Nyquist Reference Manual

Version 2.0

Roger B. Dannenberg

2 January 1996

Carnegie Mellon University
School of Computer Science
Pittsburgh, PA 15213, U.S.A.

.

Preface

This manual is a guide for users of Nyquist, a language for composition and sound synthesis. Nyquist grew out of a series of research projects, notably the languages Arctic and Canon. Along with Nyquist, these languages promote a functional style of programming and incorporate time into the language semantics.

Please help by noting any errors, omissions, or suggestions you may have. You can send your suggestions to Dannenberg@CS.CMU.EDU (internet) via computer mail, or by campus mail to Roger B. Dannenberg, School of Computer Science, or by ordinary mail to Roger B. Dannenberg, School of Computer Science, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213-3890, USA.

Nyquist is a successor to Fugue, a language originally implemented by Chris Fraley, and extended by George Polly and Roger Dannenberg. Peter Velikonja and Dean Rubine were early users, and they proved the value as well as discovered some early problems of the system. This led to Nyquist, a reimplementation of Fugue by Roger Dannenberg with help from Joe Newcomer and Cliff Mercer.

I also wish to acknowledge support from CMU, Yamaha, and IBM for this work.

.

1. Introduction and Overview

Nyquist is a language for sound synthesis and music composition. Unlike score languages that tend to deal only with events, or signal processing languages that tend to deal only with signals and synthesis, Nyquist handles both in a single integrated system. Nyquist is also flexible and easy to use because it is based on an interactive Lisp interpreter.

With Nyquist, you can design instruments by combining functions (much as you would using the orchestra languages of Music V, cmusic, or Csound). You can call upon these instruments and generate a sound just by typing a simple expression. You can combine simple expressions into complex ones to create a whole composition.

Nyquist runs under any Unix environment and it produces sound files as output. If you can play a sound file by typing a command to a Unix shell, then you can get Nyquist to play sounds for you. Nyquist is currently configured to run on an IBM RS6000 with an ACPA audio board, or a NeXT machine, using the built-in sound system to play Nyquist output. Recent versions have also run on SGI, DEC pmax, and Sun Sparc machines, and makefiles for these are included. Let me know if you have problems with any of these machines.

To use Nyquist, you should have a basic knowledge of Lisp. An excellent text by Touretzky is recommended [Touretzky 84]. Appendix III is the reference manual for XLISP, of which Nyquist is a superset.

1.1. Installation

Nyquist is a C program intended to run under the Unix operating system. Nyquist is distributed as a compressed tar file named `nyquist.tar.Z`. To install Nyquist, copy `nyquist.tar.Z` to a fresh directory on your machine and type:

```
zcat nyquist.tar.Z | tar xf -
ln -s sys/unix/next/Makefile
setenv XLISPPATH ‘pwd‘/runtime
make
```

The first line creates a `nyquist` directory and some subdirectories. The second line makes a link from the top-level directory to the Makefile for your system. In place of `next` in `sys/unix/next/Makefile`, you should substitute your system type. Current systems are `next`, `pmax`, `rs6k`, `sgi`, and `sparc`. The `setenv` command tells Nyquist where to search for lisp files to be loaded when a file is not found in the current directory. The `runtime` directory should always be on your `XLISPPATH` when you run Nyquist, so you may want to set `XLISPPATH` in your shell startup file, e.g. `.cshrc`. Assuming the make completes successfully, you can run Nyquist as follows:

```
./ny
```

When you get the prompt, you may begin typing expressions such as the ones in the following section.

Note: Nyquist looks for the file `init.lsp` in the current directory. If you look in the `init.lsp` in `src`, you will notice two things. First, `init.lsp` loads `nyquist.lsp` from the Nyquist directory, and second, `init.lsp` loads `play.lsp` which in turn defines the macro `play`. You will probably have to modify `play.lsp` to invoke the right programs on your machine.

1.2. Examples

We will begin with some simple Nyquist programs. Detailed explanations of the functions used in these examples will be presented in later chapters, so at this point, you should just read these examples to get a sense of how Nyquist is used and what it can do. The details will come later. Most of these examples can be found in the file `nyquist/sndtest/tutorial.lsp`.

Our first example makes and plays a sound:

```
;; Making a sound.
(play (osc 60))   ; generate a loud sine wave
```

This example is about the simplest way to create a sound with Nyquist. The `osc` function generates a sound using a table-lookup oscillator. There are a number of optional parameters, but the default is to compute a sinusoid with an amplitude of 1.0. The parameter `60` designates a pitch of middle C. (Pitch specification will be described in greater detail later.) The result of the `osc` function is a sound. To hear a sound, you must use the `play` function, which (at least in the current implementation) writes the sound as a 16-bit sound file and runs a Unix program that plays the file through the machine's D/A converters.

Note: when Nyquist plays a sound, it scales the signal by 2^{15}-1 and (by default) converts to a 16-bit integer format. A signal like `(osc 60)`, which ranges from +1 to -1, will play as a full-scale 16-bit audio signal. Signals are not normalized to full-scale, however, so an amplitude in excess of 1 will be clipped. See Section 4.3 for information about normalization.

1.2.1. Waveforms

Our next example will be presented in several steps. The goal is to create a sound using a wavetable consisting of several harmonics as opposed to a simple sinusoid. In order to build a table, we will use a function that computes a single harmonic and add harmonics to form a wavetable. An oscillator will be used to compute the harmonics.

The function `mkwave` calls upon `build-harmonic` to generate a total of four harmonics with amplitudes 1.0, 0.5, 0.25, and 0.12. These are scaled (using `scale`) and added (using `sim`) to create a waveform which is bound temporarily to `*table*`.

A complete Nyquist waveform is a list consisting of a sound, a pitch, and `T`, indicating a periodic waveform. The pitch gives the nominal pitch of the sound. (This is implicit in a single cycle wave table, but a sampled sound may have many periods of the fundamental.) Pitch is expressed in half-steps, where middle C is 60 steps, as in MIDI pitch numbers. The list of sound, pitch, and `T` is formed in the last line of `mkwave`: since `build-harmonic` computes signals with a duration of one second, the fundamental is 1 Hz, and the `hz-to-step` function converts to pitch (in units of steps) as required.

```
(defun mkwave ()
  (setf *table* (sim (scale 0.5  (build-harmonic 1.0 2048))
                     (scale 0.25  (build-harmonic 2.0 2048))
                     (scale 0.125 (build-harmonic 3.0 2048))
                     (scale 0.062 (build-harmonic 4.0 2048))))
  (setf *table* (list *table* (hz-to-step 1) T)))
```

Now that we have defined a function, the last step of this example is to build the wave. The following code calls `mkwave` the first time the code is executed (loaded from a file). The second time, the variable `*mkwave*` will be true, so `mkwave` will not be invoked:

```
(cond ((not (boundp '*mkwave*))
       (mkwave)
       (setf *mkwave* t)))
```

1.2.2. Sequences

Finally, we define `note` to use the waveform, and play several notes in a simple score:

```
(defun note (pitch dur)
  (osc pitch dur *table*))

(play (seq (note c4 i)
           (note d4 i)
           (note f4 i)
           (note g4 i)
           (note d4 q)))
```

Here, `note` is defined to take pitch and duration as parameters; it calls `osc` to do the work of generating a waveform, using `*table*` as a wave table.

The `seq` function is used to invoke a sequence of behaviors. Each note is started at the time the previous note finishes. The parameters to `note` are predefined in Nyquist: `c4` is middle C, `i` (for eIghth note) is 0.5, and `q` (for Quarter note) is 1.0. See Section 1.3 for a complete description. The result is the sum of all the computed sounds.

1.2.3. Envelopes

The next example will illustrate the use of envelopes. In Nyquist, envelopes are just ordinary sounds (although they normally have a low sample rate). An envelope is applied to another sound by multiplication using the `mult` function. The code shows the definition of `env-note`, defined in terms of the `note` function in the previous example. In `env-note`, a 4-phase envelope is generated using the `env` function, which is illustrated in Figure 1.

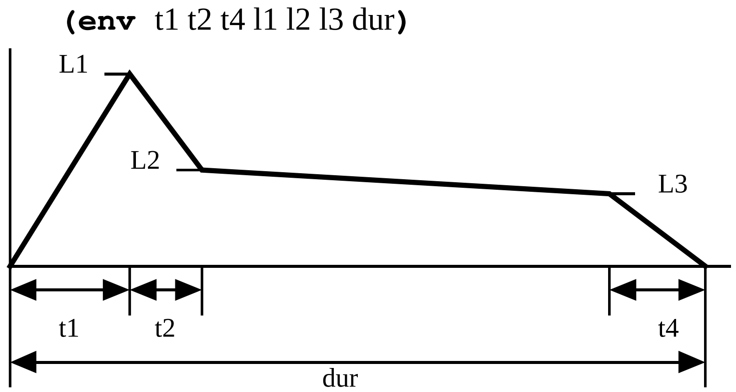

Figure 1: An envelope generated by the `env` function.

```
; env-note produces an enveloped note.  The duration
;   defaults to 1.0, but stretch can be used to change
;   the duration.
;
(defun env-note (p)
  (mult (note p 1.0)
        (env 0.05 0.1 0.5 1.0 0.5 0.4)))

; try it out:
;
(play (env-note c4))


; now use stretch to play different durations
;
(play
       (seq (stretch 0.25
                     (seq (env-note c4)
                          (env-note d4)))
            (stretch 0.5
                     (seq (env-note f4)
                          (env-note g4)))
            (env-note c4)))
```

The end of this example shows the use of `stretch` to modify durations. There are several transformations supported by Nyquist, and transformations of abstract behaviors is perhaps *the* fundamental idea behind Nyquist. Chapter 2 is devoted to explaining this concept, and further elaboration can be found elsewhere [Dannenberg 89].

1.2.4. Piece-wise Linear Functions

It is often convenient to construct signals in Nyquist using a list of (time, value) breakpoints which are linearly interpolated to form a smooth signal. Envelopes created by `env` are a special case of the more general piece-wise linear functions created by `pwl`. Since `pwl` is used in some examples later on, we will take a look at `pwl` now. The `pwl` function takes a list of parameters which denote (time, value) pairs. There is an implicit initial (time, value) pair of (0, 0), and an implicit final value of 0. There should always be an odd number of parameters, since the final time is not implicit. Here are some examples:

```
; symetric rise to 10 (at time 1) and fall back to 0 (at time 2):
;
(pwl 1 10 2)

; a square pulse of height 10 and duration 5.
; Note that the first pair (0, 10) overrides the default initial
; point of (0, 0).  Also, there are two points specified at time 5:
; (5, 10) and (5, 0).  (The last 0 is implicit).  The conflict is
; automatically resolved by pushing the (5, 10) breakpoint back to
; the previous sample, so the actual time will be 5 - 1/sr, where
; sr is the sample rate.
;
(pwl 0 10 5 10 5)

; a constant function with the value zero over the time interval
; 0 to 3.5.  This is a very degenerate form of pwl.  Recall that there
; is an implicit initial point at (0, 0) and a final implicit value of
; 0, so this is really specifying two breakpoints: (0, 0) and (3.5, 0):
;
(pwl 3.5)

; a linear ramp from 0 to 10 and duration 1.
; Note the ramp returns to zero at time 1.  As with the square pulse
; above, the breakpoint (1, 10) is pushed back to the previous sample.
;
(pwl 1 10 1)

; If you really want a linear ramp to reach its final value at the
; specified time, you need to make a signal that is one sample longer.
; The RAMP function does this:
;
(ramp 10) ; ramp from 0 to 10 with duration 1 + one sample period
;
; RAMP is based on PWL; it is defined in nyquist.lsp.
;
```

1.3. Pre-defined Constants

For convenience and readability, Nyquist pre-defines some constants, mostly based on the notation of the Adagio score language, as follows:

- **Dynamics** Note: these dynamics values are subject to change.

  `lppp` = -12.0 (dB)
  `lpp` = -9.0
  `lp` = -6.0
  `lmp` = -3.0
  `lmf` = 3.0
  `lf` = 6.0
  `lff` = 9.0
  `lfff` = 12.0
  `dB0` = 1.00
  `dB1` = 1.122
  `dB10` = 3.1623

- **Durations**

`s` = Sixteenth = 0.25
`i` = eIghth = 0.5
`q` = Quarter = 1.0
`h` = Half = 2.0
`w` = Whole = 4.0
`sd, id, qd, hd, wd` = dotted durations.
`st, it, qt, ht, wt` = triplet durations.

- **Pitches**Pitches are based on an A4 of 440Hz. To achieve a different tuning, set `*A4-Hertz*` to the desired frequency for A4, and call `(set-pitch-names)`. This will recompute the names listed below with a different tuning. In all cases, the pitch value 69.0 corresponds exactly to 440Hz, but fractional values are allowed, so for example, if you set `*A4-Hertz*` to 444 (Hz), then the symbol `A4` will be bound to 69.1567, and `C4` (middle C), which is normally 60.0, will be 60.1567.

  `c0` = 12.0
  `cs0, df0` = 13.0
  `d0` = 14.0
  `ds0, ef0` = 15.0
  `e0` = 16.0
  `f0` = 17.0
  `fs0, gf0` = 18.0
  `g0` = 19.0
  `gs0, af0` = 20.0
  `a0` = 21.0
  `as0, bf0` = 22.0
  `b0` = 23.0
  `c1` ... `b1` = 24.0 ... 35.0
  `c2` ... `b2` = 36.0 ... 47.0
  `c3` ... `b3` = 48.0 ... 59.0
  `c4` ... `b4` = 60.0 ... 71.0
  `c5` ... `b5` = 72.0 ... 83.0
  `c6` ... `b6` = 84.0 ... 95.0
  `c7` ... `b7` = 96.0 ... 107.0
  `c8` ... `b8` = 108.0 ... 119.0

2. Behavioral Abstraction

In Nyquist, all functions are subject to transformations. You can think of transformations as additional parameters to every function, and functions are free to use these additional parameters in any way. The set of transformation parameters is captured in what is referred to as the *transformation environment*. (Note that the term *environment* is heavily overloaded in computer science. This is yet another usage of the term.)

Behavioral abstraction is the ability of functions to adapt their behavior to the transformation environment. This environment may contain certain abstract notions, such as loudness, stretching a sound in time, etc. These notions will mean different things to different functions. For example, an oscillator should produce more periods of oscillation in order to stretch its output. An envelope, on the other hand, might only change the duration of the sustain portion of the envelope in order to stretch. Stretching a sample could mean resampling it to change its duration by the appropriate amount.

Thus, transformations in Nyquist are not simply operations on signals. For example, if I want to stretch a note, it does not make sense to compute the note first and then stretch the signal. Doing so would cause a drop in the pitch. Instead, a transformation modifies the *transformation environment* in which the note is computed. Think of transformations as making requests to functions. It is up to the function to carry out the request. Since the function is always in complete control, it is possible to perform transformations with ‘‘intelligence;’’ that is, the function can perform an appropriate transformation, such as maintaining the desired pitch and stretching only phase 3 of an envelope to obtain a longer note.

2.1. The Environment

The transformation environment consists of a set of special Lisp variables. These variables should not be read directly and should *never* be set directly by the programmer. Instead, there are functions to read them, and they are automatically set and restored by transformation operators, which will be described below.

The transformation environment consists of the following elements. Although each element has a ‘‘standard interpretation,’’ the designer of an instrument or the composer of a complex behavior is free to interpret the environment in any way. For example, a change in `*loud*` may change timbre more than amplitude, and `*transpose*` may be ignored by percussion instruments:

`*warp*` Time transformation, including time shift, time stretch, and continuous time warp. The value of `*warp*` is interpreted as a function from logical (local score) time to physical (global real) time. Do not access `*warp*` directly. Instead, use `(local-to-global` *t*`)` to convert from a logical (local) time to physical (global) time. Most often, you will call `(local-to-global 0)`. Several transformation operators operate on `*warp*`, including `at`, `stretch`, and `warp`.

`*loud*` Loudness, expressed in decibels. The default (nominal) loudness is 0.0 dB (no change). Do not access `*loud*` directly. Instead, use `(get-loud)` to get the current value of `*loud*` and either `loud` or `loud-abs` to modify it.

`*transpose*` Pitch transposition, expressed in semitones. (Default: 0.0). Do not access `*transpose*` directly. Instead, use `(get-transpose)` to get the current value of `*transpose*` and either `transpose` or `transpose-abs` to modify it.

`*sustain*` The ‘‘sustain,’’ ‘‘articulation,’’ ‘‘duty factor,’’ or amount by which to separate or overlap sequential notes. For example, staccato might be expressed with a `*sustain*` of 0.5, while very legato playing might be expressed with a `*sustain*` of 1.2. Specifically, `*sustain*` stretches the duration of notes (sustain) without affecting the inter-onset time (the rhythm). Do not access `*sustain*` directly. Instead, use `(get-sustain)` to get the current value of `*sustain*` and either `sustain` or `sustain-abs` to modify it.

`*start*`
: Start time of a clipping region. *Note:* unlike the previous elements of the environment, `*start*` has a precise interpretation: no sound should be generated before `*start*`. This is implemented in all the low-level sound functions, so it can generally be ignored. You can read `*start*` directly, but use `extract` or `extract-abs` to modify it. ***Note 2:*** Due to some internal confusion between the specified starting time and the actual starting time of a signal after clipping, `*start*` is not fully implemented.

`*stop*`
: Stop time of clipping region. By analogy to `*start*`, no sound should be generated after this time. `*start*` and `*stop*` allow a composer to preview a small section of a work without computing it from beginning to end. You can read `*stop*` directly, but use `extract` or `extract-abs` to modify it. ***Note:*** Due to some internal confusion between the specified starting time and the actual starting time of a signal after clipping, `*stop*` is not fully implemented.

`*control-srate*`
: Sample rate of control signals. This environment element provides the default sample rate for control signals. There is no formal distinction between a control signal and an audio signal. You can read `*control-srate*` directly, but use `control-srate` or `control-srate-abs` to modify it.

`*sound-srate*`
: Sample rate of musical sounds. This environment element provides the default sample rate for musical sounds. You can read `*sound-srate*` directly, but use `sound-srate` or `sound-srate-abs` to modify it.

2.2. Sequential Behavior

Previous examples have shown the use of `seq`, the sequential behavior operator. We can now explain `seq` in terms of transformations. Consider the simple expression:

```
(play (seq (note c4 q) (note d4 i)))
```

The idea is to create the first note at time 0, and to start the next note when the first one finishes. This is all accomplished by manipulating the environment. In particular, `*warp*` is modified so that what is locally time 0 for the second note is transformed, or warped, to the logical stop time of the first note.

One way to understand this in detail is to imagine how it might be executed: first, `*warp*` is set to an initial value that has no effect on time, and `(note c4 q)` is evaluated. A sound is returned and saved. The sound has an ending time, which in this case will be `1.0` because the duration `q` is `1.0`. This ending time, `1.0`, is used to construct a new `*warp*` that has the effect of shifting time by 1.0. The second note is evaluated, and will start at time 1. The sound that is returned is now added to the first sound to form a composite sound, whose duration will be `2.0`. `*warp*` is restored to its initial value.

Notice that the semantics of `seq` can be expressed in terms of transformations. To generalize, the operational rule for `seq` is: evaluate the first behavior according to the current `*warp*`. Evaluate each successive behavior with `*warp*` modified to shift the new note's starting time to the ending time of the previous behavior. Restore `*warp*` to its original value and return a sound which is the sum of the results.

In the Nyquist implementation, audio samples are only computed when they are needed, and the second part of the `seq` is not evaluated until the ending time (called the logical stop time) of the first part. It is still the case that when the second part is evaluated, it will see `*warp*` bound to the ending time of the first part.

A language detail: Even though Nyquist defers evaluation of the second part of the `seq`, the expression can reference variables according to ordinary Lisp scope rules. This is because the `seq` captures the expression in a closure, which retains all of the variable bindings.

2.3. Simultaneous Behavior

Another operator is `sim`, which invokes multiple behaviors at the same time. For example,

```
(play (scale 0.5 (sim (note c4 q) (note d4 i))))
```

will play both notes starting at the same time.

The operational rule for `sim` is: evaluate each behavior at the current `*warp*` and return the result. The following example uses `sim` and illustrates two concepts: first, a *sound* is not a *behavior*, and second, the `at` transformation can be used to place sounds in time.

2.4. Sounds vs. Behaviors

The following example loads a sound from a file in the current directory and stores it in `a-snd`:

```
; load a sound
;
(setf a-snd (s-read "./demo-snd.nh" :srate 22050.0))

; play it
;
(play a-snd)
```

One might think that the following would then work:

```
(seq a-snd a-snd)  ;WRONG!
```

but in fact, the result would not be a sequence of two sounds. Why? Recall that `seq` works by modifying `*warp*`, not by operating on sounds. So, `seq` will proceed by evaluating `a-snd` with different values of `*warp*`. However, the result of evaluating `a-snd` (a Lisp variable) is always the same sound, regardless of the environment; in this case, the second `a-snd` will start at time `0.0`, just like the first.

How then do we obtain a sequence of two sounds? What we really need here is a behavior that transforms a given sound according to the current transformation environment. That job is performed by `cue`. For example, the following will behave as expected, producing a sequence of two sounds:

```
(seq (cue a-snd) (cue a-snd))
```

The lesson here is very important: ***sounds are not behaviors!*** Behaviors are computations that generate sounds according to the transformation environment. Once a sound has been generated, it can be stored, copied, added to other sounds, and used in many other operations, but sounds are *not* subject to transformations. To transform a sound, use `cue`, `sound`, or `control`. The differences between these operations are discussed later. For now, here is a ‘‘cue sheet’’ style score that plays 4 copies of `a-snd`:

```
; use sim and at to place sounds in time
;
(play (sim (at 0.0 (cue a-snd))
           (at 0.7 (cue a-snd))
           (at 1.0 (cue a-snd))
           (at 1.2 (cue a-snd))))
```

2.5. The At Transformation

The second concept introduced by the previous example is the `at` operation, which shifts the `*warp*` component of the environment. For example,

```
(at 0.7 (cue a-snd))
```

can be explained operationally as follows: modify `*warp*` by shifting it by `0.7` and evaluate `(cue a-snd)`. Return the resulting sound after restoring `*warp*` to its original value. Notice how `at` is used inside a `sim` construct to locate copies of `a-snd` in time. This is the standard way to represent a note-list or a cue-sheet in Nyquist.

