Common Music 3

Introduction

Common Music 3 is a software environment for real time and score file (non-real time) algorithmic composition. It is implemented in JUCE, by Julian Storer, and S7 Scheme / SndLib, by William Schottstaedt.

The distinguishing features of Common Music 3 (CM3) compared with the previous (CM2) branch are:

- Seamless integration of Scheme (dynamic scripting) and C++ .
- Audio Ports for real-time work.
- A dedicated scheduling thread for running processes (musical algorithms) in real time.
- True drag-and-drop executables: all runtime sources, examples, instruments and documentation are embedded in the app itself.
- SAL, a simple, infix, minimalist language for algorithmic composition. In documentation Scheme formals are only shown when they differ in some significant way from the SAL versions.
- Simpler API, no object system or generic functions.
- Terser calling syntax to facilitate interactive coding:
  - No dependence on quoted symbols or keywords in argument data.

- Many functions automatically handle list mapping to reduce the need for ad hoc looping in order to process list data. (In the documentation separate formals are provided in cases where the automatic list handling results a significant difference in argument syntax.)
- Opt/key function parameters.
- Shorter function names (in general).
- Note names in the standard chromatic scale also support quarter-tone inflection.

Composition Toolbox

The following functions have been added to Scheme to facilitate algorithmic music composition.

Numbers

fit (*num*, *min*, *max* &opt *mode* = *1*)
→ number or list

Returns *num* or list of same bounded by *min* and *max* according to *mode*. If *min* <= *num* <= *max* then *num* is returned. Otherwise, if *mode* is 1 then the outlying *num* is wrapped between *min* and *max*, i.e. fit returns the remainder of number modulus the boundary range. If *mode* is 2, then the *min* and *max* boundaries reflect the outlying value back into range. If *mode* is 3 then the boundary value (*min* or *max*) is returned.

quantize (*num*, *step*)
→ number or list

Quantizes *num* or list of the same to be a multiple of *step*.

decimals (*num*, *places*)
→ number or list

Returns floating point *num* rounded to *places* fractional digits.

plus (*num...*)
plus (*list*)
plus (*list*, *num*)
→ number

Returns the addition of zero or more *nums*. If a single *list* of numbers is given they are summed together. Otherwise a list is returned of *num* added to every element in *list*.

times (*num...*)
times (*list*)
times (*list*, *num*)
→ number

Returns the multiplication of zero or more *nums*. If a single *list* of numbers is given they are multiplied together. Otherwise a list is returned of *num* multiplied with every element in *list*.

minus (*num ...*)
minus (*list*)
minus (*list*, *num*)
minus (*num*, *list*)
→ number

Returns the subtraction of one or more *nums*. If a single number is given its inverse is returned. If a single *list* of numbers is given the cumulative subtraction is returned. Otherwise a list is returned of *num* subtracted from every element in *list* or every number in *list* subtracted from *num*.

```
divide (num...)
divide (list)
divide (list, num)
divide (num, list)
```
→ number

Returns the division of one or more *nums*. If a single number is given its inverse is returned. If a single *list* of numbers is given the cumulative division is returned. Otherwise a list is returned of every number in *list* divided by *num* or every number in *list* divided into *num*.

```
mod (num1, num2)
mod (list, num2)
```
→ number

Returns *num1* modulo *num2*. If a *list* of numbers is given a list of modulo are returned.

```
minimum (num...)
minimum (list)
```
→ number

Returns the smallest of one or more *nums*. If a *list* of numbers is given the minimum of the list is returned.

```
maximum (num...)
maximum (list)
```
→ number

Returns the larger of one or more *nums*. If a *list* of numbers is given the maximum of the list is returned.

```
deltas (num...)
deltas (list)
```
→ list

Returns a list of the deltas (differences) between adjacent numbers in *nums*. The number of items returned will therefore be one less than the number of items specified to the function.

```
less? (num...)
less? (list)
```
→ boolean

Returns true if the numbers in *nums...* or in a single *list* of numbers are in monotonically increasing order, otherwise returns false.

```
greater? (num...)
greater? (list)
```
→ boolean

Returns true if the numbers in *nums...* or in a single *list* of numbers are in monotonically decreasing order, otherwise returns false.

```
less=? (num...)
less=? (list)
```
→ boolean

Returns true if the numbers in *nums...* or in a single *list* of numbers are in monotonically nondecreasing order, otherwise returns false.

```
greater=? (num...)
```

`greater=?` (*list*)
→ boolean

Returns true if the numbers in *nums...* or in a single *list* of numbers are in monotonically non-increasing order, otherwise returns false.

`log10` (*num*)
→ number

Returns log 10 of *num*.

`log2` (*num*)
→ number

Returns log 2 of *num*.

`pi`
→ 3.141592653589793

The value of *pi* as a double float.

`most-positive-fixnum`
→ Implementation dependent value

The largest positive fixed point integer representable in this Lisp implementation.

`most-negative-fixnum`
→ Implementation dependent value

The largest negative fixed point integer representable in this Lisp implementation.

Lists

`first` (*list*)
→ element

Returns the first element in *list*. Similar accessors exist for these other common element positions: `second`, `third`, `fourth`, `fifth`, `sixth`, `seventh`, `eighth`, `ninth` and `tenth`.

`last` (*list*)
→ element

Returns the last element in *list*. Note that this is not the same as Common Lisp's version (see `tail`).

`nth` (*list*, *pos*)
→ element

Returns the element at *pos* position (zero based) in *list*. Note that this is not the same as Common Lisp's version which (confusingly) reverses the arguments.

`butlast` (*list*)
→ list

Returns a list of all but the last element in *list*.

`rest` (*list*)
→ list

Returns a list of all but the first element in *list*.

tail (*list*)
→ list

Returns the tail (last cdr) of *list*.

concat (*...*)
→ list

Concatenates its inputs, which can be lists and non-lists, into a single list.

list-set! (*list*, *pos*, *val*)
→ element

Sets the element at *pos* position in *list* to *val*.

list-intersection (*list1*, *list2* &optkey *test* = *equal?*, *getter* = *#f*)
→ list

Returns the elements in *list1* that are also in *list2*. If *test* is specified it should be a function of two arguments. It is passed two elements and returns true if the values are considered the same otherwise false. If *getter* is specified it should be a function of one argument. It is passed list elements and returns the objects that the *test* function will compare.

list-union (*list1*, *list2* &optkey *test* = *equal?*, *getter* = *#f*)
→ list

Returns elements that are in either *list1* or *list2* and with duplicate elements appearing only once. See list-intersection for an explanation of *test* and *getter*.

list-difference (*list1*, *list2* &optkey *test* = *equal?*, *getter* = *#f*)
→ list

Returns the elements that are in either but not both of *list1* and *list2*. See list-intersection for an explanation of *test* and *getter*.

remove-duplicates (*list1* &optkey *test* = *equal?*, *getter* = *#f*)
→ list

Returns the elements of *list1* with none repeated. See list-intersection for an explanation of *test* and *getter*.

qsort! (*sequence* &opt *comparator* = less?)
→ sequence

Sorts and returns *sequence* (a list or vector) given a pairwise *comparator* function. The comparator is passed pairs of values from the list and should return true if the first value is to be placed before the second value in the sorted results, otherwise false. The default comparator sorts the list in ascending order. Sorting is done "in place", that is, the sequence you pass in is destructively modified to contain the sorted results and also returned as the value of the function.

Mapping and Transformation

note (*ref*)
→ string or list

Returns the note (string) of *ref*, which can be a note, keynum or list of the same. A note name consists of a pitch class letter (c d e f g a b), an optional sharp or flat letter (s f) an optional quarter tone inflection (< or >) and an octave number (00 to 9). Note names can be specified

as strings, symbols or keywords. Octave numbers inside note lists can be omitted, in which case the current note's octave defaults to the last octave specified, or to octave 4.

keynum (*ref*)
→ number or list

Returns the key number of *ref*, which can be a note, hertz or list of the same. A floating point key number *kkk.cc* is interpreted as the frequency *cc* cents above the integer key number *kkk*, e.g. 60.5 is 50 cents above Middle C.

hertz (*ref*)
→ number or list

Returns the hertz value of *ref*, which can be a note, key number or list of the same.

pitch-class (*ref*)
→ number or list

Returns pitch class (0-11) of *ref*, which can be a note, key number or list of the same.

rest? (*ref*)
→ bool

Returns true if *ref* is a 'musical rest' marker (`r` or the key number -1).

transpose (*ref*, *amount*)
→ number or list

Transposes *ref* by *amount*. If *freq* and *amount* are both pitch classes then the transposition occurs in pitch class (mod 12) space. Otherwise *nkl* should be a key number, note or list of the same and it is transposed by *amount*.

invert (*ref*)
→ number or list

If *ref* is a pitch class or list of pitch classes the pitch class complement(s) are returned. Otherwise *nkl* should be a list of notes or key numbers and the (strict) inversions around the first key number or note in the list is returned.

retrograde (*list*)
→ list

Returns a version of *list* with its elements reversed.

scale (*num*, *start*, *step...*)
scale (*num*, *start*, *list* &opt *limit = #f*, *mode = 1*)
→ list

Returns a list of *num* key numbers starting on *start* and incremented by each *step* (positive or negative) or *list* of steps provided. If a *limit* key is provided then scale degrees will always lie between *start* and *limit* according to *mode* (see `fit`).

