr field is the maximum number of arguments. The minimum is an exact non-negative integer. The maximum is either an exact non-negative integer, or '#f' meaning that the procedure has no maximum number of arguments. In our version of Scheme #f is the same as the empty list, and a pair with the empty list in the cdr field is a singleton list, so the arity will print as shown in the second column.

```
(arity (lambda () 3))           => (0 . 0) = (0 . 0)
(arity (lambda (x) x))          => (1 . 1) = (1 . 1)
(arity car)                     => (1 . 1) = (1 . 1)
(arity (lambda x x))            => (0 . #f) = (0)
(arity (lambda (x . y) x))      => (1 . #f) = (1)
(arity (lambda (x #!optional y) x)) => (1 . 2) = (1 . 2)
(arity (vector cos sin))        => (1 . 1) = (1 . 1)
```

We will now describe each of the generic operations. These operations are defined for many but not all of the mathematical datatypes. For particular datatypes we will list and discuss the operations that only make sense for them.

(inexact? x)

This procedure is a predicate -- a boolean-valued procedure. See the R<sup>4</sup>RS for an explanation of exactness of numbers. A compound object, such as a vector or a matrix, is inexact if it has inexact components.

(zero-like x)

In general, this procedure returns the additive identity of the type of its argument, if it exists. For numbers this is 0.

(one-like x)

In general, this procedure returns the multiplicative identity of the type of its argument, if it exists. For numbers this is 1.

(zero? x)

Is true if x is an additive identity.

(one? x)

Is true if x is a multiplicative identity.

(negate x) = (- (zero-like x) x)

Gives an object that when added to x yields zero.

(invert x) = (/ (one-like x) x)

Gives an object that when multiplied by x yields one.

Most of the numerical functions have been generalized to many of the datatypes, but the meaning may depend upon the particular datatype. Some are defined for numerical data only.

(= x1 x2 ...) ==> <boolean>

(+ x1 x2 ...)

(\* x1 x2 ...)

(- x1 x2 ...)

(/ x1 x2 ...)

(expt x1 x2)

(gcd n1 n2 ...)

(sqrt x) Gives a square root of x, or an approximation to it.

(exp x) = :e<sup>x</sup>

(exp10 x) = 10<sup>x</sup>

(exp2 x) = 2<sup>x</sup>

(log x)

(log10 x) = (/ (log x) (log 10))

(log2 x) = (/ (log x) (log 2))

(sin x), (cos x), (tan x)

(sec x), (csc x)

(asin x), (acos x), (atan x)

(atan x1 x2) = (atan (/ x1 x2)) but retains quadrant information

(sinh x), (cosh x), (tanh x)

(sech x), (csch x)

(make-rectangular a1 a2) = a1+ia2

(make-polar a1 a2) = a1\*:e<sup>(\* +i a2)</sup>

(real-part z)

(imag-part z)

(magnitude z)

(angle z)

(conjugate z)

If M is a quantity that can be interpreted as a square matrix,

(determinant M)

(trace M)

## Scheme Numbers

Operations on the Scheme Number datatype that are part of standard Scheme are listed here without comment; those that are not part of standard Scheme are described. In the following <n> is (any expression that denotes) an integer. <a> is any real number, <z> is any complex number, and <x> and <y> are any kind of number.

```
(type <x>)           = *number*
(inexact? <x>)      ==> <boolean>
(zero-like <x>)     = 0
(one-like <x>)      = 1
(zero? <x>)         ==> <boolean>
(one? <x>)          ==> <boolean>

(negate <x>), (invert <x>), (sqrt <x>)

(exp <x>), (exp10 <x>), (exp2 <x>)

(log <x>), (log10 <x>), (log2 <x>)

(sin <x>), (cos <x>), (tan <x>), (sec <x>), (csc <x>)

(asin <x>), (acos <x>), (atan <x>)
(atan <x1> <x2>)

(sinh <x>), (cosh <x>), (tanh <x>), (sech <x>), (csch <x>)

(= <x1> <x2> ...) ==> <boolean>
(+ <x1> <x2> ...)
(* <x1> <x2> ...)
(- <x1> <x2> ...)
(/ <x1> <x2> ...)

(expt <x1> <x2>)
(gcd <n1> <n2> ...)

(make-rectangular <a1> <a2>) = <a1>+i<a2>
(make-polar <a1> <a2>)     = <a1>*:e^(* +i <a2>)
(real-part <z>)
(imag-part <z>)
(magnitude <z>)
(angle <z>)

(conjugate <z>)
```

## Structured Objects

Scmutils supports a variety of structured object types, such as vectors, up and down tuples, matrices, and power series.

The explicit constructor for a structured object is a procedure whose name is what we call objects of that type. For example, we make explicit vectors with the procedure named "vector", and explicit lists with the procedure named "list". For example

```
(list 1 2 3 4 5) a list of the first five positive integers
(vector 1 2 3 4 5) a vector of the first five positive integers
(down 10 3 4) a down tuple with three components
```

There is no natural way to notate a matrix, except by giving its rows (or columns). To make a matrix with three rows and five columns:

```
(define M
  (matrix-by-rows (list 1 2 3 4 5)
                  (list 6 7 8 9 10)
                  (list 11 12 13 14 15)))
```

A power series may be constructed from an explicit set of coefficients

```
(series 1 2 3 4 5)
```

is the power series whose first five coefficients are the first five positive integers and all of the rest of the coefficients are zero.

Although each datatype has its own specialized procedures, there are a variety of generic procedures for selecting the components from structured objects. To get the n-th component from a linear data structure, v, such as a vector or a list, one may in general use the generic selector, "ref":

```
(ref x n)
```

All structured objects are accessed by zero-based indexing, as is the custom in Scheme programs and in relativity. For example, to get the third element (index = 2) of a vector or a list we can use

```
(ref (vector 1 2 3 4 5) 2) = 3
(ref (list 1 2 3 4 5) 2) = 3
```

If M is a matrix, then the component in the i-th row and j-th column can be obtained by (ref M i j). For the matrix given above

```
(ref M 1 3) = 9
```

Other structured objects are more magical

```
(ref cos-series 6) = -1/720
```

The number of components of a structured object can be found with the "size" generic operator:

```
(size (vector 1 2 3 4 5)) = 5
```

Besides the extensional constructors, most structured-object datatypes can be intentionally constructed by giving a procedure whose values are the components of the object. These "generate" procedures are

```
(list:generate    n  proc)
(vector:generate  n  proc)
(matrix:generate  m n proc)
(series:generate   proc)
```

For example, one may make a 6 component vector each of whose components is  $:\pi$  times the index of that component, as follows:

```
(vector:generate 6 (lambda (i) (* :pi i)))
```

Or a 3X5 matrix whose components are the sum of  $:\pi$  times the row number and the speed of light times the column number:

```
(matrix:generate 3 5 (lambda (i j) (+ (* :pi i) (* :c j))))
```

Also, it is commonly useful to deal with a structured object in an elementwise fashion. We provide special combinators for many structured datatypes that allow one to make a new structure, of the same type and size of the given ones, where the components of the new structure are the result of applying the given procedure to the corresponding components of the given structures.

```
((list:elementwise proc) <l1> ... <ln>)
((vector:elementwise proc) <v1> ... <vn>)
((structure:elementwise proc) <s1> ... <sn>)
((matrix:elementwise proc) <M1> ... <Mn>)
((series:elementwise proc) <p1> ... <pn>)
```

Thus, vector addition is equivalent to (vector:elementwise +).

## Scheme Vectors

We identify the Scheme vector data type with mathematical  $n$ -dimensional vectors. These are interpreted as up tuples when a distinction between up tuples and down tuples is made.