This also explains why sounds need to be `cue`'d in order to be shifted in time or arranged in sequence. If this were not the case, then `sim` would take all of its parameters (a set of sounds) and line them up to start at the same time. But `(at 0.7 (cue a-snd))` is just a sound, so `sim` would ''undo'' the effect of `at`, making all of the sounds in the previous example start simultaneously, in spite of the `at`. Since `sim` respects the intrinsic starting times of sounds, a special operation, `cue`, is needed to create a new sound with a new starting time.

2.6. Nested Transformations

Transformations can be combined using nested expressions. For example,

```
(sim (cue a-snd)
     (loud 6.0 (at 3.0 (cue a-snd))))
```

scales the amplitude as well as shifts the second entrance of `a-snd`.

Transformations can also be applied to groups of behaviors:

```
(loud 6.0 (sim (at 0.0 (cue a-snd))
               (at 0.7 (cue a-snd))))
```

2.7. Defining Behaviors

Groups of behaviors can be named using `defun` (we already saw this in the definitions of `note` and `note-env`). Here is another example of a behavior definition and its use. The definition has one parameter:

```
(defun snds (dly)
  (sim (at 0.0 (cue a-snd))
       (at 0.7 (cue a-snd))
       (at 1.0 (cue a-snd))
       (at (+ 1.2 dly) (cue a-snd))))

(play (snds 0.1))
(play (loud 0.25 (stretch 0.9 (snds 0.3))))
```

In the last line, `snds` is transformed: the transformations will apply to the `cue` behaviors within `snds`. The `loud` transformation will scale the sounds by `0.25`, and `stretch` will apply to the shift (`at`) amounts `0.0`, `0.7`, `1.0`, and `(+ 1.2 dly)`. The sounds themselves (copies of `a-snd`) will not be stretched because `cue` never stretches sounds.

Section 5.3 describes the full set of transformations.

2.8. Sample Rates

The global environment contains `*sound-srate*` and `*control-srate*`, which determine the sample rates of sounds and control signals. These can be overridden at any point by the transformations `sound-srate-abs` and `control-srate-abs`; for example,

```
(sound-srate-abs 44100.0 (osc c4))
```

will compute a tone using a 44.1Khz sample rate.

As with other components of the environment, you should *never* change `*sound-srate*` or `*control-srate*` directly with `setf` or even `let`. The global environment is determined by two additional variables: `*default-sound-srate*` and `*default-control-srate*`. You can add lines like the following to your `init.lsp` file to change the default global environment:

```
(setf *default-sound-srate* 44100.0)
(setf *default-control-srate* 1102.5)
```

If you have already started Nyquist and want to change the defaults, the following functions should be used:

```
(set-control-srate 1102.5)(set-sound-srate 22050.0)
```

These modify the default values and reinitialize the Nyquist environment.

3. Continuous Transformations and Time Warps

Nyquist transformations were discussed in the previous chapter, but all of the examples used scalar values. For example, we saw the `loud` transformation used to change loudness by a fixed amount. What if we want to specify a crescendo, where the loudness changes gradually over time?

It turns out that all transformations can accept signals as well as numbers, so transformations can be continuous over time. This raises some interesting questions about how to interpret continuous transformations. Should a loudness transformation apply to the internal details of a note or only affect the initial loudness? It might seem unnatural for a decaying piano note to perform a crescendo. On the other hand, a sustained trumpet sound should probably crescendo continuously. In the case of time warping (tempo changes), it might be best for a drum roll to maintain a steady rate, a trill may or may not change rates with tempo, and a run of sixteenth notes will surely change its rate.

These issues are complex, and Nyquist cannot hope to automatically do the right thing in all cases. However, the concept of behavioral abstraction provides an elegant solution. Since transformations merely modify the environment, behaviors are not forced to implement any particular style of transformation. Nyquist is designed so that the default transformation is usually the right one, but it is always possible to override the default transformation to achieve a particular effect.

3.1. Simple Transformations

The ‘‘simple’’ transformations affect some parameter, but have no effect on time itself. The simple transformations that support continuously changing parameters are: `sustain`, `loud`, and `transpose`.

As a first example, Let us use `transpose` to create a chromatic scale. First define a sequence of tones at a steady pitch: `(defun tone-seq () (seqrep (i 16) (stretch 0.25 (osc-note c4))))` Now define a linearly increasing ramp to serve as a transposition function: `(defun pitch-rise () (stretch 4.0 (scale 16 (ramp))))` This ramp has a duration of 4 seconds, and over that interval it rises from 0 to 16 (corresponding to the 16 semitones we want to transpose). Now, `pitch-rise` is used to transpose `tone-seq`: `(defun chromatic-scale () (transpose (pitch-rise) (tone-seq)))`

Similar transformations can be constructed to change the sustain or ‘‘duty factor’’ of notes and their loudness. The following expression plays the previously constructed chromatic scale with increasing note durations. The rhythm is unchanged, but the note length changes from staccato to legato: `(sustain (stretch 4 (sum 0.2 (ramp))) (chromatic-scale))` The resulting sustain function will ramp from 0.2 to 1.2. A sustain of 1.2 denotes a 20 percent overlap between notes. The `sum` has a stretch factor of 4, so it will extend over the 4 second duration of `chromatic-scale`.

What do these transformations mean? How did the system know to produce a pitch rise rather than a continuous glissando? This all relates to the idea of behavioral abstraction. It is possible to design sounds that *do* glissando under the transpose transform, and you can even make sounds that *ignore* transpose altogether. As explained in Chapter 2, the transformations modify the environment, and behaviors can reference the environment to determine what signals to generate. All built-in functions, such as `osc`, have a default behavior.

The default behavior for sound primitives under `transpose`, `sustain`, and `loud` transformations is to sample the environment at the beginning of the note. Transposition is not quantized to semitones or any other scale, but in our example, we arranged for the transposition to work out to integer numbers of semitones, so we got a chromatic scale.

Transposition only applies to the oscillator and sampling primitives `osc`, `partial`, `sampler`, `sine`, `fmosc`, and `amosc`. Sustain applies to `osc`, `env`, and `pwl`. (Note that `amosc` and `fmosc` get their durations from the modulation signal, so they may indirectly depend upon the sustain.) Loud applies to `osc`, `sampler`, `cue`, `sound`, `fmosc`, and `amosc`. (But not `pwl` or `env`.)

3.2. Time Warps

The most interesting transformations have to do with transforming time itself. The `warp` transformation provides a mapping function from logical (score) time to real time. The slope of this function tells us how many units of real time are covered by one unit of score time. This is proportional to 1/tempo. A higher slope corresponds to a slower tempo.

To demonstrate `warp`, we will define a time warp function using `pwl`:

```
(defun warper ()
  (pwl .25 .4 .75 .6 1.0 1.0 2.0 2.0 2.0))
```

This function has an initial slope of .4/.25 = 1.6. It may be easier to think in reciprocal terms: the initial tempo is .25/.4 = .625. Between 0.25 and 0.75, the tempo is .5/.2 = 2.5, and from 0.75 to 1.0, the tempo is again .625. It is important for warp functions to completely span the interval of interest (in our case it will be 0 to 1), and it is safest to extend a bit beyond the interval, so we extend the function on to 2.0 with a tempo of 1.0. Next, we stretch and scale the `warper` function to cover 4 seconds of score time and 4 seconds of real time:

```
(defun warp4 () (stretch 4 (scale 4 (warper))))
```

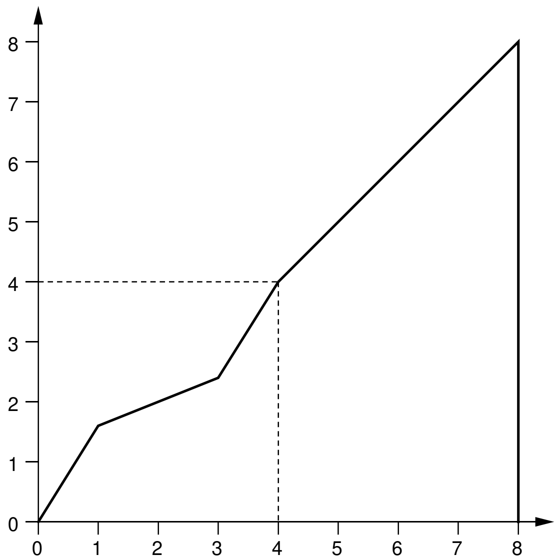

Figure 2: The result of `(warp4)`, intended to map 4 seconds of score time into 4 seconds of real time. The function extends beyond 4 seconds (the dashed lines) to make sure the function is well-defined at location (4, 4). Nyquist sounds are ordinarily open on the right.

Figure 2 shows a plot of this warp function. Now, we can warp the tempo of the `tone-seq` defined above using `warp4`:

```
(play (warp (warp4) (tone-seq)))
```

Figure 3 shows the result graphically. Notice that the durations of the tones are warped as well as their onsets. Envelopes are not shown in detail in the figure. Because of the way `env` is defined, the tones will have constant attack and decay times, and the sustain will be adjusted to fit the available time.

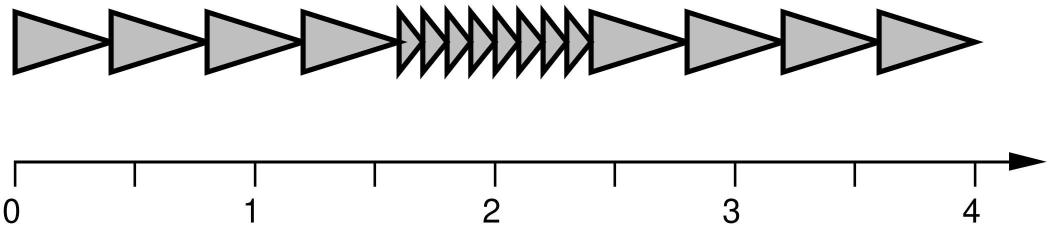

Figure 3: When `(warp4)` is applied to `(tone-seq-2)`, the note onsets and durations are warped.

3.3. Abstract Time Warps

We have seen a number of examples where the default behavior did the ‘‘right thing,’’ making the code straightforward. This is not always the case. Suppose we want to warp the note onsets but not the durations. We will first look at an incorrect solution and discuss the error. Then we will look at a slightly more complex (but correct) solution.

The default behavior for most Nyquist built-in functions is to sample the time warp function at the nominal starting and ending score times of the primitive. For many built-in functions, including `osc`, the starting logical time is 0 and the ending logical time is 1, so the time warp function is evaluated at these points to yield real starting and stopping times, say 15.23 and 16.79. The difference (e.g. 1.56) becomes the signal duration, and there is no internal time warping. The `pwl` function behaves a little differently. Here, each breakpoint is warped individually, but the resulting function is linear between the breakpoints.

A consequence of the default behavior is that notes stretch when the tempo slows down. Returning to our example, recall that we want to warp only the note onset times and not the duration. One would think that the following would work:

```
(defun tone-seq-2 ()
  (seqrep (i 16)
          (stretch-abs 0.25 (osc-note c4))))

(play (warp (warp4) (tone-seq-2)))
```

Here, we have redefined `tone-seq`, renaming it to `tone-seq-2` and changing the `stretch` to `stretch-abs`. The `stretch-abs` should override the warp function and produce a fixed duration.

If you play the example, you will hear steady sixteenths and no tempo changes. What is wrong? In a sense, the ‘‘fix’’ works too well. Recall that sequences (including `seqrep`) determine the starting time of the next note from the logical stop time of the previous sound in the sequence. When we forced the stretch to 0.25, we also forced the logical stop time to 0.25 real seconds from the beginning, so every note starts 0.25 seconds after the previous one, resulting in a constant tempo.

Now let us design a proper solution. The trick is to use `stretch-abs` as before to control the duration, but to restore the logical stop time to a value that results in the proper inter-onset time interval:

```
(defun tone-seq-3 ()
  (seqrep (i 16)
          (set-logical-stop
             (stretch-abs 0.25 (osc-note c4))
             0.25)))

(play (warp (warp4) (tone-seq-3)))
```

Notice the addition of `set-logical-stop` enclosing the `stretch-abs` expression to set the logical stop time. A possible point of confusion here is that the logical stop time is set to 0.25, the same number given to `stretch-abs`! How does setting the logical stop time to 0.25 result in a tempo change? When used within a `warp` transformation, the second argument to `set-logical-stop` refers to *score* time rather than *real* time. Therefore, the score duration of 0.25 is warped into real time, producing tempo changes according to the enviroment. Figure 4 illustrates the result graphically.

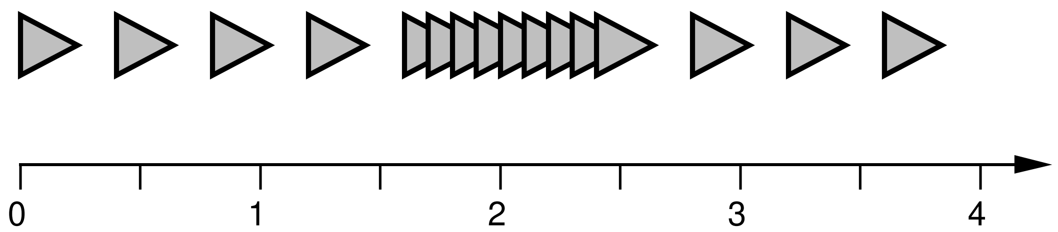

Figure 4: When `(warp4)` is applied to `(tone-seq-3)`, the note onsets are warped, but not the duration, which remains a constant 0.25 seconds. In the fast middle section, this causes notes to overlap. Nyquist will sum (mix) them.

3.4. Nested Transformations

Transformations can be nested. In particular, a simple transformation such as transpose can be nested within a time warp transformation. Suppose we want to warp our chromatic scale example with the `warp4` time warp function. As in the previous section, we will show an erroneous simple solution followed by a correct one.

The simplest approach to a nested transformation is to simply combine them and hope for the best:

```
(play (warp (warp4)
            (transpose (pitch-rise) (tone-seq))))
```

This example will not work the way you might expect. Here is why: the warp transformation applies to the `(pitch-rise)` expression, which is implemented using the `ramp` function. The default behavior of `ramp` is to interpolate linearly (in real time) between two points. Thus, the ‘‘warped’’ `ramp` function will not truly reflect the internal details of the intended time warp. What we need is a way to properly compose the warp and ramp functions. This will lead to a correct solution.

Here is the modified code to properly warp a transposed sequence. Note that the original sequence is used without modification. The only complication is producing a properly warped transposition function:

```
(play (warp (warp4)
            (transpose
                (control-warp (get-warp)
                              (warp-abs nil (pitch-rise)))
                (tone-seq))))
```

To properly warp the `pitch-rise` transposition function, we use `control-warp`, which applies a warp function to a function of score time, yielding a function of real time. We need to pass the warp desired function to `control-warp`, so we fetch it from the environment ith `(get-warp)`. Finally, since the warping is done here, we want to shield the `pitch-rise` expression from further warping, so we enclose it in `(warp-abs nil ...)`.

An aside: This last example illustrates a difficulty in the design of Nyquist. To support behavioral abstraction universally, we must rely upon behaviors to ‘‘do the right thing.’’ In this case, we would like the `ramp` function to warp continuously according to the environment. But this is inefficient and unnecessary in many other cases where `ramp` and especially `pwl` are used. (`pwl` warps its breakpoints, but still interpolates linearly between them.) Also, if the default behavior of primitives is to warp in a continuous manner, this makes it difficult to build custom abstract behaviors. The final vote is not in.

4. More Examples

This chapter explores Nyquist through additional examples. The reader may wish to browse through these and move on to Chapter 5, which is a reference section describing Nyquist functions.

4.1. Stretching Sampled Sounds

This example illustrates how to stretch a sound, resampling it in the process:

```
; if a-snd is not loaded, load sound sample:
;
(if (not (boundp 'a-snd))
    (setf a-snd
       (s-read "demo-snd.nh" :srate 22050.0)))

; the SOUND operator shifts, stretches, clips and scales
; a sound according to the current environment
;
(play (stretch 3.0 (sound a-snd)))

(defun down ()
  (seq (stretch 0.2 (sound a-snd))
       (stretch 0.3 (sound a-snd))
       (stretch 0.4 (sound a-snd))
       (stretch 0.5 (sound a-snd))
       (stretch 0.6 (sound a-snd))))

(play (down))

; that was so much fun, let's go back up:
;
(defun up ()
  (seq (stretch 0.5 (sound a-snd))
       (stretch 0.4 (sound a-snd))
       (stretch 0.3 (sound a-snd))
       (stretch 0.2 (sound a-snd))))

; and write a sequence
;
(play (seq (down) (up) (down)))
```

Notice the use of the `sound` behavior as opposed to `cue`. The `cue` behavior shifts and scales its sound according to `*warp*` and `*loud*`, but it does not change the duration or resample the sound. In contrast, `sound` not only shifts and scales its sound, but it also stretches it by resampling according to `*warp*`. If `*warp*` is a continuous warping function, then the sound will be stretched by time-varying amounts. (The `*transpose*` element of the environment is ignored by both `cue` and `sound`.)

Note: `sound` uses linear interpolation rather than a high-quality resampling algorithm. In some cases, this may introduce errors audible as noise. A future version of Nyquist will include high-quality sample interpolation as an explicit function to be called when you need it.

Notice that the overall duration of `(stretch 0.5 (sound a-snd))` will be half the duration of `a-snd`.

4.2. Saving Sound Files

So far, we have used the `play` function to play a sound. The `play` function works by writing a sound to a file and then running a system program to play the file. This can be done one step at a time, and it is often convenient to save a sound to a particular file for later use:

```
; write the sample to a file,
;    the file name can be any Unix filename.  Prepending a "./" tells
;    s-save to not prepend *default-sf-dir*
;
(s-save a-snd 1000000000 "./a-snd-file.snd")

; play a file
; (only works if you have a Unix program called "play")
(system "play a-snd-file.snd")

; delete the file (do this with care!)
;
(system "rm a-snd-file.snd")


; now let's do it using a variable as the file name
;
(setf my-sound-file "./a-snd-file.snd")

(s-save a-snd 1000000000 my-sound-file)

(system (strcat "play " my-sound-file))

(system (strcat "rm " my-sound-file))
```

This example shows how `s-save` can be used to save a sound to a file.

This example also shows how the `system` function can be used to invoke Unix shell commands, such as a command to play a file or remove it. Finally, notice that `strcat` can be used to concatenate a command name to a file name to create a complete command that is then passed to `system`. (This is convenient if the sound file name is stored in a parameter or variable.)

4.3. Memory Space and Normalization

Sound samples take up lots of memory, and often, there is not enough primary (RAM) memory to hold a complete composition. For this reason, Nyquist can compute sounds incrementally, saving the final result on disk. *However*, Nyquist can also save sounds in memory so that they can be reused efficiently. In general, if a sound is saved in a global variable, memory will be allocated as needed to save and reuse it.

The standard way to compute a sound and write it to disk is to pass an expression to the `play` command:

```
(play (my-composition))
```

Often it is nice to *normalize* sounds so that they use the full available dynamic range of 16 bits. At present, the only way to do this is to first compute the maximum value of the signal, forcing the signal to be evaluated, and then to write the signal to disk. The `s-max` function computes the maximum value. You can normalize in memory if you have enough memory; otherwise you can compute the sound twice.

The two techniques are illustrated here:

```
; normalize in memory.  First, assign the sound to a variable so
; it will be retained:
(setf mysound (sim (osc c4) (osc c5)))
; now compute the maximum value (ny:all is 1 giga-samples, you may want a
; smaller constant if you have less than 4GB of memory :-):
(setf mymax (s-max mysound NY:ALL))
(display "Computed max" mymax)
; now write out and play the sound from memory with a scale factor:
(play (scale (/ 1.0 mymax) mysound))

; if you don't have space in memory, here's how to do it:
(defun myscore () (sim (osc c4) (osc c5)))
; compute the maximum:
(setf mymax (s-max (myscore) NY:ALL))
(display "Computed max" mymax)
; now we know the max, but we don't have a the sound (it was garbage
; collected and never existed all at once in memory).  Compute the sound
; again, this time with a scale factor:
(play (scale (/ 1.0 mymax) (myscore)))
```

Another approach is to write the sound out once to figure out the maximum value (which is should be reported by s-save, but isn't yet). Then compute the file again with the proper scaling. You can also write a sound as a floating point file. This file can then be converted to 16-bit integer with the proper scaling applied. Although not implemented yet in Nyquist, some header formats can store maximum amplitudes, and some soundfile player programs can rescale floating point files on the fly, allowing normalized soundfile playback without an extra normalization pass (but at a cost of twice the disk space of 16-bit samples).

4.4. Frequency Modulation

The next example uses the Nyquist frequency modulation behavior `fmosc` to generate various sounds. The parameters to `fmosc` are:

`(fmosc` *pitch modulator table phase*`)`

but only the first two parameters are required:

```
; make a short sine tone with no frequency modulation
;
(play (fmosc c4 (pwl 0.1)))

; make a longer sine tone -- note that the duration of
;   the modulator determines the duration of the tone
;
(play (fmosc c4 (pwl 0.5)))
```

In the example above, `pwl` (for Piece-Wise Linear) is used to generate sounds that are zero for the durations of `0.1` and `0.5` seconds, respectively. In effect, we are using an FM oscillator with no modulation input, and the result is a sine tone. The duration of the modulation determines the duration of the generated tone (when the modulation signal ends, the oscillator stops).

The next example uses a more interesting modulation function, a ramp from zero to C_4, expressed in hz. More explanation of `pwl` is in order. This operation constructs a piece-wise linear function sampled at the `*control-srate*`. The first breakpoint is always at `(0, 0)`, so the first two parameters give the time and value of the second breakpoint, the second two parameters give the time and value of the third breakpoint, and so on. The last breakpoint has a value of `0`, so only the time of the last breakpoint is given. In this case, we want the ramp to end at C_4, so we cheat a bit by having the ramp return to zero ‘‘almost’’ instantaneously between times `0.5` and `0.501`.