scale-order (*list* &opt *mode = 1*)
→ list

Returns a *list* of notes, key numbers or hertz values ordered according to *mode*, where 1 means ascending order, -1 means descending order and 0 means random order.

rhythm (*rhy* &opt *tempo = 60*, *beat = 1/4*)
→ number

Returns the value of *rhy* converted to beat units (typically seconds) at a specified tempo. *Rhy* can be any fraction of a whole note, a *rhythmic symbol* or a list of the same. A fraction of a whole note is simply a ratio, e.g. 1/4=quarter note, 1/8=eighth note, 3/2=three half-notes, 7/25=seven twenty-fifths notes, and so on. A rhythmic symbol consists of a metric letter (`w`=whole,`h`=half, `q`=quarter, `e`=eighth, `s`=sixteenth, `t`=thirty-second, `x`=sixty-fourth) optionally preceded by a triplet marker (`t`) and optionally followed by one or more dots (`.`). Expressions involving symbols can be formed using `+`, `-`, `*`, `/` to join the rhythmic tokens. Examples: `q`=quarter, `tq`=triplet quarter, `e.`=dotted-eighth, `ts.`=triplet dotted sixteenth, `h...`=triple dotted half, `s+tq`=sixteenth plus triplet quarter, `w-ts`=whole minus triplet sixteenth, `w*4`=four whole notes.

in-tempo (*beats*, *tempo*)
→ number

Scales a *beat* value by a metronome *tempo* value: *beats**(60/*tempo*)

rescale (*x*, *x1*, *x2*, *y1*, *y2* &opt *base* = *1*)
→ number or list

Returns a floating point value between *y1* and *y2* that is proportional to a floating point *x* or list of the same lying between *x1* and *x2*. If *base* is 1 the rescaling is linear otherwise its exponential where *base* > 1 produce increasing spread as *x* approaches *x2* and values < 1 produce decreasing spread.

discrete (*x*, *x1*, *x2*, *i1*, *i2*, &opt *base* = *1*)
discrete (*x*, *x1*, *x2*, *list*)
→ integer, element or list

Similar to `rescale` except floating point *x* is mapped onto integer values between *i1* and *i2*, or onto the elements in *list*.

segs (*num*, *sum* &opt *mode* = *1*, *base* = *2*)
→ list

Returns *num* segments adding up to *sum*, where the relationship between each segment is determined by *mode*. If *mode* is 1 then the segments are exponentially related, where *base* values greater than 1 produce exponentially increasing segments, and reciprocals produce exponentially decreasing segments. If *mode* is 2 then the segments returned are related by the geometric series. If *mode* is 3 then random segments that add up to *sum* are returned.

interp (*x*, *x1*, *y1* ...)
interp (*x*, *list* &opt *base* = *1*)
→ number

Returns the interpolated y value of *x* in a series of *x1 y1 ...* coordinate pairs or envelope list of the same. If the coordinates are specified as a list then an optional *base* can control the type of interpolation (linear or exponential) used. See `rescale` for more information.

ratio->cents (*scaler*)
→ number or list

Converts a frequency ratio *scaler* or list of the same to the equivalent in cents.

ratio->steps (*scaler*)
→ number or list

Converts a frequency ratio *scaler* or list of the same to the equivalent in half-steps.

cents->ratio (*cents*)
→ number or list

Converts a *cent* value or list of the same to the equivalent frequency scaler.

harmonics (*h1*, *h2* &optkey *fund = 1*, *invert = #f*, *order = #f*, *keys = #f*)
→ list

Returns a list of ratios, hertz values, or key numbers based on the harmonic series between the harmonic numbers *h1* and *h2* inclusive. If 0 < *h1* < *h2* then values will belong to the overtone series, otherwise if 0 > *h1* > *h2* then values will belong to the "undertone" series. If *fund* is 1 then the values returned will be frequency ratios, otherwise the series will be based on that hertz value. If *invert* is true then the ratios between *h1* and *h2* are reciprocals, i.e. "flipped". If *order* is false then the overtone series are in increasing order and undertones are in decreasing order. Otherwise, if *order* is 1 then the list is always ascending, if it is -1 the list always descending, and if it is 0 the list order is randomized. If *keys* is true then the values returned will be coerced to key numbers.

golden (*num1*, *num2* &opt *inv? = #f*)
→ number

Returns the golden mean between *num1* and *num2*. If *inv?* is true the inverse of the golden mean is used.

fibonacci (*nth*)
→ number

Returns the *nth* number in the fibonacci series.

Randomness

All functions that perform random selection use a common random number generator whose current seed can be acquired or set using random-seed. Reseeding the random generator will cause the random number generator to repeat the same sequence of numbers based on that seed.

Uniform Distribution

random (*num*)
→ number

Returns a random number between 0 and *num* (exclusive). If *num* is an integer then an integer will be returned otherwise a floating point value is returned.

between (*a*, *b*)
→ number

Returns a randomly chosen number between *a* and *b* (exclusive). The value of *b* can be less than *a*.

vary (*value*, *pct* &opt *mode = 0*)
→ number or list

Returns a random number that deviates from *value* or list of the same by up to *pct* (1=100%) according to *mode*. If *mode* is 0 then *value* is at the center of what could be returned. If *mode* is -1 then *value* is the maximum possible value, otherwise *mode* 1 places *value* at the minimum of what could be returned.

drunk (*num*, *width* &optkey *low = #f*, *high = #f*, *mode = #f avoid = #f*)
→ number

Returns a random value constrained to lie between *num* - *width* and *num* + *width*. Calling the function and passing the previous value of *num* implements a random walk (see also ranbrown). If *low* and *high* are provided they specify lower and upper bounds on the value

returned. If the value exceeds either limit then the *mode* determines how the out-of-bounds value is handled. Possible values for *mode* are:

- -1 or :reflect, value is reflected back into bounds
- 0 or :stop, returns false as value of function
- 1 or :limit, nearest boundary value is returned
- 2 or :reset, value is centered between boundaries
- 3 or :jump, value randomly selected between boundaries

If *avoid* is a value between -width and width then random value chosen will never be equal to it.

```
odds (num &opt true = #t, false = #f)
```
→ value

Returns *true* value if a randomly chosen number between 0.0 and 1.0 is less than *num* else returns *false* value.

```
pick (arg...)
pick (list)
```
→ value

Returns a randomly selected *arg* as the value of the function. If just one argument is given it should be a *list* and a randomly selected value from the list is returned.

```
shuffle (args...)
shuffle (list)
```
→ list

Returns a version of *args...* or a *list* of the same with the elements randomly reordered.

```
shuffle! (list)
```
→ list

Like shuffle except that *list* is destructively modified by the random reordering.

```
tendency (x, low, high &opt gen = ran, args...)
```
→ void

Returns a random lying between the values of *x* in the *low* and *high* envelopes. If *low* or *high* are constant for any *x* they may be numbers rather than envelopes. *Gen* is an optional random function that will be applied to *args...* to generate the random number between *low* and *high*.

```
random-series (num, low, high &optkey series = {}, reject = #f, order = #f,
chooser = random )
```
→ list

Returns a list of *num* random values between *low* and *high* (exclusive). If *series* is specified it is the initial contents of the series to be returned by the function. By default values between *low* and *high* are randomly chosen and added to the series. If a *reject* function is specified it is passed candidate values and the series so far and if it returns true the value will not be added to the series (else it will be added). If a *sorter* function is specified it is passed two values and it returns true if the first value should appear before the second in the series returned by the function. If a *chooser* function is specified it is used as the random number generator. It is passed the range *high-low* and should return a number between 0 and *range-1*.

Nonuniform Distributions

```
ranpink ()
```
→ float

Returns a pinkish (1/f) value between -1 and 1.

`ranbrown ()`
→ float

Returns a brownish ($1/f^2$) value between -1 and 1.

`ranlow ()`
→ float

Returns a value between 0 and 1 with lower values more likely.

`ranhigh ()`
→ float

Returns a value between 0 and 1 with higher values more likely.

`ranmiddle ()`
→ float

Returns a value between 0 and 1 with middle values more likely.

`rangauss (&opt a = 1, b = 0)`
→ float

Returns unbounded value from the normal distribution with standard deviation *a* (sigma) and mean *b* (mu).

`ranexp (&opt lambda = 1)`
→ float

Returns a value greater than 0 from the exponential distribution with stretching factor *lambda*.

`ranbeta (&opt a = .5, b = .5)`
→ float

Returns value between 0 and 1 from the beta distribution.

`rangamma (&opt k = 1)`
→ float

Returns value greater than 0 from the gamma distribution.

`rancauchy ()`
→ float

Returns an unbounded value from the Cauchy distribution.

`ranpoisson (&opt lambda = 1)`
→ float

Returns an unbounded integer value from the Poisson distribution with shape factor *lambda*.

Seeding

`random-seed (&opt seed = #f)`
→ list

Returns or sets the current seed used by the random number generator. If *seed* is specified it should be a single integer or an actual *seed list* that was returned by this function. Reseeding

the generator with a saved seed list will regenerate the identical series of random numbers that the seed originally produced.

Patterns

Patterns are objects that generate data in different orderings. Data can be constants (numbers, symbols, lists, etc), sub-patterns, or thunks (functions of zero arguments). Use `next` to return the next value from a pattern; constant data are simply returned from the pattern, sub-patterns produce local orderings within the larger pattern and thunks are called to produce data.