We inherit from Scheme the constructors VECTOR and MAKE-VECTOR, the selectors VECTOR-LENGTH and VECTOR-REF, and zero-based indexing. We also get the iterator MAKE-INITIALIZED-VECTOR, and the type predicate VECTOR?. In the documentation that follows,  $\langle v \rangle$  will stand for a vector-valued expression.

```
(vector? <any>)          ==> <boolean>
(type <v>)               ==> *vector*
(inexact? <v>)          ==> <boolean>
    Is true if any component of <v> is inexact, otherwise it is false.
```

```
(vector-length <v>)      ==> <+integer>
    gets the number of components of <v>
```

```
(vector-ref <v> <i>)     ==> <vector>
    gets the <i>th (zero-based) component of vector <v>
```

```
(make-initialized-vector <n> <procedure>)
    this is also called (v:generate <n> <procedure>)
    and (vector:generate <n> <procedure>)
```

generates an  $\langle n \rangle$ -dimensional vector whose  $\langle i \rangle$ th component is the result of the application of the  $\langle \text{procedure} \rangle$  to the number  $\langle i \rangle$ .

```
(zero-like <v>)          ==> <vector>
    Gives the zero vector of the dimension of vector <v>.
```

```
(zero? <v>)              ==> <boolean>
(negate <v>)             ==> <vector>
```

```
(conjugate <v>)          ==> <vector>
    Elementwise complex-conjugate of <v>
```

Simple arithmetic on vectors is componentwise

```
(= <v1> <v2> ...)       ==> <boolean>
(+ <v1> <v2> ...)       ==> <vector>
(- <v1> <v2> ...)       ==> <vector>
```

There are a variety of products defined on vectors.

(dot-product <v1> <v2>) ==> <x>

(cross-product <v1> <v2>)

Cross product only makes sense for 3-dimensional vectors.

(\* <x> <v>) = (scalar\*vector <x> <v>) ==> <vector>

(\* <v> <x>) = (vector\*scalar <v> <x>) ==> <vector>

(/ <v> <x>) = (vector\*scalar <v> (/ 1 <x>)) ==> <vector>

The product of two vectors makes an outer product structure.

(\* <v> <v>) = (outer-product <v> <v>) ==> <structure>

(euclidean-norm <v>) = (sqrt (dot-product <v> <v>))

(abs <v>) = (euclidean-norm <v>)

(v:inner-product <v1> <v2>) = (dot-product (conjugate <v1>) <v2>)

(complex-norm <v>) = (sqrt (v:inner-product <v> <v>))

(magnitude <v>) = (complex-norm <v>)

(maxnorm <v>)

gives the maximum of the magnitudes of the components of <v>

(v:make-unit <v>) = (/ <v> (euclidean-norm <v>))

(v:unit? <v>) = (one? (euclidean-norm <v>))

(v:make-basis-unit <n> <i>)

Makes the n-dimensional basis unit vector with zero in all components except for the ith component, which is one.

(v:basis-unit? <v>)

Is true if and only if <v> is a basis unit vector.

## Up Tuples and Down Tuples

Sometimes it is advantageous to distinguish down tuples and up tuples. If the elements of up tuples are interpreted to be the components of vectors in a particular coordinate system, the elements of the down tuples may be thought of as the components of the dual vectors in that coordinate system. The union of the up tuple and the down tuple data types is the data type we call "structures."

Structures may be recursive and they need not be uniform. Thus it is possible to have an up structure with three components: the first is a number, the second is an up structure with two numerical components, and the third is a down structure with two numerical components. Such a structure has size (or length) 3, but it has five dimensions.

In Scmutils, the Scheme vectors are interpreted as up tuples, and the down tuples are distinguished. The predicate "structure?" is true of any down or up tuple, but the two can be distinguished by the predicates "up?" and "down?".

```
(up?   <any>) ==> <boolean>
(down? <any>) ==> <boolean>
```

```
(structure? <any>) = (or (down? <any>) (up? <any>))
```

In the following, <s> stands for any structure-valued expression; <up> and <down> will be used if necessary to make the distinction.

The generic type operation distinguishes the types:

```
(type <s>)          ==> *vector* or *down*
```

We reserve the right to change this implementation to distinguish Scheme vectors from up tuples. Thus, we provide (null) conversions between vectors and up tuples.

```
(vector->up <scheme-vector>) ==> <up>
(vector->down <scheme-vector>) ==> <down>
```

```
(up->vector <up>)          ==> <scheme-vector>
(down->vector <down>)      ==> <scheme-vector>
```

Constructors are provided for these types, analogous to list and vector.

```
(up . args)    ==> <up>
(down . args)  ==> <down>
```

The dimension of a structure is the number of entries, adding up the numbers of entries from substructures. The dimension of any structure can be determined by

```
(s:dimension <s>          ==> <+integer>
```

Processes that need to traverse a structure need to know the number of components at the top level. This is the length of the structure,

```
(s:length <s>)           ==> <+integer>
```

The *i*th component (zero-based) can be accessed by

```
(s:ref <s> i)
```

For example,

```
(s:ref (up 3 (up 5 6) (down 2 4)) 1)
(up 5 6)
```

As usual, the generic ref procedure can recursively access substructure

```
(ref (up 3 (up 5 6) (down 2 4)) 1 0)
5
```

Given a structure *<s>* we can make a new structure of the same type with *<x>* substituted for the *<n>*th component of the given structure using

```
(s:with-substituted-coord <s> <n> <x>)
```

We can construct an entirely new structure of length *<n>* whose components are the values of a procedure using *s:generate*:

```
(s:generate <n> up/down <procedure>)
```

The up/down argument may be either up or down.

The following generic arithmetic operations are defined for structures.

```
(zero? <s>) ==> <boolean>
```

is true if all of the components of the structure are zero.

```
(zero-like <s>) ==> <s>
```

produces a new structure with the same shape as the given structure but with all components being zero-like the corresponding component in the given structure.

```
(negate <s>) ==> <s>
```

```
(magnitude <s>) ==> <s>
```

```
(abs <s>) ==> <s>
```

```
(conjugate <s>) ==> <s>
```

produce new structures which are the result of applying the generic procedure elementwise to the given structure.

```
(= <s1> ... <sn>) ==> <boolean>
```

is true only when the corresponding components are =.

```
(+ <s1> ... <sn>) ==> <s>
```

```
(- <s1> ... <sn>) ==> <s>
```

These are componentwise addition and subtraction.

```
(* <s1> <s2>) ==> <s> or <x> , a structure or a number
```

magically does what you want: If the structures are compatible for contraction the product is the contraction (the sum of the products of the corresponding components.) If the structures are not compatible for contraction the product is the structure of the shape and length of <s2> whose components are the products of <s1> with the corresponding components of <s2>.

Structures are compatible for contraction if they are of the same length, of opposite type, and if their corresponding elements are compatible for contraction.

It is not obvious why this is what you want, but try it, you'll like it!

For example, the following are compatible for contraction:

```
(print-expression (* (up (up 2 3) (down 5 7 11))
                     (down (down 13 17) (up 19 23 29))))
```

652

Two up tuples are not compatible for contraction.  
Their product is an outer product:

```
(print-expression (* (up 2 3) (up 5 7 11)))
(up (up 10 15) (up 14 21) (up 22 33))
```

```
(print-expression (* (up 5 7 11) (up 2 3)))
(up (up 10 14 22) (up 15 21 33))
```

This product is not generally associative or commutative. It is commutative for structures that contract, and it is associative for structures that represent linear transformations.