The `pwl` behavior always expects an odd number of parameters. The resulting function is shifted and stretched linearly according to `*warp*` in the environment. Now, here is the example:

```
; make a frequency sweep of one octave; the piece-wise linear function
; sweeps from 0 to (step-to-hz c4) because, when added to the c4
; fundamental, this will double the frequency and cause an octave sweep.
;
(play (fmosc c4 (pwl 0.5 (step-to-hz c4) 0.501)))
```

The same idea can be applied to a non-sinusoidal carrier. Here, we assume that `*fm-voice*` is predefined (the next section shows how to define it):

```
; do the same thing with a non-sine table
;
(play (fmosc cs2 (pwl 0.5 (step-to-hz cs2) 0.501)
             *fm-voice* 0.0))
```

The next example shows how a function can be used to make a special frequency modulation contour. In this case the contour generates a sweep from a starting pitch to a destination pitch:

```
; make a function to give a frequency sweep, starting
; after <delay> seconds, then sweeping from <pitch-1>
; to <pitch-2> in <sweep-time> seconds and then
; holding at <pitch-2> for <hold-time> seconds.
;
(defun sweep (delay pitch-1 sweep-time pitch-2 hold-time)
  (let ((interval (- (step-to-hz pitch-2)
                     (step-to-hz pitch-1))))
    (pwl delay 0.0
         ; sweep from pitch 1 to pitch 2
         (+ delay sweep-time) interval
         ; hold until about 1 sample from the end
         (+ delay sweep-time hold-time -0.0005) interval
         ; quickly ramp to zero (pwl always does this,
         ;    so make it short)
         (+ delay sweep-time hold-time))))


; now try it out
;
(play (fmosc cs2 (sweep 0.1 cs2 0.6 gs2 0.5)
             *fm-voice* 0.0))
```

FM can be used for vibrato as well as frequency sweeps. Here, we use the `lfo` function to generate vibrato. The `lfo` operation is similar to `osc`, except it generates sounds at the `*control-srate*`, and the parameter is hz rather than a pitch:

```
(play (fmosc cs2 (scale 10.0 (lfo 6.0))
             *fm-voice* 0.0))
```

What kind of manual would this be without the obligatory fm sound? Here, a sinusoidal modulator (frequency C_4) is multiplied by a slowly increasing ramp from zero to `1000.0`.

```
(setf modulator (mult (pwl 1.0 1000.0 1.0005)
                        (osc c4)))

; make the sound
(play (fmosc c4 modulator))
```

4.5. Building a Wavetable

In Section 1.2.1, we saw how to synthesize a wavetable. A wavetable for `osc` also can be extracted from any sound. This is especially interesting if the sound is digitized from some external sound source and loaded using the `s-read` function. Recall that a table is a list consisting of a sound, the pitch of that sound, and T (meaning the sound is periodic).

In the following, a sound is first read from the file `demo-snd.nh`. Then, the `extract` function is used to extract the portion of the sound between 0.110204 and 0.13932 seconds. (These numbers might be obtained by first plotting the sound and estimating the beginning and end of a period, or by using some software to look for good zero crossings.) The result of `extract` becomes the first element of a list. The next element is the pitch (24.848422), and the last element is `T`. The list is assigned to `*fm-voice*`.

```
(if (not (boundp 'a-snd))
    (setf a-snd (s-read "demo-snd.nh" :srate 22050.0)))

(setf *fm-voice* (list
                  (extract 0.110204 0.13932 (cue a-snd))
                  24.848422
                  T))
```

The file *examples.lsp* contains an extensive example of how to locate zero-crossings, extract a period, build a waveform, and generate a tone from it. (See `ex37` through `ex40` in the file.)

4.6. Filter Examples

Nyquist provides a few simple filters. All of these filters take either real numbers or signals as parameters. If you pass a signal as a filter parameter, the filter coefficients are recomputed at the sample rate of the *control* signal. Since filter coefficients are generally expensive to compute, you may want to select filter control rates carefully. Use `control-srate-abs` (Section 5.3) to specify the default control sample rate, or use `force-srate` (Section 5.2.2) to resample a signal before passing it to a filter.

Before presenting examples, let's generate some unfiltered white noise:

```
(play (noise))
```

Now low-pass filter the noise with a 1000Hz cutoff:

```
(play (lp (noise) 1000.0))
```

The high-pass filter is the inverse of the low-pass:

```
(play (hp (noise) 1000.0))
```

Here is a low-pass filter sweep from 100Hz to 2000Hz:

```
(play (lp (noise) (pwl 0.0 100.0 1.0 2000.0 1.0)))
```

And a high-pass sweep from 50Hz to 4000Hz:

```
(play (hp (noise) (pwl 0.0 50.0 1.0 4000.0 1.0)))
```

The band-pass filter takes a center frequency and a bandwidth parameter. This example has a 500Hz center frequency with a 20Hz bandwidth. The scale factor is necessary because, due to the resonant peak of the filter, the signal amplitude exceeds 1.0:

```
(play (reson (scale 0.005 (noise)) 500.0 20.0)))
```

In the next example, the center frequency is swept from 100 to 1000Hz, using a constant 20Hz bandwidth:

```
(play (reson (scale 0.005 (noise))
             (pwl 0.0 100.0 1.0 1000.0 1.0) 20.0)))
```

5. Nyquist Functions

This chapter provides a language reference for Nyquist. Operations are categorized by functionality and abstraction level. Nyquist is implemented in two important levels: the ‘‘high level’’ supports behavioral abstraction, which means that operations like `stretch` and `at` can be applied. These functions are the ones that typical users are expected to use, and most of these functions are written in XLISP.

The ‘‘low-level’’ primitives directly operate on sounds, but know nothing of environmental variables (such as `*warp*`, etc.). The names of most of these low-level functions start with ‘‘`snd-`’’. In general, programmers should avoid any function with the ‘‘`snd-`’’ prefix. Instead, use the ‘‘high-level’’ functions, which know about the environment and react appropriately. The names of high-level functions do not have prefixes like the low-level functions.

There are certain low-level operations that apply directly to sounds (as opposed to behaviors) and are relatively ‘‘safe’’ for ordinary use. These are marked as such.

Nyquist uses both linear frequency and equal-temperament pitch numbers to specify repetition rates. Frequency is always specified in either cycles per second (hz), or pitch numbers, also referred to as ‘‘steps,’’ as in steps of the chromatic scale. Steps are floating point numbers such that 60 = Middle C, 61 = C#, 61.23 is C# plus 23 cents, etc. The mapping from pitch number to frequency is the standard exponential conversion, and fractional pitch numbers are allowed: $frequency = 440 \times 2^{(pitch - 69)/12}$. There are many predefined pitch names. By default these are tuned in equal temperament, with A4 = 440Hz, but these may be changed. (See Section 1.3).

5.1. Sounds

A sound is a primitive data type in Nyquist. Sounds can be created, passed as parameters, garbage collected, printed, and set to variables just like strings, atoms, numbers, and other data types.

5.1.1. What is a Sound?

Sounds have 5 components:

- `srate` — the sample rate of the sound.
- `samples` — the samples.
- `signal-start` — the time of the first sample.
- `signal-stop` — the time of one past the last sample.
- `logical-stop` — the time at which the sound logically ends, e.g. a sound may end at the beginning of a decay. This value defaults to `signal-stop`, but may be set to any value.

It may seem that there should be `logical-start` to indicate the logical or perceptual beginning of a sound as well as a `logical-stop` to indicate the logical ending of a sound. In practice, only `logical-stop` is needed; this attribute tells when the next sound should begin to form a sequence of sounds. In this respect, Nyquist sounds are asymmetric: it is possible to compute sequences forward in time by aligning the logical start of each sound with the `logical-stop` of the previous one, but one cannot compute ‘‘backwards’’, aligning the logical end of each sound with the logical start of its successor. The root of this asymmetry is the fact that when we invoke a behavior, we say when to start, and the result of the behavior tells us its logical duration. There is no way to invoke a behavior with a direct specification of when to stop[1].

Note: there is no way to enforce the intended ‘‘perceptual’’ interpretation of `logical-stop`. As far as Nyquist is concerned, these are just numbers to guide the alignment of sounds within various control constructs.

5.1.2. Multichannel Sounds

Multichannel sounds are represented by Lisp arrays of sounds. To create an array of sounds the XLISP `vector` function is useful. Most low-level Nyquist functions (the ones starting with `snd-`) do not operate on multichannel sounds. Most high-level functions do operate on multichannel sounds.

5.1.3. Creating Sounds

The basic operations that create sounds are described here. Remember that these are low-level functions that are not intended for normal use. Unless you are trying to understand the inner workings of Nyquist, skip to Section 5.2.

`(snd-const` *value t0 srate duration*`)`
: Returns a sound with constant *value*, starting at *t0* with the given *duration*, at the sample rate *srate*. You might want to use `pwl` (see Section 5.2.2.1) instead.

`(snd-read` *filename offset t0 format channels mode bits sr dur*`)`
: Loads a sound from a file with name *filename*. Files are assumed to consist of a header followed by frames consisting of one sample from each channel. The *format* specifies the type of header, but this information is currently ignored. Nyquist looks for a number of header formats and automatically figures out which format to read. If a header can be identified, the header is first read from the file. Then, the file pointer is advanced by the indicated *offset* (in seconds). If there is an unrecognized header, Nyquist will assume the file has no header. If the header size is a multiple of the frame size (bytes/sample * number-of-channels), you can use *offset* to skip over the header. To skip N bytes, use an *offset* of:

  `(/ (float N)` *sr* `(/` *bits* `8)` *channels*`)`

  If the header is not a multiple of the frame size, either write a header or contact the author (dannenberg@cs.cmu.edu) for assistance. Nyquist will round *offset* to the nearest sample. The resulting sound will start at time *t0*. If a header is found, the file will be interpreted according to the header information. If no header was found, *channels* tells how many channels there are, the samples are encoded according to *mode*, the sample length is *bits*, and *sr* is the sample rate. The duration to be read (in seconds) is given by *dur*. If *dur* is longer than the data in the file, then a shorter duration will be returned. If the file contains one channel, a sound is returned. If the file contains 2 or more channels, an array of sounds is returned. ***Note:*** you probably want to call `s-read` (see Section 5.5) instead of `snd-read`. Also, see Section 5.5 for information on the *mode* and *format* parameters.

`(snd-save` *expression maxlen filename format mode bits*`)`
: Evaluates the *expression*, which should result in a sound or an array of sounds, and writes the result to the given *filename*. If a multichannel sound (array) is written, the channels are up-sampled to the highest rate in any channel so that all channels have the same sample rate. The maximum number of samples written per channel is given by *maxlen*, which allows writing the initial part of a very long or infinite sound. A header is written according to *format*, samples are

---

[1]Most behaviors will stop at time 1, warped according to `*warp*` to some real time, but this is by convention and is not a direct specification.

encoded according to *mode*, using *bits* bits/sample. ***Note:*** you probably want to call `s-save` (see Section 5.5) instead. The *format* and *mode* parameters are described in Section 5.5.

`(snd-from-array` ...`)`
See page 31.

`(snd-white` *t0 sr d*`)`
Generate white noise, starting at *t0*, with sample rate *sr*, and duration *d*. You probably want to use `noise` (see Section 5.2.2.3).

`(snd-zero` *t0 srate*`)`
Creates a sound that is zero everywhere, starts at *t0*, and has sample rate *srate*. The logical stop time is immediate, i.e. also at *t0*. You probably want to use `pwl` (see Section 5.2.2.1) instead.

5.1.4. Signal Operations

This next set of functions take sounds as arguments, operate on them, and return a sound.

`(snd-add` *sound1 sound*`)`
Adds two sounds. The resulting start time is the minimum of the two parameter start times, the logical stop time is the maximum of the two parameter stop times, and the sample rate is the maximum of the two parameter sample rates. Use `sim` or `sum` instead of `snd-add` (see Section 5.4).

`(snd-compose` *f g*`)`
Compose two signals, i.e. compute *f*(*g*(*t*)), where *f* and *g* are sounds. This function is used primarily to implement time warping, but it can be used in other applications such as frequency modulation. For each sample *x* in *g*, *snd-compose* looks up the value of *f*(*x*) using linear interpolation. The resulting sample rate, start time, etc. are taken from *g*. The sound *f* is used in effect as a lookup table, but it is assumed that *g* is non-decreasing, so that *f* is accessed in time order. This allows samples of *f* to be computed and discarded incrementally. If in fact *g* decreases, the current sample of *g* is replaced by the previous one, forcing *g* into compliance with the non-decreasing restriction. See also `sref`, and `shape`). A future version will include a version of `snd-compose` that performs proper band-limited interpolation.

`(snd-copy` *sound*`)`
Makes a copy of *sound*. Since operators always make (logical) copies of their sound parameters, this function should never be needed. This function is here for debugging.

`(snd-down` *srate sound*`)`
Linear interpolation of samples down to the given sample rate *srate*, which must be lower than the sample rate of *sound*. Do not call this function. Nyquist performs sample-rate conversion automatically as needed. If you want to force a conversion, call `force-srate` (see Section 5.2.2).

`(snd-exp` *sound*`)`
Compute the exponential of each sample of *sound*. Use `s-exp` instead (see Section 5.2.2.3).

`(snd-inverse` *signal start srate*`)`
Compute the function inverse of *signal*, that is, compute *g*(*t*) such that *signal*(*g*(*t*)) = *t*. This function assumes that *signal* is non-decreasing, it uses linear interpolation, the resulting sample rate is *srate*, and the result is shifted to have a starting time of *start*. If *signal* decreases, the true inverse may be undefined, so we define `snd-inverse` operationally as follows: for each output time point *t*, scan ahead in *signal* until the value of signal exceeds *t*. Interpolate to find an exact time point *x* from *signal* and output *x* at time *t*. This function is intended for internal system use in implementing time warps.

`(snd-log` *sound*`)`
Compute the natural logorithm of each sample of *sound*. Use `s-log` instead (see Section

5.2.2.3).

`(s-max` *expression maxlen*`)`
Compute the maximum amplitude of a sound. The sound is created by evaluating *expression* (as in `s-save`). Only the first *maxlen* samples are evaluated. The *expression* is automatically quoted (`s-max` is a macro), so do not quote this parameter. If *expression* is a variable, then the *global binding* of that variable will be used. Also, since the variable retains a reference to the sound, the sound will be evaluated and left in memory. See Section 4.3 on page 20 for examples.

`(snd-max` *expression maxlen*`)`
Compute the maximum amplitude of a sound. The sound is created by evaluating *expression* (as in `snd-save`), which is therefore normally quoted by the caller. At most *maxlen* samples are computed. The result is the maximum of the absolute values of the samples. ***Notes:*** It is recommended to use `s-max` (see above) instead. If you want to find the maximum of a sound bound to a local variable and it is acceptable to save the samples in memory, then this is probably the function to call. Otherwise, use `s-max`.

`(snd-normalize` *sound*`)`
Internally, sounds are stored with a scale factor that applies to all samples of the sound. All operators that take sound arguments take this scale factor into account (although it is not always necessary to perform an actual multiply per sample), so you should never need to call this function. This function multiplies each sample of a sound by its scale factor, returning a sound that represents the same signal, but whose scale factor is 1.0.

`(snd-prod` *sound1 sound2*`)`
Computes the product of *sound1* and *sound2*. The resulting sound has its start time at the maximum of the input start times and a logical stop at the minimum logical stop of the inputs. Do not use this function. Use `mult` or `prod` instead (see Section 5.2.2). Sample rate, start time, etc. are taken from *sound*.

`(snd-pwl` *t0 sr lis*`)`
Computes a piece-wise linear function according to the breakpoints in *lis*. The starting time is *t0*, and the sample rate is *sr*. The breakpoints are passed in an XLISP list (of type `LVAL`) where the list alternates sample numbers (`FIXNUM`s, computed in samples from the beginning of the pwl function) and values (the value of the pwl function, given as a `FLONUM`). There is an implicit starting point of (0, 0). The list must contain an odd number of points, the omitted last value being implicitly zero (0). The list is assumed to be well-formed. Do not call this function. Use `pwl` instead (see Section 5.2.2.1).

`(snd-recip` *sound*`)`
Compute the reciprocal of each sample of *sound*. Use `recip` instead (see Section 5.2.2.3).

`(snd-scale` *scale sound*`)`
Scales the amplitude of *sound* by the factor *scale*. Use `scale` instead (see Section 5.2.2).

`(snd-shape` *signal table origin*`)`
A waveshaping function. This is the primitive upon which `shape` is based. The `snd-shape` function is like `shape` except that *signal* and *table* must be (single-channel) sounds. Use `shape` instead (see Section 5.2.2.2).

`(snd-up` *srate sound*`)`
Increases sample rate by linear interpolation. The *sound* is the signal to be up-sampled, and *srate* is the output sample rate. Do not call this function. Nyquist performs sample-rate conversion automatically as needed. If you want to force a conversion, call `force-srate` (see Section 5.2.2).

`(snd-xform` *sound sr time start stop scale*`)`
Makes a copy of *sound* and then alters it in the following order: (1) the start time (`snd-t0`) of the sound is shifted to *time*, (1) the sound is stretched as a result of setting the sample rate to *sr* (the start time is unchanged by this), (3) the sound is clipped from *start* to *stop*, (4) if *start* is

greater than *time*, the sound is shifted shifted by *time* - *start*, so that the start time is *time*, (5) the sound is scaled by *scale*. An empty (zero) sound at *time* will be returned if all samples are clipped. Normally, you should accomplish all this using transformations. A transformation applied to a sound has no effect, so use `cue` to create a transformable sound (see Section 5.2.1).

5.1.5. Filters

These are also ‘‘Signal Operators,’’ the subject of the previous section, but there are so many filter functions, they are documented in this special section.

Some filters allow time-varying filter parameters. In these functions, filter coefficients are calculated at the sample rate of the filter parameter, and coefficients are not interpolated.

`(snd-areson` *sound hz bw normalization*`)`
: A notch filter modeled after the `areson` unit generator in Csound. The `snd-areson` filter is an exact complement of `snd-reson` such that if both are applied to the same signal with the same parameters, the sum of the results yeilds the original signal. Note that because of this complementary design, the power is not normalized as in `snd-reson`. See `snd-reson` for details on *normalization*. You should use `areson` instead (see Section 5.2.2.2).

`(snd-aresoncv` *sound hz bw normalization*`)`
: This function is identical to `snd-areson` except the *bw* (bandwidth) parameter is a sound. Filter coefficients are updated at the sample rate of *bw*. The ‘‘`cv`’’ suffix stands for Constant, Variable, indicating that *hz* and *bw* are constant (a number) and variable (a sound), respectively. This naming convention is used throughout. You should use `areson` instead (see Section 5.2.2.2).

`(snd-aresonvc` *sound hz bw normalization*`)`
: This function is identical to `snd-areson` except the *hz* (center frequency) parameter is a sound. Filter coefficients are updated at the sample rate of *hz*. You should use `areson` instead (see Section 5.2.2.2).

`(snd-aresonvv` *sound hz bw normalization*`)`
: This function is identical to `snd-areson` except both *hz* (center frequency) and *bw* (bandwidth) are sounds. Filter coefficients are updated at the next sample of either *hz* or *bw*. You should use `areson` instead (see Section 5.2.2.2).

`(snd-atone` *sound hz*`)`
: A high-pass filter modeled after the `atone` unit generator in Csound. The `snd-atone` filter is an exact complement of `snd-tone` such that if both are applied to the same signal with the same parameters, the sum of the results yeilds the original signal. You should use `hp` instead (see Section 5.2.2.2).

`(snd-atonev` *sound hz*`)`
: This is just like `snd-atone` except that the *hz* cutoff frequency is a sound. Filter coefficients are updated at the sample rate of *hz*. You should use `hp` instead (see Section 5.2.2.2).

`(snd-delay` *sound delay feedback*`)`
: Feedback delay. The output, initially *sound*, is recursively delayed by *delay*, scaled by *feedback*, and added to itself, producing an repeating echo effect. The *feedback* should be less than one to avoid exponential amplitude blowup. Delay is rounded to the nearest sample. You should use `feedback-delay` instead (see Section 5.2.2.2)

`(snd-delaycv` *sound delay feedback*`)`
: Feedback delay with variable *feedback*. This is just like *snd-delay* except *feedback* is a sound. You should use `feedback-delay` instead (see Section 5.2.2.2).

`(snd-reson` *sound hz bw normalization*`)`

A second-order resonating (bandpass) filter with center frequency *hz* and bandwidth *bw*, modeled after the `reson` unit generator in Csound. The *normalization* parameter must be an integer and (like in Csound) specifies a scaling factor. A value of 1 specifies a peak amplitude response of 1.0; all frequencies other than *hz* are attenuated. A value of 2 specifies the overall RMS value of the amplitude response is 1.0; thus filtered white noise would retain the same power. A value of zero specifies no scaling. The result sample rate, start time, etc. are takend from *sound*. You should use `reson` instead (see Section 5.2.2.2).

`(snd-resoncv` *sound hz bw normalization*`)`
This function is identical to `snd-reson` except *bw* (bandwidth) is a sound. Filter coefficients are updated at the sample rate of *bw*. You should use `reson` instead (see Section 5.2.2.2).

`(snd-resonvc` *sound hz bw normalization*`)`
This function is identical to `snd-reson` except *hz* (center frequency) is a sound. Filter coefficients are updated at the sample rate of *hz*. You should use `reson` instead (see Section 5.2.2.2).

`(snd-resonvv` *sound hz bw normalization*`)`
This function is identical to `snd-reson` except botth *hz* (center frequency) and *bw* (bandwidth) are sounds. Filter coefficients are updated at the next sample from either *hz* or *bw*. You should use `reson` instead (see Section 5.2.2.2).

`(snd-tone` *sound hz*`)`
A first-order recursive low-pass filter, based on the *tone* unit generator of Csound. The *hz* parameter is the cutoff frequency, the response curve's half-power point. The result sample rate, start time, etc. are takend from *sound*. You should use `lp` instead (see Section 5.2.2.2).

`(snd-tonev` *sound hz*`)`
This function is identical to `snd-tone` except *hz* (cutoff frequency) is a sound. The filter coefficients are updated at the sample rate of *hz*. You should use `lp` instead (see Section 5.2.2.2).