Pattern	Description
Cycle	generates its elements in a continuous loop
Line	generates its elements sequentially and sticks on last element
Heap	generates random permutations of its elements
Palindrome	generates its elements forwards and backwards
Rotation	generates systematic permutations of its elements
Weighting	generates elements from a weighted random distribution
Markov	generates nth order Markov chains of its elements
Graph	generates elements from a directed graph
Repeater	repeats data generated by a pattern

make-cycle (*items* &optkey *for* = *#f*, *limit* = *#f*)
→ pattern

Returns a pattern that generates its elements in a continual loop. *Items* is the list of data to generate and can include sub-patterns. *For* sets the number of internal reads that will make up the next period, or chunk, of data returned from the pattern. This value can be a number or pattern of numbers and defaults to the length of *items*. *Limit* sets an optional limit on the number of periods that the pattern will generate; once this limit has been reached the pattern returns an end of data marker instead of data.

make-line (*items* &optkey *for* = *#f*, *limit* = *#f*)
→ pattern

Returns a pattern that generates *items* sequentially and sticks on the last item.

make-heap (*items* &optkey *for* = *#f*, *limit* = *#f*)
→ pattern

Generates random permutations of its *items* (sampling without replacement).

make-palindrome (*items* &optkey *for* = *#f*, *limit* = *#f*, *elide* = *#f*)
→ pattern

Returns a pattern that generates *items* forwards and backwards. *Elide* determines what happens to the first and last items when the pattern reverses itself. If the value is false then these items are repeated. If the value is true then neither item is directly repeated. Otherwise the value can should be a two element list {*bool bool*} specifying the repeat value for the first and last item.

make-weighting (*items* &optkey *for* = *#f*, *limit* = *#f*)
→ pattern

Returns a weighting pattern that generates *items* from a weighted distribution. Each item can be an item, sub-pattern, or list:

{*item* [*weight 1*] [*min 1*] [*max #f*]}

where *item* is the thing to generate, *weight* is its weight in the distribution relative to the other items in the pattern, *min* is the minimum number of times that *item* must be directly selected before another item can be considered, and *max* is the maximum number of times that *item* can be directly selected before another item must be selected.

make-rotation (*items* &optkey *for* = *#f*, *limit* = *#f*, *rotate* = *{0}*)
→ pattern

Returns a pattern that rotates its items according to *rotate*, which can be a list or a pattern of lists of the format:

{*start* [*step 1*] [*width 1*] [*end*]}

make-markov (*items* &optkey *for* = *#f*, *limit* = *#f*, *past* = *{}*)
→ pattern

Returns a pattern that generates its items according to (nth order) transitions lists:

{*past ...* -> {*next weight*} ...}

Items appearing before -> in the rule are the past choice(s) that trigger the transition and items following -> make up the weighted distribution determining the transition's next outcome. If *past* it supplied it will be the initial past choices to match the transitions against, otherwise the first transition rule is used.

markov-analyze (*list* &optkey *order* = *1*, *mode* = *1*)
→ void, pattern or list

Analyzes a *list* of data for a specified Markov *order* (0-n) and computes a table capturing the Markov transitions rules that generates a version *data* for the supplied *order*. Returns results according to *mode*. If *mode* is 1 then the transition table is printed in an easy to read row and column format. If mode is 2 then a markov pattern containing the table is returned. If *mode* is 3 then the table is returned as a list of items suitable for passing to make-markov.

make-graph (*items* &optkey *for* = *#f*, *limit* = *#f*)
→ pattern

Returns a pattern that generates its items from a directed graph of nodes with transition patterns. Each node is a list:

{*item to* [*id*]}

where *item* is the item or pattern to generate, *to* is an id or pattern of ids identifying which node should follow the current node in the graph, and *id* is a unique identifier for the node which defaults to the node's position in *items* starting from 1.

make-repeater (*pattern* &optkey *for* = *1*, *repeat* = *#f*, *limit* = *#f*)
→ pattern

Returns a specialized pattern that repeats periods of data generated from a specified pattern. *For* specifies how may elements of data will constitute one period of elements returned by the repeater. The total number of elements in the repeater's period will be equal to this value times the number of elements returned by the pattern. Alternately, *repeat* specifies how many times each period will be repeated before a new period is generated. The *repeat* initialization causes the *for* initialization to be ignored.

next (*pattern* &opt *chunk* = *#f*)
→ element or list

Returns one or more elements from *pattern* according to *chunk*. If *chunk* is false then the next element from *pattern* is returned. If *chunk* is a number then that many elements are read from

the pattern and returned in a list. Otherwise if *chunk* is true then the next period's worth of data is returned in a list.

`eop? (pattern)`
→ boolean

Returns true if *pattern* is at the end of its current period. *Pattern* can be a pattern or the value returned from the pattern by next.

`eod? (pattern)`
→ boolean

Returns true if *pattern* is at the end of its data. This condition can only happen if the pattern was created with a *repeat* value explicitly provided. *Pattern* can be a pattern or the value returned from the pattern by next.

`promise (expr)`
→ function

Turns *expr* into a function of zero arguments that, when called, returns the value of *expr*. Use promise to embed expressions in pattern data, when a promise is read by `next` the value it returns becomes the value of next.

Spectral Composition

The following functions create, access or modify *spectrum* objects. A spectrum holds a sequence of frequencies, an (optional) sequence of amplitudes and an (optional) time value. A spectrum can be sent to and from plotter displays for graphic editing and converted to other representations, for example to lists of notes and key numbers.

Creating Spectra

`fm-spectrum (carrier, ratio, index)`
→ spectrum

Returns a spectrum generated by Frequency Modulation given a *carrier* in hertz, a modulation *ratio* and an fm *index*.

`rm-spectrum (freq1, freq2)`
→ spectrum

Returns a spectrum generated by ring modulation, where *freq1* and *freq2* can be spectrum objects, lists of hertz values or single hertz values.

`sdif-import (file &opt type = #f )`
→ list

Imports SDIF *file* as a list of frame data. If *type* is not specified all frames in the file will be imported, otherwise *type* should be a valid SDIF frame type (a four character string, symbol or keyword, e.g. "1TCR") in which case only frames of that type will be imported. The data returned is a list of frames, where each frame contains an SDIF type, time and one or more matrixes:

```
(type time matrix ...)
```

and each matrix in a frame is a list containing a matix type and one or more rows of data:

```
(type row ...)
```

and each row in a matrix is a list of one or more values:

```
(val ...)
```

sdif-import-spectra (*file*)
→ list

Imports SDIF "1TCR" frame data in *file* as a list of spectrum objects.

spear-import-spectra (*file*)
→ list

Imports SPEAR frame data in *file* as a list of spectrum objects.

spear-export-spectra (*spectra*, *file*)
→ list

Exports on or more spectrum objects in *spectra* to a file suitable for importing into SPEAR. Since spectra do not have partials.

spectrum-copy (*spectrum*)
→ spectrum

Returns a copy of *spectrum*.

Accessing Spectral Information

spectrum-size (*spectrum*)
→ integer

Returns the number of frequency and amplitude pairs in *spectrum*.

spectrum-time (*spectrum*)
→ float or #f

Returns the time of *spectrum* or false if there is none.

spectrum-maxfreq (*spectrum*)
→ float

Returns the highest frequency in *spectrum*.

spectrum-minfreq (*spectrum*)
→ float

Returns the lowest frequency in *spectrum*.

spectrum-maxamp (*spectrum*)
→ float

Returns the highest amplitude in *spectrum* or 0.0 if there are no amplitudes.

spectrum-minamp (*spectrum*)
→ float

Returns the lowest amplitude in *spectrum* or 0.0 if there are no amplitudes.

spectrum-freqs (*spectrum*)
→ list

Returns the frequency values of *spectrum*.

spectrum-amps (*spectrum*)
→ list

Returns the amplitude values of *spectrum*.

spectrum-pairs (*spectrum*)
→ list

Returns a list containing the frequencies and amplitudes of *spectrum* formatted as pairs: (f_1 a_1 ... f_n a_n).

Modifying and Transforming Spectral Information

spectrum-keys (*spectrum* &optkey *order = 1*, *thresh = 0.0*, *quant = #f*, *unique = #f*, *min = #f*, *max = #f*)
→ list

Returns a list of the frequency components of *spectrum* converted to key numbers. *Order* specifies the scale order the keys will be returned in. *Threshold* specifies the minimum amplitude that frequencies must possess in order to be returned as keys. If *threshold* is a list (*min max*) then components whose amplitudes are greater than max are also suppressed. If *quant* is a number between 0.0 and 1.0 then key numbers are quantized to that fractional cent value, e.g. *quant* .5 returns key numbers quantized to the nearest quarter-tone and *quant* 1 returns key numbers rounded to the nearest integer. If *unique* is true then no duplicate key numbers will be returned. If a *min* key number is specified then spectral values lower than that will be octave shifted upward until they equal or exceed it. If a *max* key number is specified then spectral components above it will be octave shifted downward until they are equal to or lower than it.

spectrum-add! (*spectrum*, *freq*, *amp*)
→ spectrum

Adds a pair of *freq* and *amp* values into *spectrum*. If *freq* already exists in *spectrum* then *amp* is added to its current amplitude.

spectrum-flip! (*spectrum*)
→ spectrum

Inverts the frequencies in *spectrum* while preserving the minimum and maximum frequency boundaries.

spectrum-invert! (*spectrum* &optkey *point* = #f)
→ spectrum

Inverts frequencies in *spectrum* around *point*, or around the first partial if *point* is false.

spectrum-rescale! (*spectrum* &optkey *min* = #t, *max* = #t)
→ spectrum

Rescales frequencies in *spectrum* to lie between the new *min* and *max* values. If #t is specified it means that the current min or max value is preserved in the rescaling.