To yield additional flavor, the definition of square for structures is inconsistent with the definition of product. It is possible to square an up tuple or a down tuple. The result is the sum of the squares of the components. This makes it convenient to write such things as (/ (square p) (\* 2 m)), but it is sometimes confusing.

Some structures, such as the ones that represent inertia tensors, must be inverted. (The "m" above may be an inertia tensor!) Division is arranged to make this work, when possible. The details are too hairy to explain in this short document. We probably need to write a book about this!

## Matrices

There is an extensive set of operations for manipulating matrices. Let  $\langle M \rangle$ ,  $\langle N \rangle$  be matrix-valued expressions. The following operations are provided

```
(matrix? <any>)          ==> <boolean>
(type <M>)                ==> *matrix*
(inexact? <M>)           ==> <boolean>

(m:num-rows <M>)         ==> <n>,
  the number of rows in matrix M.

(m:num-cols <M>)         ==> <n>,
  the number of columns in matrix M.

(m:dimension <M>)        ==> <n>
  the number of rows (or columns) in a square matrix M.
  It is an error to try to get the dimension of a matrix
  that is not square.

(column-matrix? <M>)
  is true if M is a matrix with one column.
  Note: neither a tuple nor a scheme vector is a column matrix.

(row-matrix? <M>)
  is true if M is a matrix with one row.
  Note: neither a tuple nor a scheme vector is a row matrix.
```

There are general constructors for matrices:

```
(matrix-by-rows <row-list-1> ... <row-list-n>)
  where the row lists are lists of elements that are to appear in the
  corresponding row of the matrix

(matrix-by-cols <col-list-1> ... <col-list-n>)
  where the column lists are lists of elements that are to appear in
  the corresponding column of the matrix

(column-matrix <x1> ... <xn>)
  returns a column matrix with the given elements

(row-matrix <x1> ... <xn>)
  returns a row matrix with the given elements
```

And a standard selector for the elements of a matrix

```
(matrix-ref <M> <n> <m>)
  returns the element in the m-th column and the n-th row of matrix M.
  Remember, this is zero-based indexing.
```

We can access various parts of a matrix

```
(m:nth-col <M> <n>) ==> <scheme-vector>
  returns a Scheme vector with the elements of the n-th column of M.
```

```
(m:nth-row <M> <n>) ==> <scheme-vector>
  returns a Scheme vector with the elements of the n-th row of M.
```

```
(m:diagonal <M>) ==> <scheme-vector>
  returns a Scheme vector with the elements of the diagonal of the
  square matrix M.
```

```
(m:submatrix <M> <from-row> <upto-row> <from-col> <upto-col>)
  extracts a submatrix from M, as in the following example
```

```
(print-expression
 (m:submatrix
  (m:generate 3 4
   (lambda (i j)
     (* (square i) (cube j))))
  1 3 1 4))
(matrix-by-rows (list 1 8 27)
                (list 4 32 108))
```

```
(m:generate <n> <m> <procedure>) ==> <M>
  returns the nXm (n rows by m columns) matrix whose ij-th element is
  the value of the procedure when applied to arguments i, j.
```

```
(simplify
 (m:generate 3 4
  (lambda (i j)
    (* (square i) (cube j)))))
=> (matrix-by-rows (list 0 0 0 0)
                  (list 0 1 8 27)
                  (list 0 4 32 108))
```

```
(matrix-with-substituted-row <M> <n> <scheme-vector>)
  returns a new matrix constructed from M by substituting the Scheme
  vector v for the n-th row in M.
```

We can transpose a matrix (producing a new matrix whose columns are the rows of the given matrix and whose rows are the columns of the given matrix with:

```
(m:transpose <M>)
```

There are coercions between Scheme vectors and matrices:

```
(vector->column-matrix <scheme-vector>) ==> <M>
(column-matrix->vector <M>) ==> <scheme-vector>

(vector->row-matrix <scheme-vector>) ==> <M>
(row-matrix->vector <M>) ==> <scheme-vector>
```

And similarly for up and down tuples:

```
(up->column-matrix <up>) ==> <M>
(column-matrix->up <M>) ==> <up>

(down->row-matrix <down>) ==> <M>
(row-matrix->down <M>) ==> <down>
```

Matrices can be tested with the usual tests:

```
(zero? <M>)
(identity? <M>)
(diagonal? <M>)

(m:make-zero <n>) ==> <M>
  returns an nXn (square) matrix of zeros

(m:make-zero <n> <m>) ==> <M>
  returns an nXm matrix of zeros
```

Useful matrices can be made easily

```
(zero-like <M>) ==> <N>
  returns a zero matrix of the same dimensions as the given matrix

(m:make-identity <n>) ==> <M>
  returns an identity matrix of dimension n

(m:make-diagonal <scheme-vector>) ==> <M>
  returns a square matrix with the given vector elements on the
  diagonal and zeros everywhere else.
```

Matrices have the usual unary generic operators:

negate, invert, conjugate,

However the generic operators

exp, sin, cos,

yield a power series in the given matrix.

Square matrices may be exponentiated to any exact positive integer power:

(expt <M> <n>)

We may also get the determinant and the trace of a square matrix:

(determinant <M>)

(trace <M>)

The usual binary generic operators make sense when applied to matrices. However they have been extended to interact with other datatypes in a few useful ways. The componentwise operators

=, +, -

are extended so that if one argument is a square matrix,  $M$ , and the other is a scalar,  $x$ , then the scalar is promoted to a diagonal matrix of the correct dimension and then the operation is done on those:

(= <M> <x>) and (= <x> <M>) tests if  $M = xI$

(+ <M> <x>) and (+ <x> <M>) =  $M+xI$

(- <M> <x>) =  $M-xI$  and (- <x> <M>) =  $xI-M$

Multiplication,  $*$ , is extended to allow a matrix to be multiplied on either side by a scalar. Additionally, a matrix may be multiplied on the left by a conforming down tuple, or on the right by a conforming up tuple.

Division is interpreted to mean a number of different things depending on the types of the arguments. For any matrix  $M$  and scalar  $x$

(/ <M> <x>) = (\* <M> (/ 1 <x>))

If  $M$  is a square matrix then it is possible that it is invertible, so if  $<x>$  is either a scalar or a matrix, then

(/ <x> <M>) = (\* <x> <N>), where  $N$  is the matrix inverse of  $M$ .

In general, if  $M$  is a square matrix and  $v$  is either an up tuple or a column matrix, then

(/ <v> <M>) = <w>, where  $w$  is of the same type as  $v$  and where  $v=Mw$ .

Similarly, for  $v$  a down tuple

(/ <v> <M>) = <w>, where  $w$  is a down tuple and where  $v=wM$ .