5.1.6. Table-Lookup Oscillator Functions

These functions all use a sound to describe one period of a periodic waveform. In the current implementation, the sound samples are copied to an array (the waveform table) when the function is called. To make a table-lookup oscillator generate a specific pitch, we need to have several pieces of information:

- A waveform to put into the table. This comes from the *sound* parameter.
- The length (in samples) of the waveform. This is obtained by reading samples (starting at the sound's start time, not necessarily at time zero) until the physical stop time of the sound. (If you read the waveform from a file or generate it with functions like `sim` and `sine`, then the physical and logical stop times will be the same and will correspond to the duration you specified, rounded to the nearest sample.)
- The intrinsic sample rate of the waveform. This sample rate is simply the sample rate property of *sound*.
- The pitch of the waveform. This is supplied by the *step* parameter and indicates the pitch (in steps) of *sound*. You might expect that the pitch would be related to the period (length) of *sound*, but there is the interesting case that synthesis based on sampling often loops over multiple periods. This means that the fundamental frequency of a generated tone may be some multiple of the looping rate. In Nyquist, you always specify the perceived pitch of the looped *sound* if the sound is played at the *sound*'s own sample rate.
- The desired pitch. This is specified by the *hz* parameter in Hertz (cycles per second) in these low-level functions. Note that this is not necessarily the ‘‘loop’’ rate at which the table is scanned. Instead, Nyquist figures what sample rate conversion would be necessary to

‘‘transpose’’ from the *step* which specifies the original pitch of *sound* to *hz*, which gives the desired pitch. The mixed use of steps and Hertz came about because it seemed that sample tables would be tagged with steps (‘‘I sampled a middle-C’’), whereas frequency deviation in the `fmosc` function is linear, thus calling for a specification in Hertz.

- The desired sample rate. This is given by the *sr* parameter in Hertz.

Other parameters common to all of these oscillator functions are:

- *t0*, the starting time, and
- *phase*, the starting phase in degrees. Note that if the *step* parameter indicates that the table holds more than one fundamental period, then a starting phase of 360 will be different than a starting phase of 0.

`(snd-amosc` *sound step sr hz t0 am phase*`)`
An oscillator with amplitude modulation. The sound *am* specifies the amplitude and the logical stop time. The physical stop time is also that of *am*. You should use `amosc` instead (see Section 5.2.2).

`(snd-fmosc` *s step sr hz t0 fm phase*`)`
A Frequency Modulation oscillator. The sound *fm* specifies frequency deviation (in Hertz) from *hz*. You should use `fmosc` instead (see Section 5.2.2).

`(snd-osc` *s step sr hz t0 d phase*`)`
A simple table lookup oscillator with fixed frequency. The duration is *d* seconds. You should use `osc` instead (see Section 5.2.2).

`(snd-partial` *sr hz t0 env*`)`
This is a special case of `snd-amosc` that generates a sinusoid starting at phase 0. The *env* parameter gives the envelope or any other amplitude modulation. You should use `partial` instead (see Section 5.2.2).

`(snd-sine` *t0 hz sr d*`)`
This is a special case of `snd-osc` that always generates a sinusoid with initial phase of 0. You should use `sine` instead (see Section 5.2.2).

5.1.7. Accessing Sounds

Several functions display information concerning a sound and can be used to query the components of a sound:

`(sref` *sound time*`)`
Accesses *sound* at the point *time*. If *time* does not correspond to a sample time, then the nearest samples are linearly interpolated to form the result. To access a particular sample, either convert the sound to an array (see `snd-samples` below), or use `snd-srate` and `snd-t0` (see below) to find the sample rate and starting time, and compute a time (*t*) from the sample number (*n*): $t=(n/srate)+t0$

`(sref-inverse` *sound value*`)`
Search *sound* for the first point at which it achieves *value* and return the corresponding (linearly interpolated) time. If no inverse exists, an error is raised. This function is used by Nyquist in the implementation of time warping.

`(snd-from-array` *t0 sr array*`)`
Converts a lisp array of FLONUMs into a sound with starting time *t0* and sample rate *sr*. Safe for ordinary use. Be aware that arrays of floating-point samples use 14 bytes per sample, and an additional 4 bytes per sample are allocated by this function to create a sound type.

`(snd-extent` *sound maxsamples*`)`

Returns a list of two numbers: the starting time of *sound* and the terminate time of *sound*. Finding the terminate time requires that samples be computed. Like most Nyquist functions, this is non-destructive, so memory will be allocated to preserve the sound samples. If the sound is very long or infinite, this may exhaust all memory, so the *maxsamples* parameter specifies a limit on how many samples to compute. If this limit is reached, the terminate time will be (incorrectly) based on the sound having *maxsamples* samples. This function is safe for ordinary use.

`(snd-length` *sound maxlen*`)`
Counts the number of samples in *sound* up to the physical stop time. If the sound has more than *maxlen* samples, *maxlen* is returned. Calling this function will cause all samples of the sound to be computed and saved in memory (about 4 bytes per sample). Otherwise, this function is safe for ordinary use.

`(snd-maxsamp` *sound*`)`
Computes the maximum of the absolute value of the samples in *sound*. Calling this function will cause samples to be computed and saved in memory. (This function should have a *maxlen* parameter to allow self-defense against sounds that would exhaust available memory.) Otherwise, this function is safe for ordinary use. This function will probably be removed in a future version. See `s-max`, a replacement (page 28).

`(snd-play` *expression*`)`
Evaluates *expression* to obtain a sound or array of sounds, computes all of the samples (without retaining them in memory), and returns. If this happens faster than real time for interesting sounds, you might want to modify Nyquist to actually write the samples directly to an audio output device. Meanwhile, since this function does not save samples in memory or write them to a disk, it is useful in determining how much time is spent calculating samples. See `s-save` (Section 5.5 for saving samples to a file, and `play` (Section 5.5) to play a sound. This function is safe for ordinary use.

`(snd-print-tree` *sound*`)`
Prints an ascii representation of the internal data structures representing a sound. This is useful for debugging Nyquist. This function is safe for ordinary use.

`(snd-samples` *sound limit*`)`
Converts the samples into a lisp array. The data is taken directly from the samples, ignoring shifts. For example, if the sound starts at 3.0 seconds, the first sample will refer to time 3.0, not time 0.0. A maximum of *limit* samples is returned. This function is safe for ordinary use, but like `snd-from-array`, it requires a total of slightly over 18 bytes per sample.

`(snd-srate` *sound*`)`
Returns the sample rate of the sound. Safe for ordinary use.

`(snd-show` *sound*`)`
Print the entire (internal) structure of the sound for debugging. Safe for ordinary use.

`(snd-time` *sound*`)`
Returns the start time of the sound. This will probably go away in a future version, so use `snd-t0` instead.

`(snd-t0` *sound*`)`
Returns the time of the first sample of the sound. Note that Nyquist operators such as add always copy the sound and are allowed to shift the copy up to one half sample period in either direction to align the samples of two operands. Safe for ordinary use.

`(snd-print` *expression maxlen*`)`
Evaluates *expression* to yield a sound or an array of sounds, then prints up to *maxlen* samples to the screen (stdout). This is similar to `snd-save`, but samples appear in text on the screen instead of in binary in a file. This function is intended for debugging. Safe for ordinary use.

`(snd-set-logical-stop` *sound time*`)`

Returns a sound which is *sound*, except that the logical stop of the sound occurs at *time*. ***Note:*** do not call this function. When defining a behavior, use `set-logical-stop` or `set-logical-stop-abs` instead.

`(snd-stop-time` *sound*`)`

Returns the stop time of *sound*. Sounds can be ''clipped'' or truncated at a particular time. This function returns that time or MAX-STOP-TIME if he programmer has not specified a stop time for the sound. Safe for ordinary use.

`(soundp` *sound*`)`

Returns true iff *sound* is a SOUND. Safe for ordinary use.

`(stats)`

Prints the memory usage status. See also the XLISP `mem` function. Safe for ordinary use.

5.1.8. Sequence Support Functions

The next two functions are used to implement Nyquist's `seq` construct.

`(snd-seq` *sound closure*`)`

This function returns *sound* until the logical stop time of *sound*. Then, the XLISP *closure* is evaluated, passing it the logical stop time of *sound* as a parameter. The closure must return a sound, which is then added to *sound*. (An add is used so that *sound* can continue past its logical stop if desired.) Do not call this function. See `seq` in Section 5.4.

`(snd-multiseq` *array closure*`)`

This function is similar to `snd-seq` except the first parameter is a multichannel sound rather than a single sound. A multichannel sound is simply an XLISP array of sounds. An array of sounds is returned which is the sum of *array* and another array of sounds returned by *closure*. The *closure* is passed the logical stop time of the multichannel sound, which is the maximum logical stop time of any element of *array*. Do not call this function. See `seq` in Section 5.4.

5.1.9. Miscellaneous Functions

These are all safe and recommended for ordinary use.

`(db-to-linear` *x*`)`

Returns the conversion of *x* from decibels to linear. 0dB is converted to 1. 20dB represents a linear factor of 10. If *x* is a sound, each sample is converted and a sound is returned. If *x* is a multichannel sound, each channel is converted and a multichannel sound (array) is returned. ***Note:*** With sounds, conversion is only performed on actual samples, not on the implicit zeros before the beginning and after the termination of the sound. Sample rates, start times, etc. are taken from *x*.

`(hz-to-step` *freq*`)`

Returns a pitch number for *freq* (in hz).

`(linear-to-db` *x*`)`

Returns the conversion of *x* from linear to decibels. 1 is converted to 0. 0 is converted to -INF (a special IEEE floating point value.) A factor of 10 represents a 20dB change. If *x* is a sound, each sample is converted and a sound is returned. If *x* is a multichannel sound, each channel is converted and a multichannel sound (array) is returned. ***Note:*** With sounds, conversion is only performed on actual samples, not on the implicit zeros before the beginning and after the termination of the sound. Start times, sample rates, etc. are taken from *x*.

`(log` *x*`)`

Calculates the natural log of *x* (a FLONUM). (See `s-log` for a version that operates on signals.)

`(set-control-srate` *rate*`)`

Sets the default sampling rate for control signals to *rate* by setting `*default-control-srate*` and reinitializing the environment. Do not call this within any synthesis function (see the `control-srate-abs` transformation, Section 5.3).

`(set-sound-srate` *rate*`)`
Sets the default sampling rate for audio signals to *rate* by setting `*default-sound-srate*` and reinitializing the environment. Do not call this within any synthesis function (see the `sound-srate-abs` transformation, Section 5.3).

`(set-pitch-names)`
Initializes pitch variables (`c0`, `cs0`, `df0`, `d0`, ... `b0`, `c1`, ... `b7`). A440 (the default tuning) is represented by the step 69.0, so the variable `a4` (fourth octave A) is set to 69.0. You can change the tuning by setting `*A4-Hertz*` to a value (in Hertz) and calling `set-pitch-names` to reinitialize the pitch variables. Note that this will result in non-integer step values. It does not alter the mapping from step values to frequency. There is no built-in provision for stretched scales or non-equal temperament, although users can write or compute any desired fractional step values.

`(step-to-hz` *pitch*`)`
Returns a frequency in hz for *pitch*, a pitch number.

5.2. Behaviors

5.2.1. Using Previously Created Sounds

These behaviors take a sound and transform that sound according to the environment. These are useful when writing code to make a high-level function from a low-level function, or when cuing sounds which were previously created:

`(cue` *sound*`)`
: Applies `*loud*`, the starting time from `*warp*`, `*start*`, and `*stop*` to *sound*.

`(cue-file` *filename*`)`
: Same as `cue`, except the sound comes from the named file, which is assumed to have the current default `*sound-srate*` sample rate. ***Note:*** this function is not implemented.

`(sound` *sound*`)`
: Applies `*loud*`, `*warp*`, `*start*`, and `*stop*` to *sound*.

`(control` *sound*`)`
: This function is identical to `sound`, but by convention is used when *sound* is a control signal rather than an audio signal.

5.2.2. Sound Synthesis

These functions provide musically interesting creation behaviors that react to their environment; these are the ‘‘unit generators’’ of Nyquist:

`(env` t_1 t_2 t_4 l_1 l_2 l_3 `[`*dur*`])`
: Creates a 4-phase envelope. t_i is the duration of phase *i*, and l_i is the final level of phase *i*. t_3 is implied by the duration *dur*, and l_4 is `0.0`. If *dur* is not supplied, then `1.0` is assumed. The envelope duration is the product of *dur*, `*stretch*`, and `*sustain*`. If $t_1 + t_2$ + 2ms + t_4 is greater than the envelope duration, then a two-phase envelope is substituted that has an

attack/release time ratio of t_1/t_4. The sample rate of the returned sound is `*cntrl-srate*`. (See `pwl` for a more general piece-wise linear function generator.) The effect of time warping is to warp the starting time and ending time. The intermediate breakpoints are then computed as described above.

`(force-srate` *srate sound*`)`
Returns a sound which is up- or down-sampled to *srate*. Interpolation is linear, and no prefiltering is applied in the down-sample case, so aliasing may occur.

`(lfo` *freq* `[`*duration table phase*`])`
Just like `osc` (below) except this computes at the `*cntrl-srate*` and frequency is specified in hz. The `*transpose*` and `*sustain*` is not applied. The effect of time warping is to warp the starting and ending times. The signal itself will have a constant unwarped frequency.

`(maketable` *sound*`)`
Assumes that the samples in *sound* constitute one period of a wavetable, and returns a wavetable suitable for use as the *table* argument to the `osc` function (see below). Currently, tables are limited to 1,000,000 samples. This limit is the compile-time constant `max_table_len` set in `sound.h`.

`(build-harmonic` *n table-size*`)`
Intended for constructing wavetables, this function returns a sound of length *table-size* samples containing *n* periods of a sinusoid. These can be scaled and summed to form a waveform with the desired harmonic content. See page 2 for an example.

`(control-warp` *signal*`)`
Applies a warp function *warp-fn* to *signal* using function composition. Linear interpolation is used. *warp-fn* is a mapping from score (logical) time to real time, and *signal* is a function from score time to real values. The result is a function from real time to real values at a sample rate of `*control-srate*`. See also `sound-warp`.

`(mult` *beh_1 beh_2* `...)`
Returns the product of behaviors. The arguments may also be numbers, in which case simple multiplication is performed. If a number and sound are mixed, the `scale` function is used to scale the sound by the number. When sounds are multiplied, the resulting sample rate is the maximum sample rate of the factors.

`(prod` *beh_1 beh_2* `...)`
Same as `mult`.

`(pan` *sound where*`)`
Pans *sound* (a behavior) according to *where* (another behavior or a number). *Sound* must be monophonic. *Where* may be a monophonic sound (e.g. `(ramp)` or simply a number (e.g. `0.5`). In either case, *where* should range from 0 to 1, where 0 means pan completely left, and 1 means pan completely right. For intermediate values, the sound to each channel is scaled linearly. Presently, `pan` does not check its arguments carefully.

Returns the product of behaviors. The arguments may also be numbers, in which case simple multiplication is performed. If a number and sound are mixed, the `scale` function is used to scale the sound by the number. When sounds are multiplied, the resulting sample rate is the maximum sample rate of the factors.

`(prod` *beh_1 beh_2* `...)`
Same as `mult`.

`(scale` *scale sound*`)`
Scales the amplitude of *sound* by the factor *scale*. Identical function to `snd-scale`, except that it handles multichannel sounds. Sample rates, start times, etc. are taken from *sound*.

`(scale-db` *db sound*`)`
Scales the amplitude of *sound* by the factor *db*, expressed in decibels. Sample rates, start times,

etc. are taken from *sound*.

`(scale-srate` *sound scale*`)`
Scales the sample rate of *sound* by *scale* factor. This has the effect of linearly shrinking or stretching time (the sound is not upsampled or downsampled). This is a special case of `snd-xform` (see Section 5.1.4).

`(shift-time` *sound shift*`)`
Shift *sound* by *shift* seconds. If the sound is *f*(*t*), then the result is *f*(*t*–*shift*). See Figure 5. This is a special case of `snd-xform` (see Section 5.1.4).

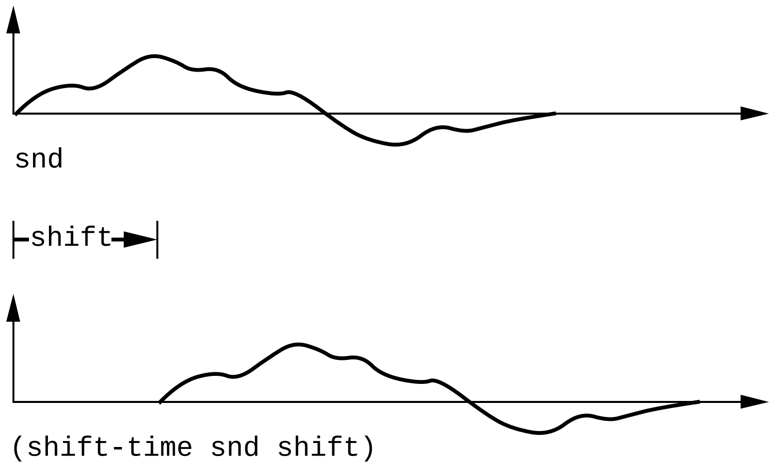

Figure 5: The `shift-time` function shifts a sound in time according to its *shift* argument.

`(sound-warp` *signal*`)`
Applies a warp function *warp-fn* to *signal* using function composition. Linear interpolation is used. *warp-fn* is a mapping from score (logical) time to real time, and *signal* is a function from score time to real values. The result is a function from real time to real values at a sample rate of `*sound-srate*`. See also `control-warp`.

`(slope` *signal*`)`
Computes the first derivative (slope) of *signal*. The start time, sample rate, etc. are taken from *signal*.

`(osc` *pitch* `[`*duration table phase*`])`
Returns a sound which is the *table* oscillated at *pitch* for the given *duration*, starting with the *phase*. Defaults are: *duration* `1.0` (second), *table* `*table*`, *phase* `0.0`. Duration is stretched by `*warp*` and `*sustain*`, amplitude is nominally 1, but scaled by `*loudness*`, the start time is logical time 0, transformed by `*warp*`, and the sample rate is `*sound-srate*`. The effect of time-warping is to warp the starting and ending times only; the signal has a constant unwarped frequency. ***Note 1:*** *table* is a list of the form

(*sound pitch-number periodic*)

where the first element is a sound, the second is the pitch of the sound (this is not redundant, because the sound may represent any number of periods), and the third element is `T` if the sound is one period of a periodic signal, or `nil` if the sound is a sample that should not be looped. The maximum table size is set by `max_table_len` in `sound.h`, and is currently set to 1,000,000.

Note 2: in the current implementation, it is assumed that the output should be periodic. See `snd-down` and `snd-up` for resampling one-shot sounds to a desired sample rate. A future version of `osc` will handle both cases.

`(partial` *pitch env*`)])`
Returns a sinusoid at the indicated pitch; the sound is multiplied by *env*. The start time and duration are taken from *env*, which is of course subject to transformations. The sample rate is `*sound-srate*`. The `partial` function is faster than `osc`.

`(sine` *pitch* `[`*duration*`])`
Returns a sinusoid at the indicated pitch. The sample rate is `*sound-srate*`. This function is like `osc` with respect to transformations. The `sine` function is faster than `osc`.

`(amosc` *pitch modulation* `[`*table phase*`])`
Returns a sound which is *table* oscillated at *pitch*. The output is multiplied by *modulation* for the duration of the sound *modulation*. *osc-table* defaults to `*table*`, and *phase* is the starting phase (default 0.0 radians) within *osc-table*. The sample rate is `*sound-srate*`.

`(fmosc` *pitch modulation* `[`*table phase*`])`
Returns a sound which is *table* oscillated at *pitch* plus *modulation* for the duration of the sound *modulation*. *osc-table* defaults to `*table*`, and *phase* is the starting phase (default 0.0 radians) within *osc-table*. The *modulation* is expressed in hz, e.g. a sinusoid modulation signal with an amplitude of 1.0 (2.0 peak to peak), will cause a +/– 1.0 hz frequency deviation in *sound*. Negative frequencies are correctly handled. The sample rate is `*sound-srate*`.

`(sampler` *pitch modulation* `[`*sample npoints*`])`
Returns a sound constructed by reading a sample from beginning to end and then splicing on copies of the same sound from a loop point to the end. ***Note:*** *this function is new and has not been tested.* The *pitch* and *modulation* parameters are used as in `fmosc` described above. The optional *sample* (which defaults to the global variable `*table*` is a list of the form

(*sound pitch-number loop-start*)

where the first element is a sound containing the sample, the second is the pitch of the sample, and the third element is the time of the loop point. The optional *npoints* specifies how many points should be used for sample interpolation. Currently this parameter defaults to 2 and only 2-point (linear) interpolation is implemented. The sample rate is `*sound-srate*`.

5.2.2.1. Piece-wise Approximations

There are a number of related behaviors for piece-wise approximations to functions. The simplest of these, `pwl` was mentioned earlier in the manual. It takes a list of breakpoints, assuming an initial point at (0, 0), and a final value of 0. An analogous piece-wise exponential function, `pwe`, is provided. Its implicit starting and stopping values are 1 rather than 0. Each of these has variants. You can specify the initial and final values (instead of taking the default). You can specify time in intervals rather than cummulative time. Finally, you can pass a list rather than an argument list. This leads to 16 versions:

- Piece-wise Linear Functions:
    - Cummulative Time:
        - Default initial point at (0, 0), final value at 0:
            - `pwl`
            - `pwl-list`
        - Explicit initial value:
            - `pwlv`
            - `pwlv-list`
    - Relative Time:
        - Default initial point at (0, 0), final value at 0:
            - `pwlr`
            - `pwlr-list`
        - Explicit initial value:
            - `pwlvr`
            - `pwlvr-list`
- Piece-wise Exponential Functions:
    - Cummulative Time:
        - Default initial point at (0, 1), final value at 1:
            - `pwe`
            - `pwe-list`
        - Explicit initial value:
            - `pwev`
            - `pwev-list`
    - Relative Time:
        - Default initial point at (0, 1), final value at 1:
            - `pwer`
            - `pwer-list`
        - Explicit initial value:
            - `pwevr`
            - `pwevr-list`

All of these functions are implemented in terms of `pwl` (see `nyquist.lsp` for the implementations. There are infinite opportunities for errors in these functions: if you leave off a data point, try to specify points in reverse order, try to create an exponential that goes to zero or negative values, or many other bad things, the behavior is not well-defined. Nyquist should not crash, but Nyquist does not necessarily attempt to report errors at this time.