Cellular Automata

make-automata (*cells*, *transfunc* &optkey *title = #f*, *colormap = {}*, *backgroundcolor = "white"*, *cellsize = 50*, *cellbordersize = 1*, *rows = 1*)
→ automata

Creates a cellular automata containing the specified initial integer *cellstates* and using the transition function *transfunc* to produce all subsequent states for cells in the automata. For 2D automata *cellstates* is a list of lists where each sublist defines a row of initial states.

The transition function *transfunc* is called automatically and passed two arguments, the automata and the index of the current cell, and must return the next state of the cell at index. Inside the transition function you can use the function `cell-state` to access the current state of any cell in the automata.

To associate a graphical window with an automata provide the optional *window* and *colormap* arguments, where *window* is the title (string) of the new window and *colormap* is a list of state and color pairs: (*state1 color1 state2 color2 ... stateN colorN*). Note that an error is signaled if *window* is already open, and that the color map must contain all possible states of the automata along with their colors in the window. Colors are symbols or strings, the list of all colors is available in the globals `*colors*` variable. The *backgroundcolor* argument is a string or symbol naming the background color for the window. The *cellsize* argument is the size in pixels of each cell in the state window display. The *cellbordersize* argument is the size in pixels of space between each cell in the state window. The *rows* argument applies to 1-dimensional automata only, and specifies the number of successive generations to display in the window, where each generation is shown as a cell "row".

Once an automata has been created use `state` to read successive states from it.

See the HELP>EXAMPLES>CELLULAR AUTOMATA menu for examples of using cellular automata.

state (*automata* &optkey *all* = #f)
→ integer or list

Return the next integer state from *automata* or a list of all the states in the current generation if *all* is true.

cell-state (*automata*, *index* &optkey *neighbor* = 0)
→ integer

Returns the current integer state of the *automata* cell at *index*. If neighboring cells to *index* must be accessed provide a positive or negative *neighbor* increment to *index* . Cell addresses are always accessed mod the size of the cell array so you do not have to perform bounds checking when you specify a neighbor increment. For 2D automata each *index* and *neighbor* is a two dimensional `xy` point. You specify 2D points and 2D increments using the `xy` function, i.e. `(cell-state index (xy -1 -1))` would access the value of the north-west neighbor of the cell at index in a 2D automata.

xy (*x*, *y*)
→ integer

Encodes *x* and *y* values -+2768 as a single point (integer) value.

Environment

print (*expr* {, *expr*}*)
→ void

Prints one or more expressions to the Console window.

now ()
→ float

Returns a current time stamp. Note that now values are not used by the scheduler or in processes, which have their own notion of `elapsed` time.

mouse-button (&optkey *upval* = #f, *downval* = #t)

→ object

Returns *downval* if the mouse button is currently depressed, otherwise returns *upval*.

mouse-x (&optkey *minval = 0.0*, *maxval = 1.0*, *warp = 1.0*)
→ float

Returns the screen mouse x position scaled *minval* to *maxval*, where *minval* is the value returned when the mouse is full-left on the screen and *maxval* is the value at full-right. If *warp* is 1 the interpolation is linear, otherwise values greater than 1 produce increasing exponential spread towards the right and values less than 1 produce decreasing spread.

mouse-y (&optkey *minval = 0.0*, *maxval = 1.0*, *warp = 1.0*)
→ float

Returns the screen mouse y position scaled *minval* to *maxval*, where *minval* is the value returned when the mouse at the top of the screen and *maxval* is the value at the bottom. If *warp* is 1 the interpolation is linear, otherwise values greater than 1 produce increasing exponential spread towards the bottom and values less than 1 produce decreasing spread.

shell (*command*)
→ void

Calls operating system's shell interpreter to execute user's *command*.

load (*file*)
→ void

Loads a Lisp or SAL *file*. If *file* has no directory component and the current working directory does not contain *file* then the embedded distribution will be searched for the *file* to load.

chdir (*directory*)
→ void

Changes the current working directory to *directory* (string).

pwd ()
→ string

Returns the full pathname string of the current working directory.

directory (*spec* &opt *recurse = #f*)
→ list

Returns a list of absolute filenames matching wildcard *spec*. If *recurse* is true then subdirectories are recursively searched.

file-version (*filename* &optkey *version = #t*, *nooverwrite = #f*)
→ string

Returns a variant of *filename* with a version number appended to its name. If *vers* is #t then successive version numbers are appended (starting with 1), if *vers* is a number then that number is used as the version number, else if *vers* is false then no version number is appended to the name. If *nooverwrite* is true the versioned file name is guaranteed to be for a new file, i.e. if that name is used for output no existing file will be overwritten.

Iteration

An (essentially) complete implementation of the Common Lisp loop macro is provided in Scheme and SAL.

```
loop [with-decl] {stepping}* {stopping}* {action}+ [finally] end
(loop {clause}*)
```

→ loop dependent value

Defines an iteration. Optional local variables can be declared using the `with` statement followed by zero or more *stepping* statements, zero or more *stopping* statements, one or more *action* statements, an optional *finally* statement and terminated with the required `end` tag.

stepping statements	
`repeat` *expr*	sets the iteration limit to *expr*
`for` *var* = *expr* [`then` *expr2*]	sets *var* to the value of *expr* on each iteration. If `then` is specified *expr* is the initial value and *expr2* are the subsequent values
`for` *var* `in` *expr*	increments *var* over elements the list *expr*
`for` *var* [`from` *expr*] [`to` | `below` *expr*] [`by` *expr*]	increments *var* from an on optional starting value to an ending value by an optional amount
stopping statements	
`while` *expr*	iteration continues while *expr* is true
`until` *expr*	iteration stops when *expr* is true
conditional statements	
`when` *expr*	action executed if *expr* is true
`until` *expr*	action executed if *expr* is false
action statements	
`do` *expr...*	scheme expressions
any SAL statement	SAL statements execute but do not return values
finally statements	
`finally` *statement*	executes *statement* as soon as iteration has completed

Audio

Depending on the which particular Audio Target you are working with, audio data can be generated in four different ways:

1. sending data to an open audio port in real time.
2. sending data to an open audio file (non-real time).
3. sprouting processes to send audio to an open audio port in real time.
4. sprouting processes to send audio to an open audio file (non-real time).

Here are small examples of each method:

```
; example 1: sending data to the open Midi Port in real time

loop for i below 10
  mp:midi(i * .1, key: between(60,90))
end

; example 2: sending data to a Midi File (non-real time)

file "test.mid" ()
  loop for i below 10
    mp:midi(i * .1, key: between(60,90))
  end
```

```
end

; example 3: sprouting process to send data to the open Midi Port in real time

process simp(num, low, high)
  repeat num
  mp:midi(key: between(60,90))
  wait .1
end

sprout(simp(10, 60, 90)

; example 4: sprouting process to send data to a Midi File (non-real time)

sprout(simp(10,60, 90), "test.mid")
```

Which method you use depends on what you want to accomplish and which methods the Audio Port in question supports. Here is a table that shows the capabilities of each Audio Target:

audio target	real time port?	score file? (non-real time)
Midi	yes	yes
Fomus	no	yes
CLM	no	yes
Csound	no	yes
OSC	yes	no

Midi Out

Before sending anything to the Midi Out port select use the AUDIO>MIDI OUT> menu to select an output device. You can test that the connection actually works by using the PORTS>MIDI OUT>TEST to send a random burst of messages to the output device.

mp:midi (&optkey *time = 0*, *dur = .5*, *key = 60*, *amp = .5*, *chan = 0*)
→ void

Send note data to midi port. Note: if a process using mp:midi is sprouted in the context of a Fomus score then mp:midi will automatically route its note date to the Fomus score, with the midi channel number used as the Fomus part id.

mp:on (&optkey *time = 0*, *key = 60*, *amp = .5*, *chan = 0*)
→ void

Send note on data to midi port. Supports floating point keynums and tunings

mp:off (&optkey *time = 0*, *key = 60*, *vel = 127*, *chan = 0*)
→ void

Sends note off data to midi port.

mp:prog (&optkey *time = 0*, *val = 0*, *chan = 0*)
→ void

Sends program change value to midi port.

mp:ctrl (&optkey *time = 0*, *num = 0*, *val = 0*, *chan = 0*)
→ void

Sends control change data to midi port.

mp:touch (&optkey *time = 0*, *val = 0*, *chan = 0*)
→ void

Sends after touch value to midi port.

mp:press (&optkey *time = 0*, *val = 0*, *chan = 0*)
→ void

Sends channel pressure value to midi port.

mp:bend (&optkey *time = 0*, *val = 8192*, *chan = 0*)
→ void

Sends pitch bend value 0-16383 to midi port.

mp:instruments (*prog ...*)
→ void

Sends up to sixteen program changes out the Midi port and updates the port's instrument table. The position of each value specifies the channel it is sent to, i.e. the first program change is sent to channel 0 and so on. To skip a particular channel provide #f as its program change value, otherwise the value should be either a program change (integer 0-127) or a GM instrument name (string), in which case the first instrument that matches or contains name as a substring will be selected.