## Functions

In Scmutils, functions are data just like other mathematical objects, and the generic arithmetic system is extended to include them. If  $\langle f \rangle$  is an expression denoting a function then

```
(function? <any>)      ==> <boolean>
(type <f>)              ==> *function*
```

Operations on functions generally construct new functions that are the composition of the operation with its arguments, thus applying the operation to the value of the functions: if  $U$  is a unary operation, if  $f$  is a function, and if  $x$  is arguments appropriate to  $f$ , then

```
((U f) x) = (U (f x))
```

If  $B$  is a binary operation, if  $f$  and  $g$  are functions, and if  $x$  is arguments appropriate to both  $f$  and  $g$ , then

```
((B f g) x) = (B (f x) (g x))
```

All of the usual unary operations are available. So if  $\langle f \rangle$  is an expression representing a function, and if  $\langle x \rangle$  is any kind of argument for  $\langle f \rangle$  then, for example,

```
((negate <f>) <x>) = (negate (f <x>))
((invert <f>) <x>) = (invert (f <x>))
((sqrt <f>) <x>)   = (sqrt (f <x>))
```

The other operations that behave this way are:

```
exp, log, sin, cos, asin, acos, sinh, cosh, abs,
real-part, imag-part, magnitude, angle, conjugate, atan
```

The binary operations are similar, with the exception that mathematical objects that may not be normally viewed as functions are coerced to constant functions for combination with functions.

```
((+ <f> <g>) <x>) = (+ (f <x>) (g <x>))
((- <f> <g>) <x>) = (- (f <x>) (g <x>))
```

For example:

```
((+ sin 1) x) = (+ (sin x) 1)
```

The other operations that behave in this way are:

```
*, /, expt, gcd, make-rectangular, make-polar
```

## Operators

Operators are a special class of functions that manipulate functions. They differ from other functions in that multiplication of operators is understood as their composition, rather than the product of their values for each input. The prototypical operator is the derivative, `D`. For an ordinary function, such as "sin"

```
((expt sin 2) x) = (expt (sin x) 2),
```

but derivative is treated differently:

```
((expt D 2) f) = (D (D f))
```

New operators can be made by combining others. So, for example, `(expt D 2)` is an operator, as is `(+ (expt D 2) (* 2 D) 3)`.

We start with a few primitive operators, the total and partial derivatives, which will be explained in detail later.

`o:identity`

derivative (also named `D`)

(partial <component-selectors>)

If `<O>` is an expression representing an operator then

```
(operator? <any>)      ==> <boolean>
(type <O>)              ==> *operator*
```

Operators can be added, subtracted, multiplied, and scaled. If they are combined with an object that is not an operator, the non-operator is coerced to an operator that multiplies its input by the non-operator.

The transcendental functions `exp`, `sin`, and `cos` are extended to take operator arguments. The resulting operators are expanded as power series.

## Power Series

Power series are often needed in mathematical computations. There are a few primitive power series, and new power series can be formed by operations on existing power series. If <p> is an expression denoting a power series then:

```
(series? <any>)          ==> <boolean>
(type <p>)                ==> *series*
```

Series can be constructed in a variety of ways. If one has a procedure that implements the general form of a coefficient then this gives the most direct method:

For example, the n-th coefficient of the power series for the exponential function is  $1/n!$ . We can write this as

```
(series:generate (lambda (n) (/ 1 (factorial n))))
```

Sometimes we have a finite number of coefficients and we want to make a series with those given coefficients (assuming zeros for all higher-order coefficients). We can do this with the extensional constructor. Thus

```
(series 1 2 3 4 5)
```

is the series whose first coefficients are the arguments given.

There are some nice initial series:

```
series:zero
  is the series of all zero coefficients
```

```
series:one
  is the series of all zero coefficients except for the first
  (constant), which is one.
```

```
(constant-series c)
  is the series of all zero coefficients except for the first
  (constant), which is the given constant.
```

```
((binomial-series a) x) = (1+x)^a
```

In addition, we provide the following initial series:

```
exp-series, cos-series, sin-series, tan-series,
cosh-series, sinh-series, atan-series
```

Series can also be formed by processes such as exponentiation of an operator or a square matrix. For example, if  $f$  is any function of one argument, and if  $x$  and  $dx$  are numerical expressions, then this expression denotes the Taylor expansion of  $f$  around  $x$ .

```
((exp (* dx D)) f) x
= (+ (f x) (* dx ((D f) x)) (* 1/2 (expt dx 2) (((expt D 2) f) x)) ...)
```

We often want to show a few ( $n$ ) terms of a series:

```
(series:print <p> <n>)
```

For example, to show eight coefficients of the cosine series we might write:

```
(series:print (((exp D) cos) 0) 8)
1
0
-1/2
0
1/24
0
-1/720
0
;Value: ...
```

We can make the sequence of partial sums of a series. The sequence is a stream, not a series.

```
(stream:for-each write-line (partial-sums (((exp D) cos) 0.)) 10)
1.
1.
.5
.5
.5416666666666666
.5416666666666666
.5402777777777777
.5402777777777777
.5403025793650793
.5403025793650793
;Value: ...
```

Note that the sequence of partial sums approaches  $(\cos 1)$ .

```
(cos 1)
;Value: .5403023058681398
```

In addition to the special operations for series, the following generic operations are defined for series

negate, invert, +, -, \*, /, expt

## Generic extensions

In addition to ordinary generic operations, there are a few important generic extensions. These are operations that apply to a whole class of datatypes, because they are defined in terms of more primitive generic operations.

```
(identity x) = x
```

```
(square x)   = (* x x)
```

```
(cube x)     = (* x x x)
```

```
(arg-shift <f> <k1> ... <kn>)
```

```
(arg-scale <f> <k1> ... <kn>)
```

Takes a function, *f*, of *n* arguments and returns a new function of *n* arguments that is the old function with arguments shifted or scaled by the given offsets or factors:

```
((arg-shift square 3) 4) ==> 49
```

```
((arg-scale square 3) 4) ==> 144
```

```
(sigma <f> <lo> <hi>)
```

Produces the sum of the values of the function *f* when called with the numbers between *lo* and *hi* inclusive.

```
(sigma square 1 5)      ==> 55
```

```
(sigma identity 1 100) ==> 5050
```

```
(compose <f1> ... <fn>)
```

Produces a procedure that computes composition of the functions represented by the procedures that are its arguments.

```
((compose square sin) 3) ==> .01991485667481699
```

```
(square (sin 3))        ==> .01991485667481699
```

## Differentiation

In this system we work in terms of functions; the derivative of a function is a function. The procedure for producing the derivative of a function is named "derivative", though we also use the single-letter symbol "D" to denote this operator.

We start with functions of a real variable to a real variable:

```
((D cube) 5) ==> 75
```

It is possible to compute the derivative of any composition of functions,

```
((D (+ (square sin) (square cos))) 3) ==> 0
```

```
(define (unity1 x)
  (+ (square (sin x)) (square (cos x))))
```

```
((D unity1) 4) ==> 0
```

```
(define (unity2
  (+ (compose square sin) (compose square cos)))
```

```
((D unity2) 4) ==> 0
```

except that the computation of the value of the function may not require evaluating a conditional.

These derivatives are not numerical approximations estimated by some limiting process. However, as usual, some of the procedures that are used to compute the derivative may be numerical approximations.

```
((D sin) 3)      ==> -.9899924966004454
(cos 3)         ==> -.9899924966004454
```

Of course, not all functions are simple compositions of univariate real-valued functions of real arguments. Some functions have multiple arguments, and some have structured values.

First we consider the case of multiple arguments. If a function maps several real arguments to a real value, then its derivative is a representation of the gradient of that function -- we must be able to multiply the derivative by an incremental up tuple to get a linear approximation to an increment of the function, if we take a step described by the incremental up tuple. Thus the derivative must be a down tuple of partial derivatives. We will talk about computing partial derivatives later.