`(pwl` t_1 l_1 t_2 l_2 `...` t_n`)`

Creates a piece-wise linear envelope with breakpoints at (0, 0), (t_1, l_1), (t_2, l_2), ... (t_n, 0). The breakpoint times are scaled linearly by the value of `*sustain*` (if `*sustain*` is a `SOUND`, it is evaluated once at the starting time of the envelope). Each breakpoint time is then mapped according to `*warp*`. The result is a linear interpolation (unwarped) between the breakpoints. The sample rate is `*control-srate*`. Breakpoint times are quantized to the nearest sample time. If you specify one or more breakpoints withing one sample period, `pwl` attempts to give a good approximation to the specified function. In particular, if two breakpoints are simultaneous, `pwl` will move one of them to an adjacent sample, producing a steepest possible step in the signal. The exact details of this ‘‘breakpoint munging’’ is subject to change in future versions. Please report any cases where breakpoint lists give unexpected behaviors. The author will try to apply the ‘‘principle of least surprise’’ to the design. Note that the times are relative to 0; they are not durations of each envelope segment.

`(pwl-list` *breakpoints*`)`

If you have a list of breakpoints, you can use `apply` to apply the `pwl` function to the breakpoints, but if the list is very long (hundreds or thousands of points), you might get a stack overflow because XLisp has a fixed-size argument stack. Instead, call `pwl-list`, passing one argument, the list of breakpoints.

`(pwlv` l_1 t_2 l_2 t_3 t_3 `...` t_n l_n`)`
Creates a piece-wise linear envelope with breakpoints at (0, l_1), (t_2, l_2), etc., ending with (t_n, l_n. Otherwise, the behavior is like that of `pwl`.

`(pwlv-list` *breakpoints*`)`
A version of `pwlv` that takes a single list of breakpoints as its argument. See `pwl-list` above for the rationale.

`(pwlr` i_1 l_1 i_2 l_2 `...` i_n`)`
Creates a piece-wise linear envelope with breakpoints at (0, 0), (t_1, l_1), (t_2, l_2), ... (t_n, 0), where t_j is the sum of i_1 through i_j. In other words, the breakpoint times are specified in terms of intervals rather than cummulative time. Otherwise, the behavior is like that of `pwl`.

`(pwlr-list` *breakpoints*`)`
A version of `pwlr` that takes a single list of breakpoints as its argument. See `pwl-list` above for the rationale.

`(pwlvr` l_1 i_2 l_2 i_3 i_3 `...` i_n l_n`)`
Creates a piece-wise linear envelope with breakpoints at (0, l_1), (t_2, l_2), etc., ending with (t_n, l_n, where t_j is the sum of i_2 through i_j. In other words, the breakpoint times are specified in terms of intervals rather than cummulative time. Otherwise, the behavior is like that of `pwlv`.

`(pwlvr-list` *breakpoints*`)`
A version of `pwlvr` that takes a single list of breakpoints as its argument. See `pwl-list` above for the rationale.

`(pwe` t_1 l_1 t_2 l_2 `...` t_n`)`
Creates a piece-wise exponential envelope with breakpoints at (0, 1), (t_1, l_1), (t_2, l_2), ... (t_n, 1). Exponential segments means that the ratio of values from sample to sample is constant within the segment. (The current implementation actually takes the log of each value, computes a piece-wise exponential from the points using `pwl`, then exponentiates each resulting sample. A faster implementation is certainly possible!) Breakpoint values (l_j) must be greater than zero. Otherwise, this function is similar to `pwl`, including stretch by `*sustain*`, mapping according to `*warp*`, sample rate based on `*control-srate*`, and "breakpoint munging" (see `pwl` described above). *Default initial and final values are of dubious value with exponentials. See* `pwev` *below for the function you are probably looking for.*

`(pwe-list` *breakpoints*`)`
A version of `pwe` that takes a single list of breakpoints as its argument. See `pwl-list` above for the rationale.

`(pwev` l_1 t_2 l_2 t_3 t_3 `...` t_n l_n`)`
Creates a piece-wise exponential envelope with breakpoints at (0, l_1), (t_2, l_2), etc., ending with (t_n, l_n. Otherwise, the behavior is like that of `pwe`.

`(pwev-list` *breakpoints*`)`
A version of `pwev` that takes a single list of breakpoints as its argument. See `pwl-list` above for the rationale.

`(pwer` i_1 l_1 i_2 l_2 `...` i_n`)`
Creates a piece-wise exponential envelope with breakpoints at (0, 1), (t_1, l_1), (t_2, l_2), ... (t_n, 1), where t_j is the sum of i_1 through i_j. In other words, the breakpoint times are specified in terms of intervals rather than cummulative time. Otherwise, the behavior is like that of `pwe`. Consider using `pwerv` instead of this one.

`(pwer-list` *breakpoints*`)`
A version of `pwer` that takes a single list of breakpoints as its argument. See `pwl-list` above for the rationale.

`(pwevr` l_1 i_2 l_2 i_3 i_3 `...` i_n l_n`)`

Creates a piece-wise exponential envelope with breakpoints at (0, l_1), (t_2, l_2), etc., ending with (t_n, l_n, where t_j is the sum of i_2 through i_j. In other words, the breakpoint times are specified in terms of intervals rather than cummulative time. Otherwise, the behavior is like that of `pwev`. Note that this is similar to the csound GEN05 generator. Which is uglier, *GEN05* or *pwevr*?

`(pwevr-list` *breakpoints*`)`

A version of `pwevr` that takes a single list of breakpoints as its argument. See `pwl-list` above for the rationale.

5.2.2.2. Filter Behaviors

`(comb` *sound hz decay*`)`

Applies a comb filter to *sound*. A comb filter emphasizes (resonates at) frequencies that are multiples of a *hz*. The decay time of the resonance is given by *decay*. This is a variation on `feedback-delay` (see below). The *hz* parameter must be a number greater than zero. It is used to compute delay, which is then rounded to the nearest integer number of samples (so the frequency is not always exact. Higher sampling rates yield better delay resolution.) The *decay* may be a sound or a number. In either case, it must also be positive. (Implementation note: an exponentiation is needed to convert *decay* into the *feedback* parameter for `feedback-delay`, and exponentiation is typically more time-consuming than the filter operation itself. To get high performance, provide *decay* at a low sample rate.) The resulting sound will have the start time, sample rate, etc. of *sound*.

`(feedback-delay` *sound delay feedback*`)`

Applies feedback delay to *sound*. The *delay* must be a number (in seconds). It is rounded to the nearest sample to determine the length of the delay. The sample rate is the maximum from *sound* and *feedback* (if feedback is also a sound). The amound of *feedback* should be less than one to avoid an exponential increase in amplitude. The start time and stop time, and logical stop time are taken from *sound*. Since output is truncated at the stop time of *sound*, you may want to append some silence to *sound* to give the filter time to decay.

`(lp` *sound cutoff*`)`

Filters *sound* using a first-order Butterworth low-pass filter. *Cutoff* may be a float or a signal (for time-varying filtering) and expresses hertz. Filter coefficients (requiring trig functions) are recomputed at the sample rate of *cutoff*. The resulting sample rate, start time, etc. are taken from *sound*.

`(tone` *sound cutoff*`)`

Same as `lp`. The name `tone` and the filter design comes from Csound.

`(hp` *sound cutoff*`)`

Filters *sound* using a first-order Butterworth high-pass filter. *Cutoff* may be a float or a signal (for time-varying filtering) and expresses hertz. Filter coefficients (requiring trig functions) are recomputed at the sample rate of *cutoff*. This filter is an exact complement of `lp`.

`(atone` *sound cutoff*`)`

Same as `hp`. The name `atone` and the filter design comes from Csound.

`(reson` *sound center bandwidth n*`)`

Apply a resonating filter to *sound* with center frequency *center* (in hertz), which may be a float or a signal. *Bandwidth* is the filter bandwidth (in hertz), which may also be a signal. Filter coefficients (requiring trig functions) are recomputed at each new sample of either *center* or *bandwidth*, and coefficients are *not* interpolated. The last parameter *n* specifies the type of normalization as in Csound: A value of 1 specifies a peak amplitude response of 1.0; all frequencies other than *hz* are attenuated. A value of 2 specifies the overall RMS value of the amplitude response is 1.0; thus filtered white noise would retain the same power. A value of zero specifies no scaling. The resulting sample rate, start time, etc. are taken from *sound*.

`(areson` *sound center bandwidth n*`)`

The `areson` filter is an exact complement of `reson` such that if both are applied to the same signal with the same parameters, the sum of the results yeilds the original signal.

`(shape` *signal table origin*`)`
A waveshaping function. Use *table* as a function; apply the function to each sample of *signal* to yield a new sound. *Signal* should range from -1 to +1. Anything beyond these bounds is clipped. *Table* is also a sound, but it is converted into a lookup table (similar to table-lookup oscillators). The *origin* is a FLONUM and gives the time which should be considered the origin of *table*. (This is important because *table* cannot have values at negative times, but *signal* will often have negative values. The *origin* gives an offset so that you can produce suitable tables.) The output at time *t* is:

$table(origin + \mathrm{clip}(signal(t))$

where $\mathrm{clip}(x) = max(1, min(-1, x))$. (E.g. if *table* is a signal defined over the interval [0, 2], then *origin* should be 1.0. The value of *table* at time 1.0 will be output when the input signal is zero.) The output has the same start time, sample rate, etc. as *signal*. The `shape` function will also accept multichannel *signal*s and *table*s.

5.2.2.3. More Behaviors

`(s-exp` *sound*`)`
A generalized exponential function. If *sound* is a SOUND, compute e^x for each sample *x*. If *sound* is a number *x*, just compute e^x. If *sound* is a multichannel sound, return a multichannel sound with `s-exp` applied to each element. The result has the type, sample rate, starting time, etc. of *sound*.

`(s-log` *sound*`)`
A generalized natural log function. If *sound* is a SOUND, compute $ln(x)$ for each sample *x*. If *sound* is a number *x*, just compute $ln(x)$. If *sound* is a multichannel sound, return a multichannel sound with `s-log` applied to each element. The result has the type, sample rate, starting time, etc. of *sound*. Note that the *ln* of 0 is undefined (some implementations return negative infinity), so use this function with care.

`(osc-note` *pitch* `[`*duration env loud table*`])`
Same as `osc`, but `osc-note` multiplies the result by *env*. The *env* may be a sound, or a list supplying (t_1 t_2 t_4 l_1 l_2 l_3). The result has a sample rate of `*sound-srate*`.

`(ramp [`*duration*`])`
Returns a linear ramp from 0 to 1 over *duration* (default is 1). The function actually reaches 1 at *duration*, and therefore has one extra sample, making the total duration be *duration* + 1/`*control-srate*`. See Figure 6 for more detail. Ramp is unaffected by the `sustain` transformation. The effect of time warping is to warp the starting and ending times only. The ramp itself is unwarped (linear). The sample rate is `*control-srate*`.

`(recip` *sound*`)`
A generalized reciprocal function. If *sound* is a SOUND, compute $1/x$ for each sample *x*. If *sound* is a number *x*, just compute $1/x$. If *sound* is a multichannel sound, return a multichannel sound with `recip` applied to each element. The result has the type, sample rate, starting time, etc. of *sound*. Note that the reciprocal of 0 is undefined (some implementations return infinity), so use this function with care on sounds. Division of sounds is accomplished by multiplying by the reciprocal. Again, be careful not to divide by zero.

`(s-rest [`*duration*`])`
Create silence (zero samples) for the given *duration* at the sample rate `*sound-srate*`. Default duration is 1.0 sec, and the sound is transformed in time according to `*warp*`. ***Note:*** `rest` is a Lisp function that is equivalent to `cdr`. Be careful to use `s-rest` when you need a sound!

`(noise [`*duration*`])`

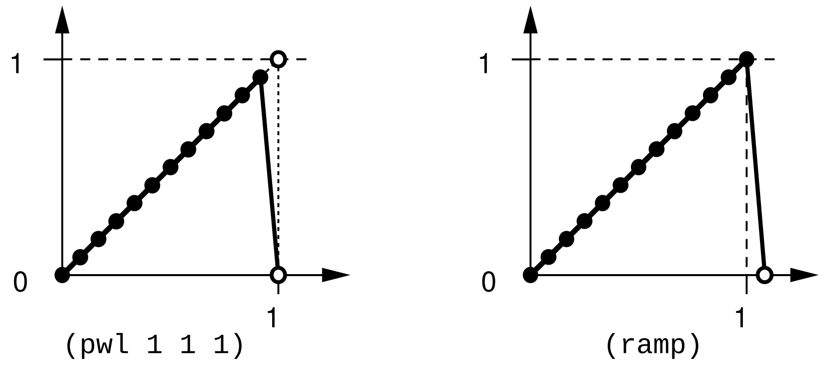

Figure 6: Ramps generated by `pwl` and `ramp` functions. The `pwl` version ramps toward the breakpoint (1, 1), but in order to ramp back to zero at breakpoint (1, 0), the function never reaches an amplitude of 1. If used at the beginning of a `seq` construct, the next sound will begin at time 1. The `ramp` version actually reaches breakpoint (1, 1); notice that it is one sample longer than the `pwl` version. If used in a sequence, the next sound after `ramp` would start at time 1 + *P*, where *P* is the sample period.

Generate noise with the given *duration*. Duration (default is 1.0) is transformed according to `*warp*`. The sample rate is `*sound-srate*` and the amplitude is +/- `*loud*`.

5.3. Transformations

These functions change the environment that is seen by other high-level functions. Note that these changes are usually relative to the current environment. There are also ‘‘absolute’’ versions of each transformation function, with the exception of `seq`, `seqrep`, `sim`, and `simrep`. The ‘‘absolute’’ versions (starting or ending with ‘‘abs’’) do not look at the current environment, but rather set an environment variable to a specific value. In this way, sections of code can be insulated from external transformations.

`(abs-env` *beh*`)`
: Compute *beh* in the default environment. This is useful for computing waveform tables and signals that are ‘‘outside’’ of time. For example, `(at 10.0 (abs-env (my-beh)))` is equivalent to `(abs-env (my-beh))` because `abs-env` forces the default environment.

`(at` *time beh*`)`
: Evaluate *beh* with `*warp*` shifted by *time*.

`(at-abs` *time beh*`)`
: Evaluate *beh* with `*warp*` shifted so that local time 0 maps to *time*.

`(continuous-control-warp` *beh*`)`
: Applies the current warp environment to the signal returned by *beh*. The result has the default control sample rate `*control-srate*`. Linear interpolation is currently used. Implementation: *beh* is first evaluated without any shifting, stretching, or warping. The result is functionally composed with the inverse of the environment's warp function.

`(continuous-sound-warp` *beh*`)`
: Applies the current warp environment to the signal returned by *beh*. The result has the default sound sample rate `*sound-srate*`. Linear interpolation is currently used. See

`continuous-control-warp` for implementation notes.

`(control-srate-abs` *srate beh*`)`
Evaluate *beh* with `*control-srate*`set to sample rate *srate*. ***Note:*** there is no ‘‘relative’’ version of this function.

`(extract` *start stop beh*`)`
Returns a sound which is the portion of *beh* between *start* and *stop*. Note that this is done relative to the current `*warp*`. The result is shifted to start according to `*warp*`, so normally the result will start without a delay of *start*.

`(extract-abs` *start stop beh*`)`
Returns a sound which is the portion of *beh* between *start* and *stop*, independent of the current `*warp*`. The result is shifted to start according to `*warp*`.

`(loud` *volume beh*`)`
Evaluates *beh* with `*loud*` incremented by *volume*. (Recall that `*loud*` is in decibels, so increment is the proper operation.)

`(loud-abs` *volume beh*`)`
Evaluates *beh* with `*loud*` set to *volume*.

`(sound-srate-abs` *srate beh*`)`
Evaluate *beh* with `*sound-srate*` set to sample rate *srate*. ***Note:*** there is no ‘‘relative’’ version of this function.

`(stretch` *factor beh*`)`
Evaluates *beh* with `*warp*` scaled by *factor*. The effect is to ‘‘stretch’’ the result of *beh* (under the current environment) by *factor*. See Chapter 3 for more information.

`(stretch-abs` *factor beh*`)`
Evaluates *beh* with `*warp*` set to a linear time transformation where each unit of logical time maps to *factor* units of real time. The effect is to stretch the nominal behavior of *beh* (under the default global environment) by *factor*. See Chapter 3 for more information.

`(transpose` *amount beh*`)`
Evaluates *beh* with `*transpose*` shifted by *amount*. The effect is relative transposition by *amount* semitones.

`(transpose-abs` *amount beh*`)`
Evaluates *beh* with `*transpose*` set to *amount*. The effect is the transposition of the nominal pitches in *beh* (under the default global environment) by *amount*.

`(warp` *fn beh*`)`
Evaluates *beh* with `*warp*` modified by *fn*. The idea is that *beh* and *fn* are written in the same time system, and *fn* warps that time system to local time. The current environment already contains a mapping from local time to global (real) time. The value of `*warp*` in effect when *beh* is evaluated is the functional composition of the initial `*warp*` with *fn*.

`(warp-abs` *fn beh*`)`
Evaluates *beh* with `*warp*` set to *fn*. In other words, the current `*warp*` is ignored and not composed with *fn* to form the new `*warp*`.

5.4. Combination and Time Structure

These behaviors combine component behaviors into structures, including sequences (melodies), simultaneous sounds (chords), and structures based on iteration.

`(seq` beh_1 `[`beh_2 `...])`
Evaluates the first behavior beh_1 according to `*time*` and each successive behavior at the `logical-stop` time of the previous one. The results are summed to form a sound whose

`logical-stop` is the `logical-stop` of the last behavior in the sequence. Each behavior can result in a multichannel sound, in which case, the logical stop time is considered to be the maximum logical stop time of any channel. The number of channels in the result is the number of channels of the first behavior. If other behaviors return fewer channels, new channels are created containing constant zero signals until the required number of channels is obtained. If other behaviors return a simple sound rather than multichannel sounds, the sound is automatically assigned to the first channel of a multichannel sound that is then filled out with zero signals. If another behavior returns more channels than the first behavior, the error is reported and the computation is stopped. Sample rates are converted up or down to match the sample rate of the first sound in a sequence.

`(seqrep (`*var limit*`)` *beh*`)`
: Iteratively evaluates *beh* with the atom *var* set with values from 0 to *limit*-1, inclusive. These sounds are then placed sequentially in time as if by `seq`.

`(sim [`beh_1 beh_2 `...])`
: Returns a sound which is the sum of the given behaviors evaluated with current value of `*warp*`. If behaviors return multiple channel sounds, the corresponding channels are added. If the number of channels does not match, the result has the maximum. For example, if a two-channel sound [L, R] is added to a four-channel sound [C1, C2, C3, C4], the result is [L + C1, R + C2, C3, C4]. Arguments to `sim` may also be numbers. If all arguments are numbers, `sim` is equivalent (although slower than) the `+` function. If a number is added to a sound, `snd-offset` is used to add the number to each sample of the sound. The result of adding a number to two or more sounds with different durations is not defined. Use `const` to coerce a number to a sound of a specified duration. See also `sum`, below.

`(simrep (`*var limit*`)` *beh*`)`
: Iteratively evaluates *beh* with the atom *var* set with values from 0 to *limit*-1, inclusive. These sounds are then placed simultaneously in time as if by `sim`.

`(set-logical-stop` *beh time*`)`
: Returns a sound with *time* as the logical stop time.

`(sum` *a* `[`*b c* `...])`
: Returns the sum of *a*, *b*, *c*, ..., allowing mixed addition of sounds, multichannel sounds and numbers. Identical to *sim*.

5.5. Sound File Input and Output

`(play` *sound*`)`
: Play the sound through the DAC. The `play` function writes a file and plays it. The details of this are system-dependent, but `play` is defined in the file `play.lsp`. The variable `*default-sfdir*` names a directory into which to save a sound file.

`(s-save` *expression maxlen filename* `[:format` *format* [:mode *mode*] [:bits *bits*])]
: Evaluates the *expression*, which should result in a sound or an array of sounds, and writes the result to the given *filename*. If a multichannel sound (array) is written, the channels are up-sampled to the highest rate in any channel so that all channels have the same sample rate. The maximum number of samples written per channel is given by *maxlen*, which allows writing the initial part of a very long or infinite sound. A header is written according to *format*, samples are encoded according to *mode*, using *bits* bits/sample. Defaults for these are `*default-sf-format*`, `*default-sf-mode*`, and `*default-sf-bits*`. The *bits* parameter may be 8, 16, or 32. The values for the *format* and *mode* options are:

Format

`snd-head-none`	No header.

`snd-head-AIFF`	AIFF format header.
`snd-head-IRCAM`	IRCAM format header.
`snd-head-NeXT`	1024-byte NeXT/SUN format header followed by IRCAM header ala CMIX. Note that the NeXT/SUN format has a header-length field, so it really is legal to have a large header, even though the normal minimal header is only 24 bytes. The additional space leaves room for maximum amplitudes, which can be used for normalizing floating-point soundfiles, and for other data. Nyquist follows the CMIX convention of placing an IRCAM format header immediately after the NeXT-style header.
`snd-head-Wave`	Microsoft Wave format header.

Mode

`snd-head-mode-adpcm`	ADPCM mode (not supported).
`snd-head-mode-pcm`	signed binary PCM mode.
`snd-head-mode-ulaw`	8-bit U-Law mode.
`snd-head-mode-alaw`	8-bit A-Law mode (not supported).
`snd-head-mode-float`	32-bit floating point mode.
`snd-head-mode-upcm`	unsigned binary PCM mode.