mp:tuning (*divisions*)
→ void

Divides midi frequency space into *divisions* steps per semitone and automatically quantizes floating point key numbers in midi data to the nearest tuning step as they are sent out the port. A *division* of 1 implements standard semitonal tuning, 2 implements quarter-tone tuning, and so on. The maximum *divisions* of 16 per semitone provides a tuning step size of 6.25 cents, very close to the minimum perceivable frequency change (see table below.) For example, in quartertone tuning there are a total of 8 midi channels available (channel numbers 0 to 7, each with their own instrument assignment) and fractional keynum values *k*.0 < *k*.5 (e.g. 60.0 < 60.5) are quantized to the semitone and *k*.5 < *K*.0 (e.g. 60.5 < 61.0) are tuned to the quartertone. In some configurations it is possible to send both standard tuning in parallel with microtonal tuning by using any remaining "standard" channels numbers for semitonal output. Channel tuning is very efficient because it claims certain channels and "pretunes" them for the appropirate divisions.

divisions	tuning step	microtuned channel #	standard channel #
1	semi tone	—	0-15
2	quarter tone (50 cents)	0-7	—
3	third tone (33 cents)	0-4	15
4	fourth tone (25 cents)	0-3	—
5	fifth tone (20 cents)	0-2	15
6	sixth tone (16.6 cents)	0-1	12-15
7	seventh tone (14.28 cents)	0-1	14-15
8	eighth tone (12.5 cents)	0-1	—
9	ninth tone (11.1 cents)	0	9-15
10	tenth tone (10 cents)	0	10-15
11	eleventh tone (9.09 cents)	0	11-15
12	twelfth tone (8.33 cents)	0	12-15
13	thirteenth tone (7.69 cents)	0	13-15
14	fourteenth tone (7.14 cents)	0	14-15

15	fifteenth tone (6.66 cents)	0	15
16	sixteenth tone (6.25 cents)	0	—

Midi In

mp:receive (*function*)
mp:receive (*opcode*, *function*)
→ void

Causes *function* to receive incoming MIDI messages. To clear an existing receiver specify #f as the *function*. Otherwise *function* should be a procedure of one argument; this procedure will be called with a list of message values (*opcode channel data1 data2*) for each message that arrives at the MIDI input port.

To associate the receiver with a specific type of message precede *function* by the MIDI *opcode* 8-15 that it should handle. MIDI opcodes 8-15 are also available as constants: mm:off, mm:on, mm:touch, mm:ctrl, mm:prog, mm:press, mm:bend. If no opcode is given then *function* becomes the "default" receiver and is called on any message that is not otherwise handled by an opcode receiver. Note that it possible to have several receivers, including a default receiver, in effect at the same time, each handling a different type of message.

To clear all current receivers call mp:receive with no arguments.

mp:receive? (&opt *opcode = #f*)
→ boolean or list

Returns true if a receiver for *opcode* is currently in effect otherwise false. If called without any *opcode* then a list containing the opcodes of all current receivers is returned. The default receiver, if any, is indicated by an opcode of 0.

mp:inchans (*chan ...*)
→ void

Enables MIDI input only on the specified channels. Channel values are integers between 0 and 15. Specify a single #t value to enable all channels and #f to disable all channels.

mp:inops (*opcode ...*)
→ void

Enables MIDI input only for the specified MIDI opcodes. Opcodes are integers between #x8 and #xE (mm:off mm:on mm:touch mm:ctrl mm:prog mm:press mm:bend). Specify a single #t to enable all opcode types and #f to disable all opcodes.

midifile-header (*file*)
→ list

Returns a list of MIDI header information from the specified *file*. The format of this list is (*tracks format tempo timesig maxtime*) where *tracks* is the number of tracks in the file, *format* is the time format of the file (if positive its the number of ticks per quarter else its SMPTE), *tempo* is the initial tempo or #f , and *maxtime* the maximum time stamp found in any track.

midifile-import (*file*, *track*, *values*)
→ list

Returns a list of MIDI *values* from the specified *track* (zero based) in *file*. Messages that match (contain) the specified values are processed, messages that do not match do not appear in the returned list. *Values* can be a single value, a single list of values or a list of lists of values. Each individual value can be a string, symbol, or keyword as defined below. To return more that one value from messages specify a single list of values. In this case lists of data are returned, where each data list is a match for the specfied values. If a list of lists of values is

specified then any message that matches any of those value lists are collected as described above.

Value	Matching Messages	Description
"op"	all	the opcode of the message. Channel message opcodes are: note = 9, aftertouch = 10, controller = 11, program = 12, pressure = 13, pitchbend = 14. Meta message opcodes are: seqnum = 0, text = 1, copyright = 2, trackname = 3, instname = 4, lyrics = 5, marker = 6, cue = 7, chanprefix = 32, tempo = 81, timesignature = 88, keysignature = 89.
"time"	all	the start time of the message in seconds
"delta"	all	the time increment from the previous message
"chan"	channel messages	the channel of the message 0-15
"rhythm"	notes	the time increment from the previous note
"dur"	notes	the duration of the note
"key"	notes	the key number of the note
"amp"	notes	the amplitude of the note 0.0-1.0
"vel"	notes	the key velocity the note 0-127
"touch"	after touches	the after touch value 0-127
"ctrl1"	control changes	the controller number 0-127
"ctrl2"	control changes	the controller value 0-127
"prog"	program changes	the program value 0-127
"press"	channel pressures	the pressure value 0-16383
"bend"	pitch bends	the pitch bend value 0-16383
"seqnum"	seqnum meta events	the sequence number
"text"	text meta events (opcodes 1-7)	the text string of the message
"chanpre"	channel meta events	the channel prefix number
"tempo"	tempo meta events	the tempo as quarters per second
"timesig"	time signatures	a time signature list (*numerator denominator*)
"keysig"	key signatures	a key signature -7 to 7

Example

```
;; find out what kind of messages are in track 0
midifile-import("zincline-1.mid", 0, "op")

;; import the start times, key numbers and durations of all notes in track 1
midifile-import("zincline-1.mid", 1, {"time" "key" "dur"})

;; import all tempo changes and program changes from track 0
midifile-import("zincline-1.mid", 0, {{"op" "tempo"} {"op" "prog" "chan"}})
```

CLM/SndLib

Use File>Load... or Audio>Instrument Browser... to load CLM instruments before working with them.

SndLib Audio File Options

Any CLM with-sound option can be passed as an argument to sprout when you generate an audio file. The most common options are listed here and are also available in the Audio>SndLib> menu.

`srate:`
: The sampling rate to use.

`channels:`
: The number of audio channels to write.

`play:`
: Play the audio file after writing it.

Fomus

Fomus generates musical scores for LilyPond, Sibelius and Finale from symbolic score information input by the user interactively or by running musical processes. Use the Audio>Fomus> menu to configure your Fomus environment and manage its score output. For more information see the examples in Help>Examples>Fomus Examples and the tutorial in Help>Tutorials>Fomus.

Fomus Events

fms:note (*time*, *dur*, *pitch* &optkey *part = #f*, *voice = 1*, *dyn = 0*, *grace*, *marks = {}*, *sets = {}*)

→ void

Send a note event to the currently selected score at *time*. Parameters *dur* and *pitch* are required. The optional parameters *marks* and *sets* are lists of marks and settings.

fms:rest (*time*, *dur* &optkey *part = #f*, *voice = 1*, *grace*, *marks = {}*, *sets = {}*)

→ void

Send a rest event to the currently selected score at *time*.

fms:mark (*time*, *dur* &optkey *part = #f*, *voice = 1*, *grace* , *marks = {}*, *sets = {}*)

→ void

Send a mark event to the currently selected score at *time*.

fms:meas (&optkey *time = 0*, *dur = 0*, *sets = {}*)

→ void

Send a measure to the currently selected to score at *time*.

Fomus Objects

fms:part (&optkey *sets = {}*)

→ void

Create a new part to use in the currently selected score. A part with id `default` exists (with instrument `default`) if no other parts are defined.

fms:inst (&optkey *sets = {}*)

→ void

Create a new instrument to use in the currently selected score. An instrument with id `default` and a grand staff exists if no other instruments are defined.

fms:percinst (&optkey *sets = {}*)
→ void

Create a new percussion instrument to use in the currently selected score.

fms:metapart (&optkey *sets = {}*)
→ void

Create a new metapart to use in the currently selected score.

fms:setting (*set*, *value* &optkey *append = #f*)
→ void

Change the value of one or more global settings in the currently selected score.

fms:measdef (&optkey *sets = {}*)
→ void

Create a new measure definition to use in the currently selected score. Measdefs are more useful in .fms files than in processes.

Fomus Commands

fms:new-score (*file*)
→ void

Create a brand new empty score with output *file*.

fms:load-score (*file*)
→ void

Load a .fms file into a new score.

fms:select-score (*file*)
→ void

Select the first score with output *file*. A new empty score is created if one with that name does not already exist.

fms:save-score (*file*)
→ void

Save the score to a .fms file, Fomus's native file format. The file can be edited and/or reloaded with the load-score command.

fms:merge-score (*fromfile* &optkey *time = 0*)
→ void

Merge the contents of the score identified by *fromfile* into the current score. Contents include note events, measures and possibly parts if they don't already exist. The optional *time* argument shifts the events in time by the specified amount.

fms:run ()
→ void

Run Fomus on the currently selected score. If an output file has not been specified, this produces an error.

fms:clear-score ()

→ void

Clears note events and measures (but not part or instrument definitions) from the currently selected score.

fms:delete-score ()

→ void

Delete the currently selected Fomus score.