Let's understand this in a simple case. Let  $f(x,y) = x^3 y^5$ .

```
(define (f x y)
  (* (expt x 3) (expt y 5)))
```

Then  $Df(x,y)$  is a down tuple with components  $[2 x^2 y^5, 5 x^3 y^4]$ .

```
(simplify ((D f) 2 3)) ==> (down 2916 3240)
```

And the inner product with an incremental up tuple is the appropriate increment.

```
(* ((D f) 2 3) (up .1 .2)) ==> 939.6
```

This is exactly the same as if we had a function of one up-tuple argument. Of course, we must supply an up-tuple to the derivative in this case:

```
(define (g v)
  (let ((x (ref v 0))
        (y (ref v 1)))
    (* (expt x 3) (expt y 5))))
```

```
(simplify ((D g) (up 2 3))) ==> (down 2916 3240)
```

```
(* ((D g) (up 2 3)) (up .1 .2)) ==> 939.6
```

Things get somewhat more complicated when we have functions with multiple structured arguments. Consider a function whose first argument is an up tuple and whose second argument is a number, which adds the cube of the number to the dot product of the up tuple with itself.

```
(define (h v x)
  (+ (cube x) (square v)))
```

What is its derivative? Well, it had better be something that can multiply an increment in the arguments, to get an increment in the function. The increment in the first argument is an incremental up tuple. The increment in the second argument is a small number. Thus we need a down-tuple of two parts, a row of the values of the partial derivatives with respect to each component of the first argument and the value of the partial derivative with respect to the second argument. This is easier to see symbolically:

```
(simplify ((derivative h) (up 'a 'b) 'c))
==> (down (down (* 2 a) (* 2 b)) (* 3 (expt c 2)))
```

The idea generalizes.

Partial derivatives are just the components of the derivative of a function that takes multiple arguments or structured arguments or both. Thus, a partial derivative of a function is a composition of a component selector and the derivative of that function.

The procedure that makes a partial derivative operator given a selection chain is named "partial".

```
(simplify (((partial 0) h) (up 'a 'b) 'c))
=> (down (* 2 a) (* 2 b))

(simplify (((partial 1) h) (up 'a 'b) 'c))
=> (* 3 (expt c 2))

(simplify (((partial 0 0) h) (up 'a 'b) 'c))
=> (* 2 a)

(simplify (((partial 0 1) h) (up 'a 'b) 'c))
=> (* 2 b)
```

This naming scheme is consistent, except for one special case. If a function takes exactly one up-tuple argument then one level of the hierarchy is eliminated, allowing one to naturally write:

```
(simplify ((D g) (up 'a 'b)))
=> (down (* 3 (expt a 2) (expt b 5))
      (* 5 (expt a 3) (expt b 4)))

(simplify (((partial 0) g) (up 'a 'b)))
=> (* 3 (expt a 2) (expt b 5))

(simplify (((partial 1) g) (up 'a 'b)))
=> (* 5 (expt a 3) (expt b 4))
```

## Symbolic Extensions

All primitive mathematical procedures are extended to be generic over symbolic arguments. When given symbolic arguments these procedures construct a symbolic representation of the required answer. There are primitive literal numbers. We can make a literal number that is represented as an expression by the symbol "a" as follows:

```
(literal-number 'a)          ==> (*number* (expression a))
```

The literal number is an object that has the type of a number, but its representation as an expression is the symbol "a".

```
(type (literal-number 'a))   ==> *number*
```

```
(expression (literal-number 'a)) ==> a
```

Literal numbers may be manipulated, using the generic operators.

```
(sin (+ (literal-number 'a) 3))
==> (*number* (expression (sin (+ 3 a))))
```

To make it easy to work with literal numbers, Scheme symbols are interpreted by the generic operations as literal numbers.

```
(sin (+ 'a 3)) ==> (*number* (expression (sin (+ 3 a))))
```

We can extract the numerical expression from its type-tagged representation with the "expression" procedure

```
(expression (sin (+ 'a 3))) ==> (sin (+ 3 a))
```

but usually we really don't want to look at raw expressions

```
(expression ((D cube) 'x)) ==> (+ (* x (+ x x)) (* x x))
```

because they are unsimplified. We will talk about simplification later, but "simplify" will usually give a better form,

```
(simplify ((D cube) 'x)) ==> (* 3 (expt x 2))
```

and "print-expression", which incorporates "simplify", will attempt to format the expression nicely.

Besides literal numbers, there are other literal mathematical objects, such as vectors and matrices, that can be constructed with appropriate constructors:

```
(literal-vector <name>)  
(literal-down-tuple <name>)  
(literal-up-tuple <name>)  
(literal-matrix <name>)  
(literal-function <name>)
```

There are currently no simplifiers that can manipulate literal objects of these types into a nice form.

We often need literal functions in our computations. The object produced by "(literal-function 'f)" acts as a function of one real variable that produces a real result. The name (expression representation) of this function is the symbol "f". This literal function has a derivative, which is the literal function with expression representation "(D f)". Thus, we may make up and manipulate expressions involving literal functions:

```
(expression ((literal-function 'f) 3)) ==> (f 3)  
  
(simplify ((D (* (literal-function 'f) cos)) 'a))  
  ==> (+ (* (cos a) ((D f) a)) (* -1 (f a) (sin a))))  
  
(simplify ((compose (D (* (literal-function 'f) cos))  
                    (literal-function 'g))  
            'a))  
  ==> (+ (* (cos (g a)) ((D f) (g a)))  
        (* -1 (f (g a)) (sin (g a)))))
```

We may use such a literal function anywhere that an explicit function of the same type may be used.

The Literal function descriptor language.

We can also specify literal functions with multiple arguments and with structured arguments and results. For example, to denote a literal function named `g` that takes two real arguments and returns a real value ( `g:RXR -> R` ) we may write:

```
(define g (literal-function 'g (-> (X Real Real) Real)))

(print-expression (g 'x 'y))
(g x y)
```

The descriptors for literal functions look like prefix versions of the standard function types. Thus, we write:

```
(literal-function 'g (-> (X Real Real) Real))
```

The base types are the real numbers, designated by "Real". We will later extend the system to include complex numbers, designated by "Complex".

Types can be combined in several ways. The cartesian product of types is designated by:

```
(X <type1> <type2> ...)
```

We use this to specify an argument tuple of objects of the given types arranged in the given order.

Similarly, we can specify an up tuple or a down tuple with:

```
(UP <type1> <type2> ...)
(DOWN <type1> <type2> ...)
```

We can also specify a uniform tuple of a number of elements of the same type using:

```
(UP* <type> [n])
(DOWN* <type> [n])
```

So we can write specifications of more general functions:

```
(define H
  (literal-function 'H
    (-> (UP Real (UP Real Real) (DOWN Real Real)) Real)))

(define s (up 't (up 'x 'y) (down 'p_x 'p_y)))

(print-expression (H s))
(H (up t (up x y) (down p_x p_y)))

(print-expression ((D H) s))
(down
  (((partial 0) H) (up t (up x y) (down p_x p_y)))
  (down (((partial 1 0) H) (up t (up x y) (down p_x p_y)))
        (((partial 1 1) H) (up t (up x y) (down p_x p_y))))
  (up (((partial 2 0) H) (up t (up x y) (down p_x p_y)))
       (((partial 2 1) H) (up t (up x y) (down p_x p_y))))))
```

## Numerical Methods

There are a great variety of numerical methods that are coded in Scheme and are available in the Scmutils system. Here we give a short description of a few that are needed in the Mechanics course.

## Univariate Minimization

One may search for local minima of a univariate function in a number of ways. The procedure "minimize", used as follows,

```
(minimize f lowx highx)
```

is the default minimizer. It searches for a minimum of the univariate function  $f$  in the region of the argument delimited by the values  $lowx$  and  $highx$ . Our univariate optimization programs typically return a list  $(x\ fx\ \dots)$  where  $x$  is the argument at which the extremal value  $fx$  is achieved. The following helps destructure this list.

```
(define extremal-arg car)
(define extremal-value cadr)
```

The procedure `minimize` uses Brent's method (don't ask how it works!). The actual procedure in the system is:

```
(define (minimize f lowx highx)
  (brent-min f lowx highx brent-error))

(define brent-error 1.0e-5)
```

We personally like Brent's algorithm for univariate minimization, as found on pages 79-80 of his book "Algorithms for Minimization Without Derivatives". It is pretty reliable and pretty fast, but we cannot explain how it works. The parameter "eps" is a measure of the error to be tolerated.

```
(brent-min f a b eps)
(brent-max f a b eps)
```

Thus, for example, if we make a function that is a quadratic polynomial with a minimum of 1 at 3,

```
(define foo (Lagrange-interpolation-function '(2 1 2) '(2 3 4)))
```

we can find the minimum quickly (in five iterations) with Brent's method:

```
(brent-min foo 0 5 1e-2) ==> (3. 1. 5)
```

Pretty good, eh?