The defaults for format, mode, and bits are as follows:

NeXT and Sun machines:	`snd-head-NeXT`, `snd-head-mode-pcm`, `16`
SGI machines:	`snd-head-AIFF`, `snd-head-mode-pcm`, `16`

`(s-read` *filename* `[:time-offset` *offset*`] [:srate` *sr*`] [:dur` *dur*`] [:nchans` *chans*`] [:format` *format*`] [:mode` *mode*`] [:bits-per-sample` *n*`])`

Reads a sound from a file. If a header is detected, the header is used to determine the format of the file, and header information overrides format information provided by keywords (except for `:time-offset` and `:dur`).

```
(s-read "mysound.snd" :srate 44100)
```

specifies a sample rate of 44100 hz, but if the file has a header specifying 22050 hz, the resulting sample rate will be 22050. The parameters are:

- `:time-offset` — the amount of time (in seconds) to skip from the beginning of the file. The default is 0.0.
- `:srate` — the sample rate of the samples in the file. Default is `*default-sf-srate*` , which is normally 44100.
- `:dur` — the maximum duration in seconds to read. Default is 10000.
- `:nchans` — the number of channels to read. It is assumed that samples from each channel are interleaved. Default is 1.
- `:format` — the header format. See `s-save` for details. Default is `*default-sf-format*`, although this parameter is currently ignored.
- `:mode` — the sample representation, e.g. PCM or float. See `s-save` for details. Default is `*default-sf-format*`.
- `:bits` — the number of bits per sample. See `s-save` for details. Default is `*default-sf-bits*`.

(soundfilename *name*)
Converts a string *name* to a soundfile name. If *name* begins with "." or "/", the name is returned without alteration. Otherwise, a path taken from *default-sf-dir* is prepended to *name*. The s-plot, s-read, and s-save functions all use soundfilename translate filenames.

(s-plot *sound n*)
Plots sound in a window. The current implementation is minimal; it simply writes time/value pairs in ascii to a file named points.dat. To plot the points, I keep a separate X-windows program running xterm with Tektronix emulation enabled. I then type pl, which is an alias for:

```
graph < points.dat | plot -Ttek
```

This runs the Unix graph program which reads the input, scales it, and adds axes and labels. The output is piped to the plot program which converts the graphics data into Tektronix commands. It's crude but works well even over a serial line. If you are interested in making a nicer plot program for either NextStep or X11, please contact the author.

(s-print-tree *sound*)
Prints an ascii representation of the internal data structures representing a sound. This is useful for debugging Nyquist. Identical to snd-print-tree.

5.6. Miscellaneous Functions

(get-loud)
Gets the current value of the *loud* environment variable. If *loud* is a signal, it is evaluated at local time 0 and a number (FLONUM) is returned.

(get-sustain)
Gets the current value of the *sustain* environment variable. If *sustain* is a signal, it is evaluated at local time 0 and a number (FLONUM) is returned.

(get-transpose)
Gets the current value of the *transpose* environment variable. If *transpose* is a signal, it is evaluated at local time 0 and a number (FLONUM) is returned.

(get-warp)
Gets a function corresponding to the current value of the *warp* environment variable. For efficiency, *warp* is stored in three parts representing a shift, a scale factor, and a continuous warp function. Get-warp is used to retrieve a signal that maps logical time to real time. This signal combines the information of all three components of *warp* into a single signal. If the continuous warp function component is not present (indicating that the time warp is a simple combination of at and stretch transformations), an error is raised. This function is mainly for internal system use. In the future, get-warp will probably be reimplemented to always return a signal and never raise an error.

(local-to-global)
Converts a score (local) time to a real (global) time according to the current environment.

Appendix I
Extending Nyquist

WARNING: Nyquist sound functions look almost like a human wrote them; they even have a fair number of comments for human readers. Don't be fooled: virtually all Nyquist functions are written by a special translator. If you try to write a new function by hand, you will probably not succeed, and even if you do, you will waste a great deal of time. (End of Warning.)

The translator code used to extend Nyquist resides in the `trnsrc` directory. This directory also contains a special `init.lsp`, so if you start XLisp or Nyquist in this directory, it will automatically read `init.lsp`, which in turn will load the translator code (which resides in several files).

Also in the `trnsrc` directory are a number of `.alg` files, which contain the source code for the translator (more on these will follow), and a number of corresponding `.h` and `.c` files. Generated code is written to the `trnsrc` directory rather than the main Nyquist source directory so that if something goes wrong during translation, you will not lose the previous version.

Since generated code resides in the `trnsrc` directory, you will have to copy it to your main Nyquist source directory. On Unix systems, use the file Makesrc (e.g. `make -f Makesrc`) in the main Nyquist directory to pull in new code from `trnsrc`.

To translate a `.alg` file to `.c` and `.h` files, you start XLisp or Nyquist in the `trnsrc` directory and type

```
(translate "prod")
```

where `"prod"` should really be replaced by the filename (without a suffix) you want to translate. Be sure you have a saved, working copy of Nyquist or Xlisp before you recompile!

After generating `prod.c` and `prod.h`, you need to recompile Nyquist. For Unix systems, you will want to generate a new Makefile. Modify `transfiles.lsp` in your main Nyquist directory, run Xlisp or Nyquist and load `makefile.lsp`. Follow the instructions to set your machine type, etc., and execute `(makesrc)` and `(makefile)`.

Now, copy the newly generated files to your Nyquist directory (perhaps by running `make -f Makesrc`, and run `make`.

The new Lisp function will generally be named with a `snd-` prefix, e.g. `snd-prod`. You can test this by running Nyquist. Debugging is usually a combination of calling the code from within the interpreter, reading the generated code when things go wrong, and using a C debugger to step through the inner loop of the generated code. An approach I like is to set the default sample rate to 10 hertz. Then, a one-second sound has only 10 samples, which are easy to print and study on a text console.

For some functions, you must write some Lisp code to impose ordinary Nyquist behaviors such as stretching and time shifting. A good approach is to find some structurally similar functions and see how they are implemented. Most of the Lisp code for Nyquist is in `nyquist.lsp`.

Finally, do not forget to write up some documentation. Also, contributions are welcome. Send your `.alg` file, documentation, Lisp support functions for `nyquist.lsp`, and examples or test programs to `rbd@cs.cmu.edu`. I will either put them in the next release or make them available at a public ftp site.

I.1. Why Translation?

Many of the Nyquist signal processing operations are similar in form, but they differ in details. This code is complicated by many factors: Nyquist uses lazy evaluation, so the operator must check to see that input samples are available before trying to access them. Nyquist signals can have different sample rates, different block sizes, different block boundaries, and different start times, all of which must be taken into account. The number of software tests is enormous. (This may sound like a lot of overhead, but the overhead is amortized over many iterations of the inner loop. Of course setting up the inner loop to run efficiently is one more programming task.)

The main idea behind the translation is that all of the checks and setup code are similar and relatively easy to generate automatically. Programmers often use macros for this sort of task, but the C macro processor is too limited for the complex translation required here. To tell the translator how to generate code, you write `.alg` files, which provide many details about the operation in a declarative style. For example, the code generator can make some optimizations if you declare that two input signals are commutative (they can be exchanged with one another). The main part of the `.alg` file is the inner loop which is the heart of the signal processing code.

I.2. Writing a .alg File

To give you some idea how functions are specified, here is the specification for `snd-prod`, which generates over 250 lines of C code:

```
(PROD-ALG
  (NAME "prod")
  (ARGUMENTS ("sound_type" "s1") ("sound_type" "s2"))
  (START (MAX s1 s2))
  (COMMUTATIVE (s1 s2))
  (INNER-LOOP "output = s1 * s2")
  (LINEAR s1 s2)
  (TERMINATE (MIN s1 s2))
  (LOGICAL-STOP (MIN s1 s2))
)
```

A `.alg` file is always of the form:

(*name*
  (*attribute value*)
  (*attribute value*)
  ...
)

There should be just one of these algorithms descriptions per file. The *name* field is arbitrary: it is a Lisp symbol whose property list is used to save the following attribute/value pairs. There are many attributes described below. For more examples, see the `.alg` files in the `trnsrc` directory.

Understanding what the attributes do is not easy, so here are three recommendations for implementors. First, if there is an existing Nyquist operator that is structurally similar to something you want to implement, make a copy of the corresponding `.alg` file and work from there. In some cases, you can merely rename the parameters and substitute a new inner loop. Second, read the generated code, especially the generated inner loop. It may not all make sense, but sometimes you can spot obvious errors and work your way back to the error in the `.alg` file. Third, if you know where something bad is generated, see if you can find where the code is generated. (The code generator files are listed in `init.lsp`.) This code is poorly written and poorly documented, but in some cases it is fairly straightforward to determine what attribute in the `.alg` file is responsible for the erroneous output.

I.3. Attributes

Here are the attributes used for code generation. Attributes and values may be specified in any order.

`(NAME "`*string*`")` specifies a base name for many identifiers. In particular, the generated filenames will be *string*`.c` and *string*`.h`, and the XLisp function generated will be `snd-`*string*.

`(ARGUMENTS` *arglist*`)`
: describes the arguments to be passed from XLisp. *Arglist* has the form: `(`*type1 name1*`) (`*type2 name2*`) ...`, where *type* and *name* are strings in double quotes, e.g. ("sound_type" "s") specifies a SOUND parameter named `s`. Note that *arglist* is not surrounded by parentheses. As seen in this example, the type names and parameter names are C identifiers. Since the parameters are passed in from XLisp, they must be chosen from a restricted set. Valid type names are: `"sound_type"`, `"rate_type"`, `"double"`, `"long"`, `"string"`, and `"LVAL"`.

`(STATE` *statelist*`)` describes additional state (variables) needed to perform the computation. A *statelist* is similar to an *arglist* (see `ARGUMENTS` above), and has the form: `(`*type1 name1 init1* `[TEMP]) (`*type2 name2 init2* `[TEMP]) ...`. The types and names are as in *arglist*, and the "inits" are double-quoted initial values. Initial values may be any C expression. State is initialized in the order implied by *statelist* when the operation is first called from XLisp. If `TEMP` is omitted the state is preserved in a structure until the sound computation completes. Otherwise, the state variable only exists at state initialization time.

`(INNER-LOOP` *innerloop-code*`)`
: describes the inner loop, written as C code. The *innerloop-code* is in double quotes, and may extend over multiple lines. To make generated code extra-beautiful, prefix each line of *innerloop-code* with 12 spaces. Temporary variables should not be declared at the beginning of *innerloop-code*. Use the `INNER-LOOP-LOCALS` attribute instead. Within *innerloop-code, each ARGUMENT of type sound_type* ***must*** *be referenced exactly one time.* If you need to use a signal value twice, assign it once to a temporary and use the temporary twice. The inner loop must also assign *one* time to the psuedo-variable *output*. The model here is that the name of a sound argument denotes the value of the corresponding signal at the current output sample time. The inner loop code will be called once for each output sample. In practice, the code generator will substitute some expression for each signal name. For example, `prod.alg` specifies

```
(INNER-LOOP "output = s1 * s2")
```

(`s1` and `s2` are `ARGUMENTS`.) This expands to the following inner loop in `prod.c`:

```
*out_ptr_reg++ = *s1_ptr_reg++ * *s2_ptr_reg++;
```

In cases where arguments have different sample rates, sample interpolation is in-lined, and the expressions can get very complex. The translator is currently very simple-minded about substituting access code in the place of parameter names, and *this is a frequent source of bugs.* Simple string substitution is performed, so *you must not use a parameter or state name that is a substring of another*. For example, if two sound parameters were named `s` and `s2`, the translator might substitute for ‘‘s’’ in two places rather than one. If this problem occurs, you will almost certainly get a C compiler syntax error. The fix is to use ‘‘more unique’’ parameter and state variable names.

`(INNER-LOOP-LOCALS "`*innerloop-code*`")`
: The *innerloop-code* contains C declarations of local variables set and referenced in

the inner loop.

`(SAMPLE-RATE "`*expr*`")`
specifies the output sample rate; *expr* can be any C expression, including a parameter from the `ARGUMENTS` list. You can also write `(SAMPLE-RATE (MAX` *name1 name2* `...))` where names are unquoted names of arguments.

`(SUPPORT-HEADER "`*c-code*`")`
specifies arbitrary C code to be inserted in the generated `.h` file. The code typically contains auxiliarly function declarations and definitions of constants.

`(SUPPORT-FUNCTIONS "`*c-code*`")`
specifies arbitrary C code to be inserted in the generated `.c` file. The code typically contains auxiliarly functions and definitions of constants.

`(FINALIZATION "`*c-code*`")`
specifies code to execute when the sound has been fully computed and the state variables are about to be decallocated. This is the place to deallocate buffer memory, etc.

`(CONSTANT "`*name1*`" "`*name2*`" ...)`
specifies state variables that do not change value in the inner loop. The values of state variables are loaded into registers before entering the inner loop so that access will be fast within the loop. On exiting the inner loop, the final register values are preserved in a ‘‘suspension’’ structure. If state values do not change in the inner loop, this `CONSTANT` declaration can eliminate the overhead of storing these registers.

`(START` *spec*`)`
specifies when the output sound should start (a sound is zero and no processing is done before the start time). The *spec* can take several forms: `(MIN` *name1 name2* `...)` means the start time is the minimum of the start times of input signals *name1*, *name2*, ... Note that these names are not quoted.

`(TERMINATE` *spec*`)`
specifies when the output sound terminates (a sound is zero after this termination time and no more samples are computed). The *spec* can take several forms: `(MIN` *name1 name2* `...)` means the terminate time is the minimum of the terminate times of input arguments *name1*, *name2*, ... Note that these names are not quoted. To terminate at the time of a single argument `s1`, specify `(MIN s1)`. To terminate after a specific duration, use `(AFTER "`*c-expr*`")`, where *c-expr* is a C variable or expression. To terminate at a particular time, use `(AT "`*c-expr*`")`. *spec* may also be `COMPUTED`, which means to use the maximum sample rate of any input signal.

`(LOGICAL-STOP` *spec*`)`
specifies the logical stop time of the output sound. This *spec* is just like the one for `TERMINATE`. If no `LOGICAL-STOP` attribute is present, the logical stop will coincide with the terminate time.

`(ALWAYS-SCALE` *name1 name2* `...)`
says that the named sound arguments (not in quotes) should always be multiplied by a scale factor. This is a space-time tradeoff. When Nyquist sounds are scaled, the scale factor is merely stored in a structure. It is the responsibility of the user of the samples to actually scale them (unless the scale factor is exactly 1.0). The default is to generate code with and without scaling and to select the appropriate code at run time. If there are N signal inputs, this will generate 2^N versions of the code. To avoid this code explosion, use the `ALWAYS-SCALE` attribute.

`(INLINE-INTERPOLATION T)`
specifies that sample rate interpolation should be performed in-line in the inner loop. There are two forms of sample rate interpolation. One is intended for use when the

rate change is large and many points will be interpolated. This form uses a divide instruction and some setup at the low sample rate, but the inner loop overhead is just an add. The other form, intended for less drastic sample rate changes, performs interpolation with 2 multiplies and several adds per sample at the high sample rate. Nyquist generates various inner loops and selects the appropriate one at run-time. If `INLINE-INTERPOLATION` is not set, then much less code is generated and interpolation is performed as necessary by instantiating a separate signal processing operation.

`(STEP-FUNCTION` *name1 name2* `...)`
: Normally all argument signals are linearly interpolated to the output sample rate. The linear interpolation can be turned off with this attribute. This is used, for example, in Nyquist variable filters so that filter coefficients are computed at low sample rates. In fact, this attribute was added for the special case of filters.

`(DEPENDS` *spec1 spec2* `...)`
: Specifies dependencies. This attribute was also introduced to handle the case of filter coefficients (but may have other applications.) Use it when a state variable is a function of a potentially low-sample-rate input where the input is in the `STEP-FUNCTION` list. Consider a filter coefficient that depends upon an input signal such as bandwidth. In this case, you want to compute the filter coefficient only when the input signal changes rather than every output sample, since output may occur at a much higher sample rate. A *spec* is of the form

`("`*name*`" "`*arg*`" "`*expr*`" [TEMP` *"type"*`])`

which is interpreted as follows: *name* depends upon *arg*; when *arg* changes, recompute *expr* and assign it to *name*. The *name* must be declared as a `STATE` variable unless `TEMP` is present, in which case *name* is not preserved and is used only to compute other state. Variables are updated in the order of the `DEPENDS` list.

`(FORCE-INTO-REGISTER` *name1 name2* `...)`
: causes *name1*, *name2*, ... to be loaded into registers before entering the inner loop. If the inner loop references a state variable or argument, this happens automatically. Use this attribute only if references are ‘‘hidden’’ in a `#define`’d macro or referenced in a `DEPENDS` specification.

`(NOT-REGISTER` *name1 name2* `...)`
: specifies state and arguments that should not be loaded into registers before entering an inner loop. This is sometimes an optimization for infrequently accessed state.

`(NOT-IN-INNER-LOOP "`*name1*`" "`*name2*`" ...)`
: says that certain arguments are not used in the inner loop. Nyquist assumes all arguments are used in the inner loop, so specify them here if not. For example, tables are passed into functions as sounds, but these sounds are not read sample-by-sample in the inner loop, so they should be listed here.

`(MAINTAIN ("`*name1*`" "`*expr1*`") ("`*name2*`" "`*expr2*`") ... )`
: Sometimes the IBM XLC compiler generates better loop code if a variable referenced in the loop is not referenced outside of the loop after the loop exit. Technically, optimization is better when variables are dead upon loop exit. Sometimes, there is an efficient way to compute the final value of a state variable without actually referencing it, in which case the variable and the computation method are given as a pair in the `MAINTAIN` attribute. This suppresses a store of the value of the named variable, making it a dead variable. Where the store would have been, the expression is computed and assigned to the named variable. See `partial.alg` for an example. This optimization is never necessary and is only for fine-tuning.

`(LINEAR` *name1 name2* `...)`

specifies that named arguments (without quotes) are linear with respect to the output. What this *really* means is that it is numerically OK to eliminate a scale factor from the named argument and store it in the output sound descriptor, avoiding a potential multiply in this inner loop. For example, both arguments to `snd-prod` (signal multiplication) are ‘‘linear.’’ The inner loop has a single multiplication operator to multiply samples vs. a potential 3 multiplies if each sample were also scaled. To handle scale factors on the input signals, the scale factors are automatically multiplied and the product becomes the scale factor of the resulting output. (This effectively ‘‘passes the buck’’ to some other, or perhaps more than one, signal processing function, which is not always optimal. On the other hand, it works great if you multiply a number of scaled signals together: all the scale factors are ultimately handled with a single multiply.)

`(INTERNAL-SCALING` *name1 name2* `...)`
indicates that scaling is handled in code that is hidden from the code generator for *name1*, *name2*, ..., which are sound arguments. Although it is the responsibility of the reader of samples to apply any given scale factor, sometimes scaling can be had for free. For example, the `snd-recip` operation computes the reciprocal of the input samples by peforming a division. The simple approach would be to specify an inner loop of `output = 1.0/s1`, where `s1` is the input. With scaling, this would generate an inner loop something like this:

```
*output++ = 1.0 / (s1_scale_factor * *s1++);
```

but a much better approach would be the following:

```
*output++ = my_scale_factor / *s1++
```

where `my_scale_factor` is initialized to `1.0 / s1->scale`. Working backward from the desired inner loop to the `.alg` inner loop specification, a first attempt might be to specify:

```
(INNER-LOOP "output = my_scale_factor / s1")
```

but this will generate the following:

```
*output++=my_scale_factor/(s1_scale_factor * *s1++);
```

Since the code generator does not know that scaling is handled elsewhere, the scaling is done twice! The solution is to put `s1` in the `INTERNAL-SCALING` list, which essentially means ‘‘I’ve already incorporated scaling into the algorithm, so suppress the multiplication by a scale factor.’’

`(COMMUTATIVE (`*name1 name2* `...))`
specifies that the results will not be affected by interchanging any of the listed arguments. When arguments are commutative, Nyquist rearranges them at run-time into decreasing order of sample rates. If interpolation is in-line, this can dramatically reduce the amount of code generated to handle all the different cases. The prime example is `prod.alg`.

`(TYPE-CHECK "`*code*`")`
specifies checking code to be inserted after argument type checking at initialization time. See `downproto.alg` for an example where a check is made to guarantee that the output sample rate is not greater than the input sample rate. Otherwise an error is raised.

I.4. Generated Names

The resulting .c file defines a number of procedures. The procedures that do actual sample computation are named something like *name_interp-spec_*FETCH, where *name* is the NAME attribute from the .alg file, and *interp-spec* is an interpolation specification composed of a string of the following letters: n, s, i, and r. One letter corresponds to each sound argument, indicating no interpolation (r), scaling only (s), ordinary linear interpolation with scaling (i), and ramp (incremental) interpolation with scaling (r). The code generator determines all the combinations of n, s, i, and r that are necessary and generates a separate fetch function for each.

Another function is *name*_toss_fetch, which is called when sounds are not time-aligned and some initial samples must be discarded from one or more inputs.

The function that creates a sound is snd_make_*name*. This is where state allocation and initialization takes place. The proper fetch function is selected based on the sample rates and scale factors of the sound arguments, and a sound_type is returned.

Since Nyquist is a functional language, sound operations are not normally allowed to modify their arguments through side effects, but even reading samples from a sound_type causes side effects. To hide these from the Nyquist programmer, sound_type arguments are first copied (this only copies a small structure. The samples themselves are on a shared list). The function snd_*name* performs the necessary copies and calls snd_make_*name*. It is the snd_*name* function that is called by XLisp. The XLisp name for the function is SND-*NAME*. Notice that the underscore in C is converted to a dash in XLisp. Also, XLisp converts identifiers to upper case when they are read, so normally, you would type snd-*name* to call the function.

I.5. Scalar Arguments

If you want the option of passing either a number (scalar) or a signal as one of the arguments, you have two choices, neither of which is automated. Choice 1 is to coerce the constant into a signal from within XLisp. The naming convention would be to DEFUN a new function named *NAME* or S-*NAME* for ordinary use. The *NAME* function tests the arguments using XLisp functions such as TYPE-OF, NUMBERP, and SOUNDP. Any number is converted to a SOUND, e.g. using CONST. Then SND-*NAME* is called with all sound arguments. The disadvantage of this scheme is that scalars are expanded into a sample stream, which is slower than having a special inner loop where the scalar is simply kept in a register, avoiding loads, stores, and addressing overhead.

Choice 2 is to generate a different sound operator for each case. The naming convention here is to append a string of c's and v's, indicating constant (scalar) or variable (signal) inputs. For example, the reson operator comes in four variations: reson, resoncv, resonvc, and resonvv. The resonvc version implements a resonating filter with a variable center frequency (a sound type) and a constant bandwidth (a FLONUM). The RESON function in Nyquist is an ordinary Lisp function that checks types and calls one of SND-RESON, SND-RESONCV, SND-RESONVC, or SND-RESONVV.