OSC (Open Sound Control)

The OSC port provides input and output communication with other processes via the Open Sound Control protocol. To send and receive OSC messages first open an OSC connection via AUDIO>OSC CONNECTIONS... dialog or by calling the "osc:open" function.

See the HELP>EXAMPLES>OSC menu for examples of using OSC.

osc:open (*serverport*, *targetport*)

→ void

Opens an OSC connection with *serverport* as the CM application's input port and *targetport* as its output port. If the target is not on the local machine then *targetport* can be a string "*host*:*port*", where *host* is the host's network name or IP address and *port* is the port to route data to on that machine.

osc:close ()

→ void

Closes an open OSC connection.

osc:message (*path*, *data ...*)
osc:message (*list*)

→ void

Sends an OSC message consisting of an OSC *path* (string) followed by zero or more optionally tagged *data* values. Message values can be specified as separate arguments or in a single *list* argument to the function. Scheme values integer, float, string, symbol, character, boolean true, boolean false all map directly to their OSC equivalents and do not need tagging. To distinguish between different OSC types for a particular scheme value (e.g. 32 bit vs 64 bit integers), precede the Scheme value by the appropriate OSC data tag as shown in the table below. Lists holding OSC blobs or OSC four-byte midi messages must be tagged as such using `:b` or `:m`, respectively. The special OSC tags `:T :F :N :I` are themselves data, i.e. they not take Scheme values.

OSC Tag	Osc Type
`:i`	32 bit integer
`:h`	64 bit integer
`:f`	32 bit float
`:d`	64 bit float (double)
`:c`	character
`:s`	string
`:S`	symbol
`:b`	blob
`:m`	midi
`:T`	OSC true

:F	OSC false
:N	OSC nil
:I	OSC infinitude

osc:bundle (*time*, *message ...*)
osc:bundle (*time*, *list*)
→ void

Sends an OSC bundle consisting of a *time* tag (number) followed by one or more OSC *messages* . To send an "immediate" bundle, specify 0 as the time tag. Otherwise the relative time value will be converted to an OSC time value based on January 1 1900. Messages can be specified as separate arguments or in a single *list* argument to the function.

osc:receive (*function*)
osc:receive (*path*, *function*)
→ void

Causes *function* to receive incoming OSC messages. To clear an existing receive specify #f as the *function*. Otherwise *function* should be a procedure of one argument; this procedure will be called asynchronously and passed a list of message values (*data1 data2 ...*) for each message that arrives at the OSC input port. To associate receiving with a specific osc message precede *function* by the OSC *path* (string) that it should handle. If no path is given then *function* becomes the "default" receiver and is called on any message that is not otherwise handled by a specific path receiver. Message data for a default receiver will include the path of the message received: (*path data1 data2 ...*). Note that it possible to have several receivers, including a default receiver, in effect at the same time, each handling a different type of message.

To clear all current receivers call osc:receive with no arguments.

osc:receive? (&opt *path*)
→ boolean or list

Returns true if a receiver for *path* is currently in effect otherwise false. If called without any *path* then a list containing the paths of all current receivers is returned. The default receiver, if any, is indicated by a path of "/".

Csound

Csound score files can be written and played in all versions of CM. Use the Audio>Csound>Settings... dialog to configure the Csound environment, or pass the options directly to the scorefile when you generate it with sprout.

cs:i (*ins*, *start*, *dur ...*)
→ void

Sends instrument data to Csound. The first value should be the instrument's number followed by the rest of its pfield values.

cs:f (*num ...*)
→ void

Sends function table data to Csound. Similar to "cs:i" in all other respects.

cs:event (*list*)
→ void

Sends an *event list* to Csound. The first element in the list must be the event's type, either `i` or `f`. Similar to "cs:i" and "cs:f" in all other respects.

Csound Scorefile Options

The following scorefile options are available in the AUDIO>CSOUND> menu and can be passed to Csound scores when you generate them using sprout or file.

`play:`
: Send the scorefile to Csound after it is written.

`orchestra:`
: Use this file as the Csound .orc file when the scorefile is played.

`header:`
: Include this string as the header of the scorefile.

`write:`
: Specify #f to put data in an internal score without writing it as a file. This lets you build a score incrementally, over consecutive sprouts, and to export score data to an editor window or the clipboard.

The Scheduler

The scheduler is a JUCE thread that executes (runs) musical processes in real-time or in faster-than-real-time if files are being generated. The scheduler uses metronomes to manage the rate of execution and is controlled using functions that sprout, stop, pause and continue process execution.

Processes

Processes are iterative functions that you `sprout` into the scheduler to execute. In SAL you create a process using the process statement. In Scheme you create a process using the process macro. The following two examples are equivalent process definitions:

SAL process definition

```
process simp (reps, rate, low, hi)
  repeat reps
      mp:midi(key: between(low, hi))
      wait rate
  end

sprout(simp(20, .5, 60, 80))
```

Scheme process definition

```
(define (simp reps rate low hi)
  (process repeat reps
           do
           (mp:midi :key (between low hi))
           (wait rate)))

(sprout (simp 20 .5 60 80))
```

process *name* ({*parameters*}*) {*iteration*}+ **end**
→ function

Returns an iterative function to execute in the scheduler in real time or faster-than real time. *Parameters* consists of zero or more required parameters, optionally followed by `&optkey` and any optkey arguments. Following the parameter list comes a series of iteration clauses as documented under loop and with the following two additional features:

```
wait time
```
→ void

Causes the process to wait *time* seconds before the next iteration. To stop the process provide negative *time* value. This function only exists within the lexical scope of a process definition.

Expressions defined within the body of a process have access to `elapsed`, a (load) function that returns the time of the current process iteration:

```
elapsed (&opt scoretime? = #f)
```
→ seconds

Returns the amount of time since the process was first sprouted, or the absolute score time (the time since the initial sprout) if the optional *scoretime* argument is `#t` This function only exists within the lexical scope of a process definition.

Sprouting Processes

The main entry point into the scheduler is called sprout. Sprout schedules processes to run in real time, or in faster-than real time if a file (string) is specified as the third argument to the function.

When the scheduler runs in real time processes can be manipulated interactively using metronomes to control execution rates (tempi) and process *ids* (see below) to stop or replace individual process currently running in the scheduler.

When the scheduler runs faster-than real time for file generation then time values inside processes are treated as if they were seconds but the scheduling clock is running as fast as it can, i.e. the scheduling thread advances process times but does not actually wait any amount of time before considering the next process in the queue.

Sprouting Processes in Real Time

```
sprout (process &optkey ahead = 0, id = 0)
```
→ void

Starts one or more processes running in real-time. Each *process* begins executing either *ahead* seconds from now or ahead beats from now if sync is used in conjuction with a metronome. Both *process* and *ahead* can be lists, in which case processes and start time are paired in a left-to-right order.

Each process is identified by an *id* in the scheduler. If no id is provided then the newly sprouted process is tagged with the special id 0. If an *id* is provided it should be a number or string that uniquely identifies the process in the scheduling queue. Uniquely tagged processes can then be selectively removed using stop or *replaced* by sprouting a new process using the same id. In this case the running process is stopped and substituted by the new process. The new process will start in time, completing the wait time already underway. This allows you to "redefine" running processes on the fly, as in live coding.

Sprouting Processes to Generate Files

```
sprout (process, file ...)
sprout (process, ahead, file ...)
```
→ void

If a *file* is specified then the scheduler will run in faster-than real time and all output will be sent to the open file. CM currently supports the following file types:

- `".mid"` midi file.
- `".wav .aiff .snd"` audio file using SndLib.
- `".ly .xml .fms"` notated score using Fomus.
- `".sco"` Csound score file.

Following *file* comes zero or more keyword options pairs suitable for the audio target selected by *file*:

- Midi file options
- SndLib audio file options
- Fomus score options
- Csound score options

Other Scheduling Functions

`stop (id...)`
→ void

Stops processes currently running in the scheduling queue. If no ids are given all the processes are stopped and any pending events in the midi output queue are also flushed. Otherwise only processes with the specified ids are stopped. Note that multiple processes can only share the default id (0), in which case all processes sprouted with the default id can be stopped if *id* = 0. Otherwise, processes have unique id numbers, and can be stopped by indicating their id.

Note that *id* can be a number or a string. If a string is given, it is converted into a unique id number. If a process was sprouted with a string id, pass the same string into *stop*.

`pause ()`
→ void

Suspends the process scheduler.

`cont ()`
→ void

Resumes the scheduler if it is currently paused.

Metronomes

Metronomes allow you to change the tempo of running processes in real-time. At least one metronome is always active, if you do not specify a metronome the 'default' metronome will be in effect. Each metronome has a unique identifier that you use to access the metronome from code; the id of the default metronome is 0.

`make-metro (tempo)`
→ id

Creates a new metronome with an initial *tempo* in beats per minute. Returns the unique id of the new metronome.

`metro? (metro)`
→ bool

Returns true if *metro* is the valid id of a metronome.