Golden Section search is sometimes an effective method, but it must be supplied with a convergence-test procedure, called `good-enuf?`.

```
(golden-section-min f lowx highx good-enuf?)
(golden-section-max f lowx highx good-enuf?)
```

The predicate `good-enuf?` takes seven arguments. It is true if convergence has occurred. The arguments to `good-enuf?` are

```
lowx, minx, highx, flowx, fminx, fhighx, count
```

where `lowx` and `highx` are values of the argument that the minimum has been localized to be between, and `minx` is the argument currently being tendered. The values `flowx`, `fminx`, and `fhighx` are the values of the function at the corresponding points; `count` is the number of iterations of the search. For example, suppose we want to squeeze the minimum of the polynomial function `foo` to a difference of argument positions of `.001`.

```
(golden-section-min foo 0 5
 (lambda (lowx minx highx flowx fminx fhighx count)
  (< (abs (- highx lowx)) .001)))

=> (3.0001322139227034 1.0000000174805213 17)
```

This is not so nice. It took 17 iterations and we didn't get anywhere near as good an answer as we got with Brent. On the other hand, we understand how this works!

We can find a number of local minima of a multimodal function using a search that divides the initial interval up into a number of subintervals and then does Golden Section search in each interval. For example, we may make a quartic polynomial:

```
(define bar
 (Lagrange-interpolation-function '(2 1 2 0 3) '(2 3 4 5 6)))
```

Now we can look for local minima of this function in the range `-10` to `+10`, breaking the region up into 15 intervals as follows:

```
(local-minima bar -10 10 15 .0000001)
=> ((5.303446964995252 -.32916549541536905 18)
 (2.5312725379910592 .42583263999526233 18))
```

The search has found two local minima, each requiring 18 iterations to localize. The local maxima are also worth chasing:

```
(local-maxima bar -10 10 15 .0000001)
=> ((3.8192274368217713 2.067961961032311 17)
 (10 680 31)
 (-10 19735 29))
```

Here we found three maxima, but two are at the endpoints of the search.

## Multivariate minimization

The default multivariate minimizer is multidimensional-minimize, which is a heavily sugared call to the Nelder-Mead minimizer. The function `f` being minimized is a function of a Scheme list. The search starts at the given initial point, and proceeds to search for a point that is a local minimum of `f`. When the process terminates, the continuation function is called with three arguments. The first is true if the process converged and false if the minimizer gave up. The second is the actual point that the minimizer has found, and the third is the value of the function at that point.

```
(multidimensional-minimize f initial-point continuation)
```

Thus, for example, to find a minimum of the function

```
(define (baz v)
  (* (foo (ref v 0)) (bar (ref v 1))))
```

made from the two polynomials we constructed before, near the point `(4 3)`, we can try:

```
(multidimensional-minimize baz '(4 3) list)
=> (#t #(2.9999927603607803 2.5311967755369285) .4258326193383596)
```

Indeed, a minimum was found, at about  `#(3 2.53)` with value `.4258...`

Of course, we usually need to have more control of the minimizer when searching a large space. Without the sugar, the minimizers act on functions of Scheme vectors (not lists, as above). The simplest minimizer is the Nelder Mead downhill simplex method, a slow but reasonably reliable method.

```
(nelder-mead f start-pt start-step epsilon maxiter)
```

We give it a function, a starting point, a starting step, a measure of the acceptable error, and a maximum number of iterations we want it to try before giving up. It returns a message telling whether it found a minimum, the place and value of the purported minimum, and the number of iterations it performed. For example, we can allow the algorithm an initial step of 1, and it will find the minimum after 21 steps

```
(nelder-mead baz #(4 3) 1 .00001 100)
=> (ok #(2.9955235887900926 2.5310866303625517) .4258412014077558) 21)
```

or we can let it take steps of size 3, which will allow it to wander off into oblivion

```
(nelder-mead baz #(4 3) 3 .00001 100)
==> (maxcount
      (#(-1219939968107.8127 5.118445485647498) . -2.908404414767431e23)
      100)
```

The default minimizer uses the values:

```
(define nelder-start-step .01)
(define nelder-epsilon 1.0e-10)
(define nelder-maxiter 1000)
```

If we know more than just the function to minimize we can use that information to obtain a better minimum faster than with the Nelder-Mead algorithm.

In the Davidon-Fletcher-Powell algorithm,  $f$  is a function of a single vector argument that returns a real value to be minimized,  $g$  is the vector-valued gradient of  $f$ ,  $x_0$  is a (vector) starting point, and estimate is an estimate of the minimum function value.  $ftol$  is the convergence criterion: the search is stopped when the relative change in  $f$  falls below  $ftol$  or when the maximum number of iterations is exceeded.

The procedure `dfp` uses Davidon's line search algorithm, which is efficient and would be the normal choice, but `dfp-brent` uses Brent's line search, which is less efficient but more reliable. The procedure `bfgs`, due to Broyden, Fletcher, Goldfarb, and Shanno, is said to be more immune than `dfp` to imprecise line search.

```
(dfp f g x0 estimate ftol maxiter)
(dfp-brent f g x0 estimate ftol maxiter)
(bfgs f g x0 estimate ftol maxiter)
```

These are all used in the same way:

```
(dfp baz (compose down->vector (D baz)) #(4 3) .4 .00001 100)
==> (ok (#(2.9999717563962305 2.5312137271310036) . .4258326204265246) 4)
```

They all converge very fast, four iterations in this case.

## Quadrature

Quadrature is the process of computing definite integrals of functions. A sugared default procedure for quadrature is provided, and we hope that it is adequate for most purposes.

```
(definite-integral <integrand>
  <lower-limit> <upper-limit>
  [compile? #t/#f])
```

The integrand must be a real-valued function of a real argument. The limits of integration are specified as additional arguments. There is an optional fourth argument that can be used to suppress compilation of the integrand, thus forcing it to be interpreted. This is usually to be ignored.

Because of the many additional features of numerical quadrature that can be independently controlled we provide a special uniform interface for a variety of mechanisms for computing the definite integrals of functions. The quadrature interface is organized around definite-integrators. A definite integrator is a message-acceptor that organizes the options and defaults that are necessary to specify an integration plan.

To make an integrator, and to give it the name I, do:

```
(define I (make-definite-integrator))
```

You may have as many definite integrators outstanding as you like. An definite integrator can be given the following commands:

```
(I 'integrand)
  returns the integrand assigned to the integrator I.
(I 'set-integrand! <f>)
  sets the integrand of I to the procedure <f>.
```

The integrand must be a real-valued function of one real argument.

```
(I 'lower-limit)
  returns the lower integration limit.
(I 'set-lower-limit! <ll>)
  sets the lower integration limit of the integrator to <ll>.
```

```
(I 'upper-limit)
  returns the upper integration limit.
(I 'set-upper-limit! <ul>)
  sets the upper integration limit of the integrator to <ul>.
```

The limits of integration may be numbers, but they may also be the special values `:+infinity` or `:-infinity`.

```
(I 'integral)
  performs the integral specified and returns its value.

(I 'error)
  returns the value of the allowable error of integration.
(I 'set-error! <epsilon>)
  sets the allowable error of integration to <epsilon>.
  The default value of the error is 1.0e-11.

(I 'method)
  returns the integration method to be used.
(I 'set-method! <method>)
  sets the integration method to be used to <method>.
  The default method is open-open.
  Other methods are      open-closed, closed-open, closed-closed
                        romberg, bulirsch-stoer
```

The quadrature methods are all based on extrapolation. The Romberg method is a Richardson extrapolation of the trapezoid rule. It is usually worse than the other methods, which are adaptive rational function extrapolations of trapezoid and Euler-MacLaurin formulas.