Since each of these SND- functions performs further selection of implementation based on sample rates and the need for scaling, there are 25 different functions for computing RESON! So far, however, Nyquist is smaller than Common Lisp and it's about half the size of Microsoft Word. Hopefully, exponential growth in memory density will outpace linear (as a function of programming effort) growth of Nyquist.

Appendix II
Intgen

This documentation describes Intgen, a program for generating XLISP to C interfaces. Intgen works by scanning `.h` files with special comments in them. Intgen builds stubs that implement XLISP SUBR's. When the SUBR is called, arguments are type-checked and passed to the C routine declared in the `.h` file. Results are converted into the appropriate XLISP type and returned to the calling XLISP function. Intgen lets you add C functions into the XLISP environment with very little effort.

The interface generator will take as command-line input:

- the name of the `.c` file to generate (do not include the `.c` extension; e.g. write `xlexten`, not `xlexten.c`);
- a list of `.h` files.

The output is:

- a single `.c` file with one SUBR defined for each designated routine in a `.h` file.
- a `.h` file that declares each new C routine. E.g. if the `.c` file is named `xlexten.c`, this file will be named `xlextendefs.h`;
- a `.h` file that extends the SUBR table used by Xlisp. E.g. if the `.c` file is named `xlexten.c`, then this file is named `xlextenptrs.h`;
- a `.lsp` file with lisp initialization expressions copied from the `.h` files. This file is only generated if at least one initialization expression is encountered.

For example, the command line

```
intgen seint ~setypes.h access.h
```

generates the file `seint.c`, using declarations in `setypes.h` and `access.h`. Normally, the `.h` files are included by the generated file using `#include` commands. A ~ before a file means do not include the `.h` file. (This may be useful if you extend `xlisp.h`, which will be included anyway). Also generated will be `setintdefs.h` and `seintptrs.h`.

II.0.1. Extending Xlisp

Any number of `.h` files may be named on the command line to Intgen, and Intgen will make a single `.c` file with interface routines for all of the `.h` files. On the other hand, it is not necessary to put all of the extensions to Xlisp into a single interface file. For example, you can run Intgen once to build interfaces to window manager routines, and again to build interfaces to a new data type. Both interfaces can be linked into Xlisp.

To use the generated files, you must compile the `.c` files and link them with all of the standard Xlisp object files. In addition, you must edit the file `localdefs.h` to contain an `#include` for each `*defs.h` file, and edit the file `localptrs.h` to include each `*ptrs.h` file. For example, suppose you run Intgen to build `soundint.c`, `fugueint.c`, and `tableint.c`. You would then edit `localdefs.h` to contain the following:

```
#include "soundintdefs.h"
#include "fugueintdefs.h"
#include "tableintdefs.h"
```

and edit `localptrs.h` to contain:

```
#include "soundintptrs.h"
#include "fugueintptrs.h"
#include "tableintptrs.h"
```

These `localdefs.h` and `localptrs.h` files are in turn included by `xlftab.c` which is where Xlisp builds a table of SUBRs.

To summarize, building an interface requires just a few simple steps:

- Write C code to be called by Xlisp interface routines. This C code does the real work, and in most cases is completely independent of Xlisp.
- Add comments to `.h` files to tell Intgen which routines to build interfaces to, and to specify the types of the arguments.
- Run Intgen to build interface routines.
- Edit `localptrs.h` and `localdefs.h` to include generated `.h` files.
- Compile and link Xlisp, including the new C code.

II.1. Header file format

Each routine to be interfaced with Xlisp must be declared as follows:

type-name routine-name`(); /* LISP: (`*func-name* $type_1$ $type_2$ `...) */`

The comment may be on the line following the declaration, but the declaration and the comment must each be on no more than one line. The characters `LISP:` at the beginning of the comment mark routines to put in the interface. The comment also gives the type and number of arguments. The function, when accessed from lisp will be known as *func-name*, which need not bear any relationship to *routine-name*. By convention, underscores in the C *routine-name* should be converted to dashes in *func-name*, and *func-name* should be in all capitals. None of this is enforced or automated though.

Legal type_names are:

`LVAL`	returns an Xlisp datum.
`atom_type`	equivalent to `LVAL`, but the result is expected to be an atom.
`value_type`	a value as used in Dannenberg's score editor.
`event_type`	an event as used in Dannenberg's score editor.
`int`	interface will convert int to Xlisp `FIXNUM`.
`boolean`	interface will convert int to `T` or `nil`.
`float` or `double`	interface converts to `FLONUM`.
`char *` or `string` or `string_type`	interface converts to `STRING`. The result string will be copied into the XLISP heap.
void	interface will return `nil`.

It is easy to extend this list. Any unrecognized type will be coerced to an `int` and then returned as a `FIXNUM`, and a warning will be issued.

The ‘‘`*`’’ after char must be followed by *routine-name* with no intervening space.

Parameter types may be any of the following:

`FIXNUM`	C routine expects an int.

FLONUM or FLOAT	C routine expects a double.
STRING	C routine expects char *, the string is not copied.
VALUE	C routine expects a value_type. (Not applicable to Fugue.)
EVENT	C routine expects an event_type. (Not applicable to Fugue.)
ANY	C routine expects LVAL.
ATOM	C routine expects LVAL which is a lisp atom.
FILE	C routine expects FILE *.
SOUND	C routine expects a SoundPtr.

Any of these may be followed by ‘‘*’’: FIXNUM*, FLONUM*, STRING*, ANY*, FILE*, indicating C routine expects int *, double *, char **, LVAL *, or FILE ** . This is basically a mechanism for returning more than one value, *not* a mechanism for clobbering XLisp values. In this spirit, the interface copies the value (an int, double, char *, LVAL, or FILE *) to a local variable and passes the address of that variable to the C routine. On return, a list of resulting ‘‘*’’ parameters is constructed and bound to the global XLisp symbol *RSLT*. (Strings are copied.) If the C routine is void, then the result list is also returned by the corresponding XLisp function.

Note 1: this does not support C routines like strcpy that modify strings, because the C routine gets a pointer to the string in the XLisp heap. However, you can always add an intermediate routine that allocates space and then calls strcpy, or whatever.

Note 2: it follows that a new XLisp STRING will be created for each STRING* parameter.

Note 3: putting results on a (global!) symbol seems a bit unstructured, but note that one could write a multiple-value binding macro that hides this ugliness from the user if desired. In practice, I find that pulling the extra result values from *RSLT* when needed is perfectly acceptable.

For parameters that are result values only, the character ‘‘^’’ may be substituted for ‘‘*’’. In this case, the parameter is *not* to be passed in the XLisp calling site. However, the address of an initialized local variable of the given type is passed to the corresponding C function, and the resulting value is passed back through *RSLT* as ordinary result parameter as described above. The local variables are initialized to zero or NULL.

II.2. Using #define’d macros

If a comment of the form:

/* LISP: *type-name* (*routine-name-2 type-1 type-2* ...) */

appears on a line by itself and there was a #define on the previous line, then the preceding #define is treated as a C routine, e.g.

```
#define leftshift(val, count) ((val) << (count))
/* LISP: int (LOGSHIFT INT INT) */
```

will implement the LeLisp function LOGSHIFT.

The *type-name* following ‘‘LISP:’’ should have no spaces, e.g. use ANY*, not ANY *.

II.3. Lisp Include Files

Include files often define constants that we would like to have around in the Lisp world, but which are easier to initialize just by loading a text file. Therefore, a comment of the form:

```
/* LISP-SRC: (any lisp expression) */
```

will cause Intgen to open a file *name*`.lsp` and append

```
(any lisp expression)
```

to *name*`.lsp`, where *name* is the interface name passed on the command line. If none of the include files examined have comments of this form, then no *name*`.lsp` file is generated. ***Note:** the LISP-SRC comment must be on a new line.*

II.4. Example

This file was used for testing Intgen. It uses a trick (ok, it's a hack) to interface to a standard library macro (tolower). Since tolower is already defined, the macro ToLower is defined just to give Intgen a name to call. Two other routines, strlen and tough, are interfaced as well.

```
/* igtest.h -- test interface for intgen */

#define ToLower(c) tolower(c)
/* LISP: int (TOLOWER FIXNUM) */

int strlen();   /* LISP: (STRLEN STRING) */

void tough();
  /* LISP: (TOUGH FIXNUM* FLONUM* STRING ANY FIXNUM) */
```

II.5. More Details

Intgen has some compiler switches to enable/disable the use of certain types, including `VALUE` and `EVENT` types used by Dannenberg's score editing work, the `SOUND` type used by Fugue, and `DEXT` and `SEXT` types added for Dale Amon. Enabling all of these is not likely to cause problems, and the chances of an accidental use of these types getting through the compiler and linker seems very small.

Appendix III
XLISP: An Object-oriented Lisp

Version 2.0

February 6, 1988

by
David Michael Betz
127 Taylor Road
Peterborough, NH 03458

(603) 924-6936 (home)

III.1. Introduction

XLISP is an experimental programming language combining some of the features of Common Lisp with an object-oriented extension capability. It was implemented to allow experimentation with object-oriented programming on small computers.

There are currently implementations of XLISP running on the IBM- PC and clones under MS-DOS, on the Macintosh, the Atari-ST and the Amiga. It is completely written in the programming language C and is easily extended with user written built-in functions and classes. It is available in source form to non-commercial users.

Many Common Lisp functions are built into XLISP. In addition, XLISP defines the objects Object and Class as primitives. Object is the only class that has no superclass and hence is the root of the class hierarchy tree. Class is the class of which all classes are instances (it is the only object that is an instance of itself).

This document is a brief description of XLISP. It assumes some knowledge of LISP and some understanding of the concepts of object-oriented programming.

I recommend the book *Lisp* by Winston and Horn and published by Addison Wesley for learning Lisp. The first edition of this book is based on MacLisp and the second edition is based on Common Lisp. XLISP will continue to migrate towards compatibility with Common Lisp.

You will probably also need a copy of *Common Lisp: The Language* by Guy L. Steele, Jr., published by Digital Press to use as a reference for some of the Common Lisp functions that are described only briefly in this document.

III.2. A Note From The Author

If you have any problems with XLISP, feel free to contact me [me being David Betz - RBD] for help or advice. Please remember that since XLISP is available in source form in a high level language, many users [e.g. that Dannenberg fellow - RBD] have been making versions available on a variety of machines. If you call to report a problem with a specific version, I may not be able to help you if that version runs on a machine to which I don't have access. Please have the version number of the version that you are running readily accessible before calling me.

If you find a bug in XLISP, first try to fix the bug yourself using the source code provided. If you are successful in fixing the bug, send the bug report along with the fix to me. If you don't have access to a C compiler or are unable to fix a bug, please send the bug report to me and I'll try to fix it.

Any suggestions for improvements will be welcomed. Feel free to extend the language in whatever way suits your needs. However, PLEASE DO NOT RELEASE ENHANCED VERSIONS WITHOUT CHECKING WITH ME FIRST!! I would like to be the clearing house for new features added to XLISP. If you want to add features for your own personal use, go ahead. But, if you want to distribute your enhanced version, contact me first. Please remember that the goal of XLISP is to provide a language to learn and experiment with LISP and object-oriented programming on small computers. I don't want it to get so big that it requires megabytes of memory to run.

III.3. XLISP Command Loop

When XLISP is started, it first tries to load the workspace `xlisp.wks` from the current directory. If that file doesn't exist, XLISP builds an initial workspace, empty except for the built-in functions and symbols.

Then XLISP attempts to load `init.lsp` from the current directory. It then loads any files named as parameters on the command line (after appending `.lsp` to their names).

XLISP then issues the following prompt:

```
>
```

This indicates that XLISP is waiting for an expression to be typed.

When a complete expression has been entered, XLISP attempts to evaluate that expression. If the expression evaluates successfully, XLISP prints the result and then returns to the initial prompt waiting for another expression to be typed.

III.4. Break Command Loop

When XLISP encounters an error while evaluating an expression, it attempts to handle the error in the following way:

If the symbol `*breakenable*` is true, the message corresponding to the error is printed. If the error is correctable, the correction message is printed.

If the symbol `*tracenable*` is true, a trace back is printed. The number of entries printed depends on the value of the symbol `*tracelimit*`. If this symbol is set to something other than a number, the entire trace back stack is printed.

XLISP then enters a read/eval/print loop to allow the user to examine the state of the interpreter in the context of the error. This loop differs from the normal top-level read/eval/print loop in that if the user invokes the function `continue`, XLISP will continue from a correctable error. If the user invokes the function `clean-up`, XLISP will abort the break loop and return to the top level or the next lower numbered break loop. When in a break loop, XLISP prefixes the break level to the normal prompt.

If the symbol `*breakenable*` is `nil`, XLISP looks for a surrounding errset function. If one is found, XLISP examines the value of the print flag. If this flag is true, the error message is printed. In any case, XLISP causes the errset function call to return `nil`.

If there is no surrounding errset function, XLISP prints the error message and returns to the top level.

III.5. Data Types

There are several different data types available to XLISP programmers.

- lists
- symbols
- strings
- integers

- characters
- floats
- objects
- arrays
- streams
- subrs (built-in functions)
- fsubrs (special forms)
- closures (user defined functions)

III.6. The Evaluator

The process of evaluation in XLISP:

- Strings, integers, characters, floats, objects, arrays, streams, subrs, fsubrs and closures evaluate to themselves.
- Symbols act as variables and are evaluated by retrieving the value associated with their current binding.
- Lists are evaluated by examining the first element of the list and then taking one of the following actions:
    - If it is a symbol, the functional binding of the symbol is retrieved.
    - If it is a lambda expression, a closure is constructed for the function described by the lambda expression.
    - If it is a subr, fsubr or closure, it stands for itself.
    - Any other value is an error.

  Then, the value produced by the previous step is examined:
    - If it is a subr or closure, the remaining list elements are evaluated and the subr or closure is called with these evaluated expressions as arguments.
    - If it is an fsubr, the fsubr is called using the remaining list elements as arguments (unevaluated).
    - If it is a macro, the macro is expanded using the remaining list elements as arguments (unevaluated). The macro expansion is then evaluated in place of the original macro call.

III.7. Lexical Conventions

The following conventions must be followed when entering XLISP programs:

Comments in XLISP code begin with a semi-colon character and continue to the end of the line.

Symbol names in XLISP can consist of any sequence of non-blank printable characters except the following:

```
( ) ' ` , " ;
```

Uppercase and lowercase characters are not distinguished within symbol names. All lowercase characters are mapped to uppercase on input.

Integer literals consist of a sequence of digits optionally beginning with a `+` or `-`. The range of values an integer can represent is limited by the size of a C `long` on the machine on which XLISP is running.

Floating point literals consist of a sequence of digits optionally beginning with a `+` or `-` and including an embedded decimal point. The range of values a floating point number can represent is limited by the size of a C `float` (`double` on machines with 32 bit addresses) on the machine on which XLISP is running.

Literal strings are sequences of characters surrounded by double quotes. Within quoted strings the ‘‘\’’ character is used to allow non-printable characters to be included. The codes recognized are:

- `\\` means the character ‘‘\’’
- `\n` means newline
- `\t` means tab
- `\r` means return
- `\f` means form feed
- `\nnn` means the character whose octal code is nnn

III.8. Readtables

The behavior of the reader is controlled by a data structure called a *readtable*. The reader uses the symbol `*readtable*` to locate the current readtable. This table controls the interpretation of input characters. It is an array with 128 entries, one for each of the ASCII character codes. Each entry contains one of the following things:

- `NIL` — Indicating an invalid character
- `:CONSTITUENT` — Indicating a symbol constituent
- `:WHITE-SPACE` — Indicating a whitespace character
- `(:TMACRO .` *fun*`)` — Terminating readmacro
- `(:NMACRO .` *fun*`)` — Non-terminating readmacro
- `:SESCAPE` — Single escape character (’\’)
- `:MESCAPE` — Multiple escape character (’|’)

In the case of `:TMACRO` and `:NMACRO`, the *fun* component is a function. This can either be a built-in readmacro function or a lambda expression. The function should take two parameters. The first is the input stream and the second is the character that caused the invocation of the readmacro. The readmacro function should return `NIL` to indicate that the character should be treated as white space or a value consed with `NIL` to indicate that the readmacro should be treated as an occurence of the specified value. Of course, the readmacro code is free to read additional characters from the input stream.

XLISP defines several useful read macros:

- ’<*expr*> == (quote <*expr*>)
- #’<*expr*> == (function <*expr*>)
- #(<*expr*>...) == an array of the specified expressions

- #x<*hdigits*> == a hexadecimal number (0-9,A-F)
- #o<*odigits*> == an octal number (0-7)
- #b<*bdigits*> == a binary number (0-1)
- #\<*char*> == the ASCII code of the character
- #| ... |# == a comment
- #:<*symbol*> == an uninterned symbol
- ‘<*expr*> == (backquote <*expr*>)
- ,<*expr*> == (comma <*expr*>)
- ,@<*expr*> == (comma-at <*expr*>)

III.9. Lambda Lists

There are several forms in XLISP that require that a ‘‘lambda list’’ be specified. A lambda list is a definition of the arguments accepted by a function. There are four different types of arguments.

The lambda list starts with required arguments. Required arguments must be specified in every call to the function.

The required arguments are followed by the &optional arguments. Optional arguments may be provided or omitted in a call. An initialization expression may be specified to provide a default value for an &optional argument if it is omitted from a call. If no initialization expression is specified, an omitted argument is initialized to `NIL`. It is also possible to provide the name of a `supplied-p` variable that can be used to determine if a call provided a value for the argument or if the initialization expression was used. If specified, the supplied- p variable will be bound to T if a value was specified in the call and `NIL` if the default value was used.

The &optional arguments are followed by the &rest argument. The &rest argument gets bound to the remainder of the argument list after the required and &optional arguments have been removed.

The &rest argument is followed by the &key arguments. When a keyword argument is passed to a function, a pair of values appears in the argument list. The first expression in the pair should evaluate to a keyword symbol (a symbol that begins with a ‘‘`:`’’). The value of the second expression is the value of the keyword argument. Like &optional arguments, &key arguments can have initialization expressions and supplied-p variables. In addition, it is possible to specify the keyword to be used in a function call. If no keyword is specified, the keyword obtained by adding a ‘‘`:`’’ to the beginning of the keyword argument symbol is used. In other words, if the keyword argument symbol is `foo`, the keyword will be `':foo`.

The &key arguments are followed by the &aux variables. These are local variables that are bound during the evaluation of the function body. It is possible to have initialization expressions for the &aux variables.

Here is the complete syntax for lambda lists:

(*rarg*...
[&optional [*oarg* | (*oarg* [*init* [*svar*]])]...]
[&rest *rarg*]
[&key
[*karg* | ([*karg* | (*key karg*)] [*init* [*svar*]])]...
&allow-other-keys]
[&aux
[*aux* | (*aux* [*init*])]...])

where:

rarg is a required argument symbol
oarg is an &optional argument symbol
rarg is the &rest argument symbol
karg is a &key argument symbol
key is a keyword symbol
aux is an auxiliary variable symbol
init is an initialization expression
svar is a supplied-p variable symbol

III.10. Objects

Definitions:

- selector — a symbol used to select an appropriate method
- message — a selector and a list of actual arguments
- method — the code that implements a message

Since XLISP was created to provide a simple basis for experimenting with object-oriented programming, one of the primitive data types included is *object*. In XLISP, an object consists of a data structure containing a pointer to the object's class as well as an array containing the values of the object's instance variables.

Officially, there is no way to see inside an object (look at the values of its instance variables). The only way to communicate with an object is by sending it a message.

You can send a message to an object using the `send` function. This function takes the object as its first argument, the message selector as its second argument (which must be a symbol) and the message arguments as its remaining arguments.

The `send` function determines the class of the receiving object and attempts to find a method corresponding to the message selector in the set of messages defined for that class. If the message is not found in the object's class and the class has a super-class, the search continues by looking at the messages defined for the super-class. This process continues from one super-class to the next until a method for the message is found. If no method is found, an error occurs.

A message can also be sent from the body of a method by using the current object, but the method lookup starts with the object's superclass rather than its class. This allows a subclass to invoke a standard method in its parent class even though it overrides that method with its own specialized version.

When a method is found, the evaluator binds the receiving object to the symbol `self` and evaluates the method using the remaining elements of the original list as arguments to the method. These arguments are always evaluated prior to being bound to their corresponding formal arguments. The result of evaluating the method becomes the result of the expression.

III.11. The ‘‘Object’’ Class

`Object` — the top of the class hierarchy.

Messages:

`:show` — show an object’s instance variables.
returns — the object

`:class` — return the class of an object
returns — the class of the object

`:isnew` — the default object initialization routine
returns — the object

`:sendsuper` *sel args*... — send superclass a message
sel — the message selector
args — the message arguments
returns — the result of sending the message

III.12. The ‘‘Class’’ Class

`Class` — class of all object classes (including itself)

Messages:

:new — create a new instance of a class
returns — the new class object

:isnew *ivars* [*cvars* [*super*]] — initialize a new class
ivars — the list of instance variable symbols
cvars — the list of class variable symbols
super — the superclass (default is object)
returns — the new class object

:answer *msg fargs code* — add a message to a class
msg — the message symbol
fargs — the formal argument list (lambda list)
code — a list of executable expressions
returns — the object

When a new instance of a class is created by sending the message `:new` to an existing class, the message `:isnew` followed by whatever parameters were passed to the `:new` message is sent to the newly created object.

When a new class is created by sending the `:new` message to the object `Class`, an optional parameter may be specified indicating the superclass of the new class. If this parameter is omitted, the new class will be a subclass of `Object`. A class inherits all instance variables, class variables, and methods from its super-class.

III.13. Profiling

The Xlisp 2.0 release has been extended with a profiling facility, which counts how many times and where `eval` is executed. A separate count is maintained for each named function, closure, or macro, and a count indicates an `eval` in the immediately (lexically) enclosing named function, closure, or macro. Thus, the count gives an indication of the amount of time spent in a function, not counting nested function calls. The list of all functions executed is maintained on the global `*profile*` variable. These functions in turn have `*profile*` properties, which maintain the counts. The profile system merely increments counters and puts symbols on the `*profile*` list. It is up to the user to initialize data and gather results. Profiling is turned on or off with the `profile` function. Unfortunately, methods cannot be profiled with this facility.