`delete-metro (metro)`

→ void

Deletes the metronome with the id *metro*. The default metronome cannot be deleted.

metros (&optkey *user = #f*)
→ list

Returns a list of all existing metronome ids. If *user* is false (the default) then the list returned will include the default metronome at the front of the list. If *user* is #t, then the list will omit the default metronome and include only metronome that were created by the user.

metro (*tempo*, &optkey *secs = 0*, *metro = 0*)
→ void

Changes the tempo of metronome *metro* over time *secs*. If *secs* = 0, then the tempo change will be effective immediately.

metro-beat (&optkey *metro = 0*)
→ number

Returns the exact beat of metronome *metro*.

metro-tempo (&optkey *metro = 0*)
→ number

Returns the current exact tempo of metronome *metro*.

metro-dur (*beats* &optkey *metro = 0*)
→ number

Returns the equivalent time value for *beats* number of beats according to metronome *metro*. One can use this function to adjust duration and time offset values sent inside of midi messages, as mp:midi expects its duration in absolute seconds.

sync (&optkey *ahead = 1*, *metro = 0*)
→ thunk

Returns a special ahead value (a thunk) that sprout will use to quantize the start of the process to the *ahead* beat (or beat subdivision) of *metro*. For example, if *ahead* is 1, then the process will start on the beginning of the next beat. A value of .5 will start the process at the nearest offbeat. 1.5 will wait until the next beat, then start it on the offbeat of that beat. 2.75 will wait 2 beats, then start on the 3rd sixteenth note of that beat, and so on. Note that the ahead beat can never be zero because that beat is already underway.

metro-sync (*metro* &optkey *beats = 1*, *tempo = 0*, *master-metro = 0*, *mode = 0*)
→ bool

Will adjust metronome *metro* such that it will align a downbeat to coincide with metronome *master-metro*. This alignment will be completed *beats* later according to the *master-metro*. If *tempo* is 0, the tempo of the *metro* will be set back to its former state once the sync is complete. If *tempo* is a positive number, the tempo of *metro* will be changed throughout the length of the syncing. Thus, once the specified number of *beats* has passed, *metro* will be at its new tempo and in sync with *master-metro*. In order to accomplish the syncing, *metro* must either speed up or slow down for a brief period of time. *mode* = -1 will force *metro* to slow down, while *mode* = 1 will force it to speed up. *mode* 0 will choose whichever method is most efficient. Will return true if a tempo adjustment was made. Will return false if the function decided to do nothing. This will happen if either the metronomes are already in sync, or if *metro-sync()* is called before a previous call to the function is complete.

metro-phase (*fitbeats*, *beatspace* &optkey *metro = 0*)
→ bool

Will adjust the tempo of metronome *metro* such that over what would normally be the time of *beatspace*, *fitbeats* number of beats will happen instead. For example, if *fitbeats* = 11, and *beatspace* = 10, the tempo will be altered such that 11 beats will fit within the space of 10 beats. After this is completed, the metronome returns to its previous tempo. Will return true if a phase adjustment was made. Will return false if the function decided to do nothing. This will happen if *metro-phase()* is called before a previous call to the function is complete.

Plotting

Plot windows allow you to display, edit and play back multiple layers of n-tuple point data. See HELP>EXAMPLES>PLOTTING for various examples of creating plots by code.

Plot Window Point Commands:

- Add new point: Control-Mouseclick
- Call a plot hook: Control-Option-Mouseclick
- Select points: drag region or click on point
- Add or remove points from selection: Shift-Click on point
- Delete selection: Delete key

The functions defined below provided an API for working with plots in Scheme or SAL buffers.

plot (*{setting}* {pointlist {setting}*}**)
→ void

Creates a new plot given zero or more global plot settings followed by zero or more *layers* of point lists with layer-specific settings. Point lists can be specified in one of the two following formats:

(*x1 y1 x2 y2 ... xn yn*)
: A flat list of *x y* point data.

((*v1 v2 ... vn*) ...)
: Each sublist defines a *point record* containing a value for every field of plotted data.

A *setting* is a plotting keyword followed by a value. In general, setting values can be specified as symbols, keywords or strings. Settings that appear before the first point list are treated as *global settings*, i.e. they apply to all the point lists. Settings specified after a point list apply only to that layer of points.

plot supports following settings:

title: *string*
: As a global setting is specifies the window title or the title of the point layer if it appears after a point list.

xaxis: *type* | (*type from to by ticks*)
: Specifies the horizontal axis type and (optionally) its characteristics.

Available axis types and their default characteristics

type	from	to	by	ticks
unit	0	1	.25	5
percent	0	100	25	5
keynum	0	127	12	1
midi	0	127	16	2
seconds	0	60	1	4
ordinal	0	length	1	1
circle	-1	1	.25	4

See the fields: setting for complete control over record fields and axes.

`yaxis:` *type* | (*type from to by ticks*)
Specifies the vertical axis type and (optionally) its characteristics. See the fields: setting for complete control over record fields and axes.

`layer:` *points*
A list of points defining a layer in the plot

`color:` *name* | *"rgb"*
The color for drawing the specified plot list.

`style:` *name*
The style for drawing point records.

- `envelope` (points connected by lines)
- `line` (lines only)
- `point` (points only)
- `pianoroll` (horizontal boxes, requires 3 fields of data)
- `impulse` (vertical lines)
- `aerial` (vertical lines with points)
- `bar` (vertical bars)

`fields:` ((*field1 axis initval*)(*field2 axis initval*) ...)
Defines the data fields of the points being displayed in the window. Field lists are requred for points with more than two fields of data and the left to right order of fields corresponds to the left-to-right values in the point records. (The standard 2-dimensional plots do not need to field definitions, they can simply specify global :xaxis and :yaxis for the window.). For each file, *name* is the name of the field *axis* is its display axis definition as described in the xaxis and yaxis settings above and *initval* is the intial value of a field when a new point is created without an explicit value for that field. Fields can "share" the axis of a field to its left. For example in typical pianoroll plot the 3rd field (say, duration) shares the same axis as the first dimensino (seconds). To specify a shared axis use as its axis specificaion the field name of the field it should share with.

plot-data (*title* &opt *all* = *#t*)
→ list

Returns the point data from the plot window whose *title* is specified. If *all* is false then only the points in the topmost (focus) layer will be returned otherwise all layer data will be returned.

plot-hook (*title*, *function*, *userdata...*)
→ list

Associates *function* with Control-Option-Mouseclick inside the plot window named *title*. When a Control-Option-Mouseclick occurs, *function* will be be called automatically and passed the X and Y values of the mouse click along with any additional values specified in *userdata*. The *function* (hook) should return a list of fully specified point records to add to the window, or an empty list if no points should be added.

SAL (version 2)

Sal is a minimalist language for algorithmic music composition. It provides infix expressions and a handful of control statements and reserved symbols:

`begin`	`finally`	`loop`	`until`
`below`	`for`	`process`	`variable`
`by`	`from`	`repeat`	`wait`
`else`	`function`	`then`	`when`
`end`	`if`	`to`	`while`
`file`	`in`	`unless`	`with`

Expressions

Numbers

SAL number representation includes integers, floats and ratios (e.g 1/2, 1/3, 11/17).

Arithmetic and Logical Expressions

Arithmetic and logical expressions are infix. Operators must be delimited from their operands using spaces and/or parentheses (in other words, 2 + 4 is addition and 2+4 is nothing.) Standard operator precedence rules apply if no parentheses are provided. The following math and logic operators are defined:

operator	meaning
+	addition
-	subtraction
*	multiplication
/	division
%	modulus (remainder after division)
^	exponentiation
=	equal
!=	not equal
>	greater than
<	less than
>=	greater than or equal
<=	less than or equal
~=	general (looks like) equality
&	logical and
|	logical or
!	logical not

Booleans

`#t` is boolean true and `#f` is boolean false.

Conditional Values

```
#? (test, true [, false])
```

Strings

A string of uninterpreted characters enclosed within `""`, e.g. `"Hi ho!"`.

Symbols

Alphanumeric names for variables and functions, e.g. `foo` and `23-skidoo`

Scheme Keywords (constants)

```
:symbol
```

Constant Lists

Sequences of data enclosed within {}, e.g. {1 2 3}. May contain numbers, booleans, symbols, strings, and other lists. Quasi-quotation in constant lists is supported. Use `#$` to unquote and `#^` to unquote-splice.

List element notation

A variable followed by an index inside `[]`, e.g. `foo[1]`

Function call notation

A function name followed by zero or more comma-delimited arguments enclosed within `()`.

Assignment

```
set variable op value [, variable op value ...]
```
→ void

Assigns each *variable* to *value* according to the assignment operator *op*:

operator	meaning
=	set *variable* to *value*
+=	increment *variable* by *value*
*=	scale *variable* by *value*
&=	concatenate *value* onto the end of the list in *variable*
^=	splice the list *value* onto the end of the list in *variable*
@=	concatenate *value* onto the front of the list in *variable*
<=	minimize *variable* with respect to *value*
>=	maximize *variable* with respect to *value*

Conditionals

```
if (test) ... [else ...] end
```
→ value of last expression

If *test* is true execute statements otherwise execute optional *else* statements. See also: conditional value. Note that the parentheses around the *test* expression are required.

Sequencing

```
begin [with ...] ... end
```
→ value of last expression

Sequences one or more SAL statements as a single statement. The optional **with** statement is a local variable declaration that can appear at the start of block statements like `begin`, `loop`, `function` and `process`:

```
with var [= expr] {, var [= expr]}*
```
→ void

which will Define one or more local variables (within the surrounding block), optionally setting inital values to *expr* (or to false if *expr* is not provided).

Definitions

```
variable var [= expr] {, var [= expr]}*
```
→ value of last expression

Defines one or more variables as "global" within the current context.

```
function name ({parameters}*) [with ...] {statement}+ end
```
→ void

A function definition consists of the *name* of the function, a parameter list containing zero or more *required parameters* optionally followed by &optkey and any optkey argument. Following the parameter list comes the statements to be executed when the function is called. The first of these statements can be a with declaration. The last statement in the body of the function becomes the value of the function call.