Closed integrators are best if we can include the endpoints of integration. This cannot be done if the endpoint is singular: thus the open formulas. Also, open formulas are forced when we have infinite limits.

Let's do an example, it is as easy as pi!

```
(define witch
  (lambda (x)
    (/ 4.0 (+ 1.0 (* x x)))))

(define integrator (make-definite-integrator))

(integrator 'set-method! 'romberg)
(integrator 'set-error! 1e-12)
(integrator 'set-integrand! witch)
(integrator 'set-lower-limit! 0.0)
(integrator 'set-upper-limit! 1.0)
(integrator 'integral)
;Value: 3.141592653589793
```

## Programming with optional arguments

Definite integrators are so common and important that, to make the programming a bit easier we allow one to be set up slightly differently. In particular, we can specify the important parameters as optional arguments to the maker. The specification of the maker is:

```
(make-definite-integrator [#!optional integrand
                           lower-limit upper-limit
                           allowable-error
                           method])
```

So, for example, we can investigate the following integral easily:

```
(define (foo n)
  ((make-definite-integrator
    (lambda (x) (expt (log (/ 1 x)) n))
    0.0 1.0
    1e-12 'open-closed)
   'integral))
```

```
(foo 0)
;Value: 1.
```

```
(foo 1)
;Value: .9999999999979357
```

```
(foo 2)
;Value: 1.9999999999979101
```

```
(foo 3)
;Value: 5.99999999999799
```

```
(foo 4)
;Value: 23.999999999997893
```

```
(foo 5)
;Value: 119.99999999999828
```

Do you recognize this function? What is (foo 6)?

## ODE Initial Value Problems

Initial-value problems for ordinary differential equations can be attacked by a great many specialized methods. Numerical analysts agree that there is no best method. Each has situations where it works best and other situations where it fails or is not very good. Also, each technique has numerous parameters, options and defaults. The default integration method is Bulirsch-Stoer. Usually, the Bulirsch-Stoer algorithm will give better and faster results than others, but there are applications where a quality-controlled trapezoidal method or a quality-controlled 4th order Runge-Kutta method is appropriate. The algorithm used can be set by the user:

```
(set-ode-integration-method! 'qcrk4)
(set-ode-integration-method! 'bulirsch-stoer)
(set-ode-integration-method! 'qcctrp2)
(set-ode-integration-method! 'explicit-gear)
```

The integration methods all automatically select the step sizes to maintain the error tolerances. But if we have an exceptionally stiff system, or a bad discontinuity, for most integrators the step size will go down to zero and the integrator will make no progress. If you encounter such a disaster try explicit-gear.

We have programs that implement other methods of integration, such as an implicit version of Gear's stiff solver, and we have a whole language for describing error control, but these features are not available through this interface.

The two main interfaces are "evolve" and "state-advancer".

The procedure "state-advancer" is used to advance the state of a system according to a system of first order ordinary differential equations for a specified interval of the independent variable. The state may have arbitrary structure, however we require that the first component of the state is the independent variable.

The procedure "evolve" uses "state-advancer" to repeatedly advance the state of the system by a specified interval, examining aspects of the state as the evolution proceeds.

In the following descriptions we assume that "sysder" is a user provided procedure that gives the parametric system derivative. The parametric system derivative takes parameters, such as a mass or length, and produces a procedure that takes a state and returns the derivative of the state. Thus, the system derivative takes arguments in the following way:

```
((sysder parameter-1 ... parameter-n) state)
```

There may be no parameters, but then the system derivative procedure must still be called with no arguments to produce the procedure that takes states to the derivative of the state.

For example, if we have the differential equations for an ellipse centered on the origin and aligned with the coordinate axes:

$$\begin{aligned} Dx(t) &= -a y(t) \\ Dy(t) &= +b x(t) \end{aligned}$$

We can make a parametric system derivative for this system as follows:

```
(define ((ellipse-sysder a b) state)
  (let ((t (ref state 0))
        (x (ref state 1))
        (y (ref state 2)))
    (up 1
        (* -1 a y) ; dt/dt
        (* b x))) ; dx/dt
        ; dy/dt
```

The procedure "evolve" is invoked as follows:

```
((evolve sysder . parameters) initial-state monitor dt final-t eps)
```

The user provides a procedure (here named "monitor") that takes the state as an argument. The monitor is passed successive states of the system as the evolution proceeds. For example it might be used to print the state or to plot some interesting function of the state.

The interval between calls to the monitor is the argument "dt". The evolution stops when the independent variable is larger than "final-t". The parameter "eps" specifies the allowable error.

For example, we can draw our ellipse in a plotting window:

```
(define win (frame -2 2 -2 2 500 500))

(define ((monitor-xy win) state)
  (plot-point win (ref state 1) (ref state 2)))

((evolve ellipse-sysder 0.5 2.)
 (up 0. .5 .5) ; initial state
 (monitor-xy win) ; the monitor
 0.01 ; plotting step
 10.) ; final value of t
```

To take more control of the integration one may use the state advancer directly.

The procedure "state-advancer" is invoked as follows:

```
((state-advancer sysder . parameters) start-state dt eps)
```

The state advancer will give a new state resulting from evolving the start state by the increment dt of the independent variable. The allowed local truncation error is eps.

For example,

```
((state-advancer ellipse-sysder 0.5 2.0) (up 0. .5 .5) 3.0 1e-10)
;Value: #(3. -0.5302762503146702 -0.3538762402420404)
```

For a more complex example that shows the use of substructure in the state, consider two-dimensional harmonic oscillator:

```
(define ((harmonic-sysder m k) state)
  (let ((q (coordinate state)) (p (momentum state)))
    (let ((x (ref q 0)) (y (ref q 1))
          (px (ref p 0)) (py (ref p 1)))
      (up 1
         (up (/ px m) (/ py m)) ;dq/dt
         (down (* -1 k x) (* -1 k y)) ;dp/dt
         ))))
```

We could monitor the energy (the Hamiltonian):

```
(define ((H m k) state)
  (+ (/ (square (momentum state))
        (* 2 m))
      (* 1/2 k
         (square (coordinate state)))))

(let ((m 0.5) (k 2.0))
  ((evolve harmonic-sysder m k)
   (up 0. ; initial state
      (up .5 .5)
      (down 0.1 0.0))
   (lambda (state) ; the monitor
     (write-line
      (list (time state) ((H m k) state))))
   1.0 ; output step
   10.))

(0. .51)
(1. .5099999999999999)
(2. .5099999999999997)
(3. .5099999999999992)
(4. .5099999999999997)
(5. .5099999999999997)
(6. .5099999999999973)
(7. .5099999999999975)
(8. .5100000000000032)
(9. .5100000000000036)
(10. .5100000000000033)
```

## Constants

There are a few constants that we find useful, and are thus provided in Scmutils. Many constants have multiple names.

There are purely mathematical constants:

```
(define zero 0)                (define :zero zero)
(define one 1)                 (define :one one)
(define -one -1)               (define :-one -one)
(define two 2)                 (define :two two)
(define three 3)               (define :three three)

(define pi (* 4 (atan 1 1)))    (define :pi pi)
(define -pi (- pi))            (define :+pi pi)
                                (define :-pi -pi)

(define pi/6 (/ pi 6))         (define :pi/6 pi/6)
(define -pi/6 (- pi/6))        (define :+pi/6 pi/6)
                                (define :-pi/6 -pi/6)

(define pi/4 (/ pi 4))         (define :pi/4 pi/4)
(define -pi/4 (- pi/4))        (define :+pi/4 pi/4)
                                (define :-pi/4 -pi/4)

(define pi/3 (/ pi 3))         (define :pi/3 pi/3)
(define -pi/3 (- pi/3))        (define :+pi/3 pi/3)
                                (define :-pi/3 -pi/3)

(define pi/2 (/ pi 2))         (define :pi/2 pi/2)
(define -pi/2 (- pi/2))        (define :+pi/2 pi/2)
                                (define :-pi/2 -pi/2)

(define 2pi (+ pi pi))         (define :2pi 2pi)
(define -2pi (- 2pi))          (define :+2pi 2pi)
                                (define :-2pi -2pi)
```

For numerical analysis, we provide the smallest number that when added to 1.0 makes a difference.

```
(define *machine-epsilon*
  (let loop ((e 1.0))
    (if (= 1.0 (+ e 1.0))
        (* 2 e)
        (loop (/ e 2)))))

(define *sqrt-machine-epsilon*
  (sqrt *machine-epsilon*))
```

## Useful units conversions

```
(define arcsec/radian
  (/ (* 60 60 360) :+2pi))
```

```
;;; Physical Units
```

```
(define kg/amu
  1.661e-27)
```

```
(define joule/eV
  1.602e-19)
```

```
(define joule/cal
  4.1840)
```

```
;;; Universal Physical constants
```

```
(define light-speed           ;c
  2.99792458e8                ;meter/sec)
```

```
(define :c light-speed)
```

```
(define esu/coul
  (* 10 light-speed))
```

```
(define gravitation           ;G
  6.6732e-11                  ;(Newton*meter^2)/kg^2)
```

```
(define :G gravitation)
```

```
(define electron-charge      ;e
  1.602189e-19                ;Coulomb
  ;:=4.80324e-10 esu)
```

```
(define :e electron-charge)
```

```
(define electron-mass        ;m_e
  9.10953e-31                 ;kg)
```

```
(define :m_e electron-mass)
```

```
(define proton-mass          ;m_p
  1.672649e-27)             ;kg

(define :m_p proton-mass)

(define neutron-mass         ;m_n
  1.67492e-27)             ;kg

(define :m_n neutron-mass)

(define planck                ;h
  6.62618e-34)             ;Joule*sec

(define :h planck)

(define h-bar                 ;\bar{h}
  (/ planck :+2pi))        ;Joule*sec

(define :h-bar h-bar)

(define permittivity          ;\epsilon_0
  8.85419e-12)             ;Coulomb/(volt*meter)

(define :epsilon_0 permittivity)

(define boltzman              ;k
  1.38066e-23)             ;Joule/Kelvin

(define :k boltzman)

(define avogadaro             ;N_A
  6.02217e23)              ;1/mol

(define :N_A avogadaro)
```

```
(define faraday                ;F
  9.64867e4)                   ;Coulomb/mol

(define gas                    ;R
  8.31434)                     ;Joule/(mol*Kelvin)

(define :R gas)

(define gas-atm                ;R
  8.2054e-2)                   ;(liter*atm)/(mol*Kelvin)

(define radiation
  (/ (* 8 (expt pi 5) (expt boltzman 4))
     (* 15 (expt light-speed 3) (expt planck 3))))

(define stefan-boltzman
  (/ (* light-speed radiation) 4))

(define thomson-cross-section
  6.652440539306967e-29)      ;m^2

;;; (define thomson-cross-section
;;;   (/ (* 8 pi (expt electron-charge 4))
;;;      (* 3 (expt electron-mass 2) (expt light-speed 4))))
;;; in the SI version of ESU.

;;; Observed and measured

(define background-temperature ;Cobe 1994
  2.726)                       ;+- .005 Kelvin

;;; Thermodynamic

(define water-freezing-temperature
  273.15)                      ;Kelvin

(define room-temperature
  300.00)                      ;Kelvin

(define water-boiling-temperature
  373.15)                      ;Kelvin
```

```
;;; Astronomical

(define m/AU
  1.49597892e11)

(define m/pc
  (/ m/au (tan (/ 1 arcsec/radian))))

(define AU/pc
  (/ 648000 pi))                               ;=(/ m/pc m/AU)

(define sec/sidereal-yr
  3.1558149984e7)                               ;1900

(define sec/tropical-yr
  31556925.9747)                               ;1900

(define m/lyr
  (* light-speed sec/tropical-yr))

(define AU/lyr (/ m/lyr m/AU))

(define lyr/pc (/ m/pc m/lyr))

(define m/parsec
  3.084e16)

(define m/light-year
  9.46e15)

(define sec/day
  86400)
```

```
;;; Earth

(define earth-orbital-velocity
  29.8e3)                                ;meter/sec

(define earth-mass
  5.976e24)                              ;kg

(define earth-radius
  6371e3)                                ;meters

(define earth-surface-area
  5.101e14)                              ;meter^2

(define earth-escape-velocity
  11.2e3)                                ;meter/sec

(define earth-grav-accel
  9.80665)                              ;g
                                          ;meter/sec^2

(define earth-mean-density
  5.52e3)                                ;kg/m^3

;;; This is the average amount of
;;; sunlight available at Earth on an
;;; element of surface normal to a
;;; radius from the sun. The actual
;;; power at the surface of the earth,
;;; for a panel in full sunlight, is
;;; not very different, because, in
;;; absence of clouds the atmosphere
;;; is quite transparent. The number
;;; differs from the obvious geometric
;;; number
;;; (/ sun-luminosity (* 4 :pi (square m/AU)))
;;; ;Value: 1360.454914748201
;;; because of the eccentricity of the
;;; Earth's orbit.

(define earth-incident-sunlight
  1370.)                                ;watts/meter^2

(define vol@stp
  2.24136e-2)                            ;(meter^3)/mol

(define sound-speed@stp
  331.45)                                ;c_s
                                          ;meter/sec

(define pressure@stp
  101.325)                               ;kPa

(define earth-surface-temperature
  (+ 15 water-freezing-temperature))
```

```
;;; Sun
```

```
(define sun-mass  
  1.989e30) ;kg
```

```
(define :m_sun sun-mass)
```

```
(define sun-radius  
  6.9599e8) ;meter
```

```
(define :r_sun sun-radius)
```

```
(define sun-luminosity  
  3.826e26) ;watts
```

```
(define :l_sun sun-luminosity)
```

```
(define sun-surface-temperature  
  5770.0) ;Kelvin
```

```
(define sun-rotation-period  
  2.14e6) ;sec
```

```
;;; The Gaussian constant
```

```
(define GMsun ;>(* gravitation sun-mass)  
  1.32712497e20)
```

MORE TO BE DOCUMENTED

Solutions of Equations

Linear Equations (lu, gauss-jordan, full-pivot)

Linear Least Squares (svd)

Roots of Polynomials

Searching for roots of other nonlinear disasters

Matrices

Eigenvalues and Eigenvectors

Series and Sequence Extrapolation

Special Functions

Displaying results

Lots of other stuff that we cannot remember.