Files

```
file "file.ext" ({keyword value}*) [with ...] {statement}+ end
```
→ void

A score file definition. The string "*file.ext*" is the pathname of the file to create, where *file* can include a directory component and *ext* must be one of the supported score file types. Following the name of the file comes zero or more keyword options as suitable for the type of file being created. Inside the body of `file` you can write any expressions that send data to the currently open score file.

Appendix

A. Documentation Meta-syntax

- Definition terms are printed in fixed-width bold face:

  cents->scaler

- Literals and programming code are printed in fixed-width face:

  ```
  list(note(60), rhythm("h."))
  ```

- Variables and metavariables (names that stand for pieces of code) are printed in italic face:

  number

- Terms separated by | are exclusive choices where exactly one must apply:

  number | *boolean*

- Braces { } and brackets [ ] are sometimes used to associate or group the enclosed terms and are not part of actual code.

  - Braces group the enclosed terms into a single entity. Thus:

    {*number* | *boolean*}

    describes single value that is either a number or a boolean.

  - Brackets make the enclosed term optional. An optional term may appear at most one time. For example, in the following definition:

    odds(*prob* [,*trueval*] [,*falseval*])

    prob is a required argument that may be followed by up to two optional values. Thus, each of the following calls is correct:

    ```
    odds(.5)
    odds(1, #t)
    odds(.2, "Yup", "Nope")
    ```

- A star * after a brace means that the enclosed terms may be specified zero or more times. For example:

  {*keyword value*}*

  means that any number of *keyword* and *value* pairs, including none, may be specified.

- A plus + after a brace means that the enclosed terms may be specified one or more times.
- A sequence of three dots means the rest of the arguments, no matter how many. For example:

  pick(*arg ...*)

  means that any number of additional values, including none, may follow *arg*.
- Syntax highlighting distinguishes important syntactic units:

keyword	string	constant	reserved	comment
music:	"Hello, world"	#t	loop	;; bug!

B. Function Arguments

Several different types of function arguments are supported:

Required arguments

Required arguments must be specified, that is, each required argument must have an explicit value provided in the function call.

Optional arguments

Optional arguments may be specified or not. If not, they receive a default value, which is #f (false) if not otherwise indicated.

Optional/keyword arguments

Opt/key arguments are optional arguments that can be specified positionally or by preceding the argument value by its keyword name. In Scheme keyword argument names start with a colon and in SAL they end with a colon. For example, if a function `foo` has two opt/key arguments *a* and *b* then in SAL the function could be called in any of the following ways:

```
foo() foo(4) foo(2,3) foo(a: 3) foo(b: 99) foo(a: 2, b: 4) foo(b: 99, a: 2)
```

and in Scheme:

```
(foo) (foo 4) (foo 2 3) (foo :a 3) (foo :b 99) (foo :a 2 :b 4) (foo :b 99 :a 2)
```

Variable arguments

A variable argument takes zero or more values. A variable argument is shown in the documentation as `...` which means 'the rest of the arguments'.

C. Table of SAL, Scheme and Common Lisp

A. Translation table for SAL, Scheme and Common Lisp entities.

	SAL	Scheme [1]	Common Lisp [2]
Booleans	#f	#f	nil

	SAL	Scheme [1]	Common Lisp [2]
	#t	#t	t
Keywords	foobar:	#:foobar *or* :foobar	:foobar
Lists	{}	'()	() *or* nil
	{a b {1 2 3} c}	'(a b (1 2 3) e)	'(a b (1 2 3) e)
	{a #$ ran(100) c}	`(a ,(random 100) c}	`(a ,(random 100) c)
	{a #^ list(ran(100), ran(100)) c}	`(a ,@ (list (random 100) (random 100)) c}	`(a ,@(list (random 100) (random 100)) c)
Arithmetic	1 + 2	(+ 1 2)	(+ 1 2)
	- 1	(- 1)	(- 1)
	1 + 2 + 3 * 4	(+ 1 2 (* 3 4))	(+ 1 2 (* 3 4))
	3 ^ 4	(expt 3 4)	(expt 3 4)
	3 % 4	(modulo 3 4)	(mod 3 4)
	a = b	(= a b)	(= a b)
	a / b != b * c	(not (= (/ a b) (* b c)))	(/= (/ a b) (* b c))
	a ~= b	(equal a b)	(equal a b)
Logic	a | b	(or a b)	(or a b)
	! a | b	(or (not a) b)	(or (not a) b)
	a & b | ! c	(or (and a b) (not c))	(or (and a b) (not c))
Function calls	list()	(list)	(list)
	list(10)	(list 10)	(list 10)
	list(1, 2 + 3, 4)	(list 1 (+ 2 3) 4)	(list 1 (+ 2 3) 4)
	list(1, 2, random(10))	(list 1 2 (random 10))	(list 1 2 (random 10))

Conditionals	if (a = b)
	  list(1,2)
	else
	  list(3,4)
	end
		(if (= a b)
		  (list 1 2)
		  (list 3 4))
			(if (= a b)
			  (list 1 2)
			  (list 3 4))

	#?(a = b, -99, 2)	(if (= a b) -99 2)	(if (= a b) -99 2)

	SAL	Scheme [1]	Common Lisp [2]
Global variables	`variable a`	`(define a #f)`	`(defparameter a nil)`

Function definitions	`function foo (a)`
	`  a + ran(10)`
	`end`
		`(define (foo a)`
		`  (+ a (random 10)))`
			`(defun foo (a)`
			`  (+ a (random 10)))`

Code blocks	`begin`
	`  ...`
	`end`
		`(begin`
		`  ...`
		`  )`
			`(progn`
			`  ...`
			`  )`

	`begin`
	`  with a = 1, b = a + 2`
	`  ...`
	`end`
		`(let* ((a 1)`
		`  (b (+ a 2))`
		`  ...`
		`  )`
			`(let* ((a 1)`
			`  (b (+ a 2))`
			`  ...`
			`  )`

Assignment	`set a = 123`	`(set! a 123)`	`(setf a 123)`
	`set a = 123, b = a`	`(set! a 123)` `(set! b a)`	`(setf a 123 b a)`
	`set a += 123`	`(set! a (+ a 123))`	`(incf a 123)`
	`set a *= 123`	`(set! a (* a 123))`	`(setf a (* a 123))`
	`set a &= 123`	`(set! a (append a (list 123)))`	`(setf a (append a (list 123)))`
	`set a @= 123`	`(set! a (cons 123 a))`	`(push 123 a)`
	`set a <= 123`	`(set! a (min 123 a))`	`(setf a (min 123 a))`
	`set a >= 123`	`(set! a (max 123 a))`	`(setf a (max 123 a))`
Iteration	`loop ... end`	`(loop ...)`	`(loop ... )`

1. Reference Scheme implementation: Chicken Scheme
2. Reference Common Lisp: Steel Bank Common Lisp (SBCL)

Index

b

between, butlast

c

cell-state, cents->ratio, chdir, concat, cont, cs:event, cs:f, cs:i

d

decimals, deltas, directory, discrete, divide, drunk

e

elapsed, eod?, eop?

f

file-version, first, fit, fm-spectrum, fms:clear-score, fms:delete-score, fms:inst, fms:load-score, fms:mark, fms:meas, fms:measdef, fms:merge-score, fms:metapart, fms:new-score, fms:note, fms:part, fms:percinst, fms:rest, fms:run, fms:save-score, fms:select-score, fms:setting

g

greater=?, greater?

h

harmonics, hertz

i

in-tempo, interp, invert

k

keynum

l

last, less=?, less?, list-difference, list-intersection, list-set!, list-union, load, log10, log2

m

make-automata, make-cycle, make-graph, make-heap, make-list, make-markov, make-palindrome, make-repeater, make-rotation, make-spectrum, make-weighting, markov-analyze, midifile-header, midifile-import, minus, most-negative-fixnum, most-positive-fixnum, mouse-button, mouse-x, mouse-y, mp:bend, mp:ctrl, mp:inchans, mp:inops, mp:instruments, mp:midi, mp:off, mp:on, mp:press, mp:prog, mp:receive, mp:receive?, mp:touch

n

next, note, now, nth

o

odds, osc:bundle, osc:close, osc:message, osc:open, osc:receive, osc:receive?

p

pause, pitch-class, pi, pick, plot, plot-data, plot-hook, plus, print, promise, pwd

q

quantize qsort!

r

ran, ranbeta, ranbrown, rancauchy, random-seed, random-series, ranexp, rangamma, rangauss, ranhigh, ranlow, ranmiddle, ranpink, ranpoisson, ratio->cents, ratio->steps, rescale, remove-duplicates, rest, rest?, retrograde, rhythm, rm-spectrum

s

scale, scale-order, sdif-import, sdif-import-spectra, segs, shell, shuffle, shuffle!, spear-export-spectra, spear-import-spectra, spectrum-add!, spectrum-amps, spectrum-copy, spectrum-flip!, spectrum-freqs, spectrum-invert!, spectrum-keys, spectrum-maxamp, spectrum-maxfreq, spectrum-minamp, spectrum-minfreq, spectrum-pairs, spectrum-rescale!, spectrum-size, spectrum-time, sprout, sprout-hook, state, stop

t

tail, tendency, times, transpose

v

vary

x

xy