III.14. SYMBOLS

- `self` — the current object (within a method context)
- `*obarray*` — the object hash table
- `*standard-input*` — the standard input stream
- `*standard-output*` — the standard output stream
- `*error-output*` — the error output stream
- `*trace-output*` — the trace output stream
- `*debug-io*` — the debug i/o stream
- `*breakenable*` — flag controlling entering break loop on errors
- `*tracelist*` — list of names of functions to trace
- `*tracenable*` — enable trace back printout on errors
- `*tracelimit*` — number of levels of trace back information
- `*evalhook*` — user substitute for the evaluator function
- `*applyhook*` — (not yet implemented)
- `*readtable*` — the current readtable
- `*unbound*` — indicator for unbound symbols
- `*gc-flag*` — controls the printing of gc messages
- `*gc-hook*` — function to call after garbage collection
- `*integer-format*` — format for printing integers (‘‘%d’’ or ‘‘%ld’’)
- `*float-format*` — format for printing floats (‘‘%g’’)
- `*print-case*` — symbol output case (:upcase or :downcase)

There are several symbols maintained by the read/eval/print loop. The symbols `+`, `++`, and `+++` are bound to the most recent three input expressions. The symbols `*`, `**` and `***` are bound to the most recent three results. The symbol `-` is bound to the expression currently being evaluated. It becomes the value of `+` at the end of the evaluation.

III.15. Evaluation Functions

(eval *expr*) — evaluate an xlisp expression
expr — the expression to be evaluated
returns — the result of evaluating the expression

(apply *fun args*) — apply a function to a list of arguments
fun — the function to apply (or function symbol)
args — the argument list
returns — the result of applying the function to the arguments

(funcall *fun arg*...) — call a function with arguments
fun — the function to call (or function symbol)
arg — arguments to pass to the function
returns — the result of calling the function with the arguments

(quote *expr*) — return an expression unevaluated
expr — the expression to be quoted (quoted)
returns — *expr* unevaluated

(function *expr*) — get the functional interpretation
expr — the symbol or lambda expression (quoted)
returns — the functional interpretation

(backquote *expr*) — fill in a template
expr — the template
returns — a copy of the template with comma and comma-at
expressions expanded

(lambda *args expr*...) — make a function closure
args — formal argument list (lambda list) (quoted)
expr — expressions of the function body
returns — the function closure

(get-lambda-expression *closure*) — get the lambda expression
closure — the closure
returns — the original lambda expression

(macroexpand *form*) — recursively expand macro calls
form — the form to expand
returns — the macro expansion

(macroexpand-1 *form*) — expand a macro call

form — the macro call form
returns — the macro expansion

III.16. Symbol Functions

(set *sym expr*) — set the value of a symbol
sym — the symbol being set
expr — the new value
returns — the new value

(setq [*sym expr*]...) — set the value of a symbol
sym — the symbol being set (quoted)
expr — the new value
returns — the new value

(psetq [*sym expr*]...) — parallel version of setq
sym — the symbol being set (quoted)
expr — the new value
returns — the new value

(setf [*place expr*]...) — set the value of a field
place — the field specifier (quoted):
sym — set value of a symbol
(car *expr*) — set car of a cons node
(cdr *expr*) — set cdr of a cons node
(nth *n expr*) — set nth car of a list
(aref *expr n*) — set nth element of an array
(get *sym prop*) — set value of a property
(symbol-value *sym*) — set value of a symbol
(symbol-function *sym*) — set functional value of a symbol
(symbol-plist *sym*) — set property list of a symbol
expr — the new value
returns — the new value

(defun *sym fargs expr*...) — define a function
(defmacro *sym fargs expr*...) — define a macro
sym — symbol being defined (quoted)
fargs — formal argument list (lambda list) (quoted)
expr — expressions constituting the body of the
function (quoted) returns — the function symbol

(gensym [*tag*]) — generate a symbol
tag — string or number
returns — the new symbol

(intern *pname*) — make an interned symbol

pname — the symbol's print name string
returns — the new symbol

(make-symbol *pname*) — make an uninterned symbol
pname — the symbol's print name string
returns — the new symbol

(symbol-name *sym*) — get the print name of a symbol
sym — the symbol
returns — the symbol's print name

(symbol-value *sym*) — get the value of a symbol
sym — the symbol
returns — the symbol's value

(symbol-function *sym*) — get the functional value of a symbol
sym — the symbol
returns — the symbol's functional value

(symbol-plist *sym*) — get the property list of a symbol
sym — the symbol
returns — the symbol's property list

(hash *sym n*) — compute the hash index for a symbol
sym — the symbol or string
n — the table size (integer)
returns — the hash index (integer)

III.17. Property List Functions

(get *sym prop*) — get the value of a property
sym — the symbol
prop — the property symbol
returns — the property value or `nil`

(putprop *sym val prop*) — put a property onto a property list
sym — the symbol
val — the property value
prop — the property symbol
returns — the property value

(remprop *sym prop*) — remove a property
sym — the symbol
prop — the property symbol
returns — `nil`

III.18. Array Functions

(aref *array n*) — get the nth element of an array
array — the array
n — the array index (integer)
returns — the value of the array element

(make-array *size*) — make a new array
size — the size of the new array (integer)
returns — the new array

(vector *expr*...) — make an initialized vector
expr — the vector elements
returns — the new vector

III.19. List Functions

(car *expr*) — return the car of a list node
expr — the list node
returns — the car of the list node

(cdr *expr*) — return the cdr of a list node
expr — the list node
returns — the cdr of the list node

(c*xx*r *expr*) — all c*xx*r combinations

(c*xxx*r *expr*) — all c*xxx*r combinations

(c*xxxx*r *expr*) — all c*xxxx*r combinations

(first *expr*) — a synonym for car

(second *expr*) — a synonym for cadr

(third *expr*) — a synonym for caddr

(fourth *expr*) — a synonym for cadddr

(rest *expr*) — a synonym for cdr

(cons *expr1 expr2*) — construct a new list node
expr1 — the car of the new list node
expr2 — the cdr of the new list node
returns — the new list node

(list *expr*...) — create a list of values

expr — expressions to be combined into a list
returns — the new list

(append *expr*...) — append lists
expr — lists whose elements are to be appended
returns — the new list

(reverse *expr*) — reverse a list
expr — the list to reverse
returns — a new list in the reverse order

(last *list*) — return the last list node of a list
list — the list
returns — the last list node in the list

(member *expr list* &key :test :test-not) — find an expression in a list
expr — the expression to find
list — the list to search
:test — the test function (defaults to eql)
:test-not — the test function (sense inverted)
returns — the remainder of the list starting with the expression

(assoc *expr alist* &key :test :test-not) — find an expression in an a-list
expr — the expression to find
alist — the association list
:test — the test function (defaults to eql)
:test-not — the test function (sense inverted)
returns — the alist entry or `nil`

(remove *expr list* &key :test :test-not) — remove elements from a list
expr — the element to remove
list — the list
:test — the test function (defaults to eql)
:test-not — the test function (sense inverted)
returns — copy of list with matching expressions removed

(remove-if *test list*) — remove elements that pass test
test — the test predicate
list — the list
returns — copy of list with matching elements removed

(remove-if-not *test list*) — remove elements that fail test
test — the test predicate
list — the list
returns — copy of list with non-matching elements removed

(length *expr*) — find the length of a list, vector or string

expr — the list, vector or string
returns — the length of the list, vector or string

(nth *n list*) — return the nth element of a list
n — the number of the element to return (zero origin)
list — the list
returns — the nth element or `nil` if the list isn't that long

(nthcdr *n list*) — return the nth cdr of a list
n — the number of the element to return (zero origin)
list — the list
returns — the nth cdr or `nil` if the list isn't that long

(mapc *fcn list1 list*...) — apply function to successive cars
fcn — the function or function name
listn — a list for each argument of the function
returns — the first list of arguments

(mapcar *fcn list1 list*...) — apply function to successive cars
fcn — the function or function name
listn — a list for each argument of the function
returns — a list of the values returned

(mapl *fcn list1 list*...) — apply function to successive cdrs
fcn — the function or function name
listn — a list for each argument of the function
returns — the first list of arguments

(maplist *fcn list1 list*...) — apply function to successive cdrs
fcn — the function or function name
listn — a list for each argument of the function
returns — a list of the values returned

(subst *to from expr* &key :test :test-not) — substitute expressions
to — the new expression
from — the old expression
expr — the expression in which to do the substitutions
:test — the test function (defaults to eql)
:test-not — the test function (sense inverted)
returns — the expression with substitutions

(sublis *alist expr* &key :test :test-not) — substitute with an a-list
alist — the association list
expr — the expression in which to do the substitutions
:test — the test function (defaults to eql)
:test-not — the test function (sense inverted)
returns — the expression with substitutions

III.20. Destructive List Functions

(rplaca *list expr*) — replace the car of a list node
    list — the list node
    expr — the new value for the car of the list node
    returns — the list node after updating the car

(rplacd *list expr*) — replace the cdr of a list node
    list — the list node
    expr — the new value for the cdr of the list node
    returns — the list node after updating the cdr

(nconc *list*...) — destructively concatenate lists
    list — lists to concatenate
    returns — the result of concatenating the lists

(delete *expr* &key :test :test-not) — delete elements from a list
    expr — the element to delete
    list — the list
    :test — the test function (defaults to eql)
    :test-not — the test function (sense inverted)
    returns — the list with the matching expressions deleted

(delete-if *test list*) — delete elements that pass test
    test — the test predicate
    list — the list
    returns — the list with matching elements deleted

(delete-if-not *test list*) — delete elements that fail test
    test — the test predicate
    list — the list
    returns — the list with non-matching elements deleted

(sort *list test*) — sort a list
    list — the list to sort
    test — the comparison function
    returns — the sorted list

III.21. Predicate Functions

(atom *expr*) — is this an atom?
    expr — the expression to check
    returns — `t` if the value is an atom, `nil` otherwise

(symbolp *expr*) — is this a symbol?
expr — the expression to check
returns — `t` if the expression is a symbol, `nil` otherwise

(numberp *expr*) — is this a number?
expr — the expression to check
returns — `t` if the expression is a number, `nil` otherwise

(null *expr*) — is this an empty list?
expr — the list to check
returns — `t` if the list is empty, `nil` otherwise

(not *expr*) — is this false?
expr — the expression to check
return — `t` if the value is `nil`, `nil` otherwise

(listp *expr*) — is this a list?
expr — the expression to check
returns — `t` if the value is a cons or `nil`, `nil` otherwise

(endp *list*) — is this the end of a list
list — the list
returns — `t` if the value is `nil`, `nil` otherwise

(consp *expr*) — is this a non-empty list?
expr — the expression to check
returns — `t` if the value is a cons, `nil` otherwise

(integerp *expr*) — is this an integer?
expr — the expression to check
returns — `t` if the value is an integer, `nil` otherwise

(floatp *expr*) — is this a float?
expr — the expression to check
returns — `t` if the value is a float, `nil` otherwise

(stringp *expr*) — is this a string?
expr — the expression to check
returns — `t` if the value is a string, `nil` otherwise

(characterp *expr*) — is this a character?
expr — the expression to check
returns — `t` if the value is a character, `nil` otherwise

(arrayp *expr*) — is this an array?
expr — the expression to check

returns — `t` if the value is an array, `nil` otherwise

(streamp *expr*) — is this a stream?
expr — the expression to check
returns — `t` if the value is a stream, `nil` otherwise

(objectp *expr*) — is this an object?
expr — the expression to check
returns — `t` if the value is an object, `nil` otherwise

(boundp *sym*) — is a value bound to this symbol?
sym — the symbol
returns — `t` if a value is bound to the symbol, `nil` otherwise

(fboundp *sym*) — is a functional value bound to this symbol?
sym — the symbol
returns — `t` if a functional value is bound to the symbol, `nil` otherwise

(minusp *expr*) — is this number negative?
expr — the number to test
returns — `t` if the number is negative, `nil` otherwise

(zerop *expr*) — is this number zero?
expr — the number to test
returns — `t` if the number is zero, `nil` otherwise

(plusp *expr*) — is this number positive?
expr — the number to test
returns — `t` if the number is positive, `nil` otherwise

(evenp *expr*) — is this integer even?
expr — the integer to test
returns — `t` if the integer is even, `nil` otherwise

(oddp *expr*) — is this integer odd?
expr — the integer to test
returns — `t` if the integer is odd, `nil` otherwise

(eq *expr1 expr2*) — are the expressions identical?
expr1 — the first expression
expr2 — the second expression
returns — `t` if they are equal, `nil` otherwise

(eql *expr1 expr2*) — are the expressions identical? (works with all numbers)
expr1 — the first expression

expr2 — the second expression
returns — `t` if they are equal, `nil` otherwise

(equal *expr1 expr2*) — are the expressions equal?
expr1 — the first expression
expr2 — the second expression
returns — `t` if they are equal, `nil` otherwise

III.22. Control Constructs

(cond *pair*...) — evaluate conditionally
pair — pair consisting of:
(*pred expr*...)
where:
pred — is a predicate expression
expr — evaluated if the predicate is not `nil`
returns — the value of the first expression whose predicate is not `nil`

(and *expr*...) — the logical and of a list of expressions
expr — the expressions to be anded
returns — `nil` if any expression evaluates to `nil`, otherwise the value of the last expression (evaluation of expressions stops after the first expression that evaluates to `nil`)

(or *expr*...) — the logical or of a list of expressions
expr — the expressions to be ored
returns — `nil` if all expressions evaluate to `nil`, otherwise the value of the first non-`nil` expression (evaluation of expressions stops after the first expression that does not evaluate to `nil`)

(if *texpr expr1* [*expr2*]) — evaluate expressions conditionally
texpr — the test expression
expr1 — the expression to be evaluated if texpr is non-`nil`
expr2 — the expression to be evaluated if texpr is `nil`
returns — the value of the selected expression

(when *texpr expr*...) — evaluate only when a condition is true
texpr — the test expression
expr — the expression(s) to be evaluated if texpr is non-`nil`
returns — the value of the last expression or `nil`

(unless *texpr expr*...) — evaluate only when a condition is false
texpr — the test expression
expr — the expression(s) to be evaluated if texpr is `nil`
returns — the value of the last expression or `nil`

(case *expr case*...) — select by case

expr — the selection expression
case — pair consisting of:
(*value expr*...)
where:
value — is a single expression or a list of expressions (unevaluated)
expr — are expressions to execute if the case matches
returns — the value of the last expression of the matching case

(let (*binding*...) *expr*...) — create local bindings

(let* (*binding*...) *expr*...) — let with sequential binding
binding — the variable bindings each of which is either:
1) a symbol (which is initialized to `nil`)
2) a list whose car is a symbol and whose cadr is an initialization expression
expr — the expressions to be evaluated
returns — the value of the last expression

(flet (*binding*...) *expr*...) — create local functions

(labels (*binding*...) *expr*...) — flet with recursive functions

(macrolet (*binding*...) *expr*...) — create local macros
binding — the function bindings each of which is:
(*sym fargs expr*...)
where:
sym — the function/macro name
fargs — formal argument list (lambda list)
expr — expressions constituting the body of the function/macro
expr — the expressions to be evaluated
returns — the value of the last expression

(catch *sym expr*...) — evaluate expressions and catch throws
sym — the catch tag
expr — expressions to evaluate
returns — the value of the last expression the throw expression

(throw *sym* [*expr*]) — throw to a catch
sym — the catch tag
expr — the value for the catch to return (defaults to `nil`)
returns — never returns

(unwind-protect *expr cexpr*...) — protect evaluation of an expression
expr — the expression to protect
cexpr — the cleanup expressions
returns — the value of the expression
Note: unwind-protect guarantees to execute the cleanup expressions even if a non-local exit terminates the evaluation of the protected expression

III.23. Looping Constructs

(loop *expr*...) — basic looping form
expr — the body of the loop
returns — never returns (must use non-local exit)

(do (*binding*...) (*texpr rexpr*...) *expr*...)
(do* (*binding*...) (*texpr rexpr*...) *expr*...)
binding — the variable bindings each of which is either:
1) a symbol (which is initialized to `nil`)
2) a list of the form: (*sym init* [*step*]) where:
sym — is the symbol to bind
init — is the initial value of the symbol
step — is a step expression
texpr — the termination test expression
rexpr — result expressions (the default is `nil`)
expr — the body of the loop (treated like an implicit prog)
returns — the value of the last result expression

(dolist (*sym expr* [*rexpr*]) *expr*...) — loop through a list
sym — the symbol to bind to each list element
expr — the list expression
rexpr — the result expression (the default is `nil`)
expr — the body of the loop (treated like an implicit prog)

(dotimes (*sym expr* [*rexpr*]) *expr*...) — loop from zero to n-1
sym — the symbol to bind to each value from 0 to n-1
expr — the number of times to loop
rexpr — the result expression (the default is `nil`)
expr — the body of the loop (treated like an implicit prog)

III.24. The Program Feature

(prog (*binding*...) *expr*...) — the program feature
(prog* (*binding*...) *expr*...) — prog with sequential binding
binding — the variable bindings each of which is either:
1) a symbol (which is initialized to `nil`)
2) a list whose car is a symbol and whose cadr is an initialization expression
expr — expressions to evaluate or tags (symbols)
returns — `nil` or the argument passed to the return function

(block *name expr*...) — named block
name — the block name (symbol)
expr — the block body

returns — the value of the last expression

(return [*expr*]) — cause a prog construct to return a value
expr — the value (defaults to `nil`)
returns — never returns

(return-from *name* [*value*]) — return from a named block
name — the block name (symbol)
value — the value to return (defaults to `nil`)
returns — never returns

(tagbody *expr*...) — block with labels
expr — expression(s) to evaluate or tags (symbols)
returns — `nil`

(go *sym*) — go to a tag within a tagbody or prog
sym — the tag (quoted)
returns — never returns

(progv *slist vlist expr*...) — dynamically bind symbols
slist — list of symbols
vlist — list of values to bind to the symbols
expr — expression(s) to evaluate
returns — the value of the last expression

(prog1 *expr1 expr*...) — execute expressions sequentially
expr1 — the first expression to evaluate
expr — the remaining expressions to evaluate
returns — the value of the first expression

(prog2 *expr1 expr2 expr*...) — execute expressions sequentially
expr1 — the first expression to evaluate
expr2 — the second expression to evaluate
expr — the remaining expressions to evaluate
returns — the value of the second expression

(progn *expr*...) — execute expressions sequentially
expr — the expressions to evaluate
returns — the value of the last expression (or `nil`)

III.25. Debugging and Error Handling

(trace *sym*) — add a function to the trace list
sym — the function to add (quoted)
returns — the trace list

(untrace *sym*) — remove a function from the trace list
sym — the function to remove (quoted)
returns — the trace list

(error *emsg* [*arg*]) — signal a non-correctable error
emsg — the error message string
arg — the argument expression (printed after the message)
returns — never returns

(cerror *cmsg* *emsg* [*arg*]) — signal a correctable error
cmsg — the continue message string
emsg — the error message string
arg — the argument expression (printed after the message)
returns — `nil` when continued from the break loop

(break [*bmsg* [*arg*]]) — enter a break loop
bmsg — the break message string (defaults to `**break**`)
arg — the argument expression (printed after the message)
returns — `nil` when continued from the break loop

(clean-up) — clean-up after an error
returns — never returns

(top-level) — clean-up after an error and return to the top level
returns — never returns

(continue) — continue from a correctable error
returns — never returns

(errset *expr* [*pflag*]) — trap errors
expr — the expression to execute
pflag — flag to control printing of the error message
returns — the value of the last expression consed with `nil`
or `nil` on error

(baktrace [*n*]) — print n levels of trace back information
n — the number of levels (defaults to all levels)
returns — `nil`

(evalhook *expr* *ehook* *ahook* [*env*]) — evaluate with hooks
expr — the expression to evaluate
ehook — the value for `*evalhook*`
ahook — the value for `*applyhook*`
env — the environment (default is `nil`)
returns — the result of evaluating the expression

(profile[2] *flag*) — turn profiling on or off.
flag — `nil` turns profiling off, otherwise on
returns — the previous state of profiling.

III.26. Arithmetic Functions

(truncate *expr*) — truncates a floating point number to an integer
expr — the number
returns — the result of truncating the number

(float *expr*) — converts an integer to a floating point number
expr — the number
returns — the result of floating the integer

(+ *expr*...) — add a list of numbers
expr — the numbers
returns — the result of the addition

(- *expr*...) — subtract a list of numbers or negate a single number
expr — the numbers
returns — the result of the subtraction

(* *expr*...) — multiply a list of numbers
expr — the numbers
returns — the result of the multiplication

(/ *expr*...) — divide a list of numbers
expr — the numbers
returns — the result of the division

(1+ *expr*) — add one to a number
expr — the number
returns — the number plus one

(1- *expr*) — subtract one from a number
expr — the number
returns — the number minus one

(rem *expr*...) — remainder of a list of numbers
expr — the numbers
returns — the result of the remainder operation

(min *expr*...) — the smallest of a list of numbers

[2] This is not a standard XLisp function.

expr — the expressions to be checked
returns — the smallest number in the list

(max *expr*...) — the largest of a list of numbers
expr — the expressions to be checked
returns — the largest number in the list

(abs *expr*) — the absolute value of a number
expr — the number
returns — the absolute value of the number

(gcd *n1 n2*...) — compute the greatest common divisor
n1 — the first number (integer)
n2 — the second number(s) (integer)
returns — the greatest common divisor

(random *n*) — compute a random number between 1 and n-1
n — the upper bound (integer)
returns — a random number

(sin *expr*) — compute the sine of a number
expr — the floating point number
returns — the sine of the number

(cos *expr*) — compute the cosine of a number
expr — the floating point number
returns — the cosine of the number

(tan *expr*) — compute the tangent of a number
expr — the floating point number
returns — the tangent of the number

(expt *x-expr y-expr*) — compute x to the y power
x-expr — the floating point number
y-expr — the floating point exponent
returns — x to the y power

(exp *x-expr*) — compute e to the x power
x-expr — the floating point number
returns — e to the x power

(sqrt *expr*) — compute the square root of a number
expr — the floating point number
returns — the square root of the number

(< *n1 n2*...) — test for less than

(<= *n1 n2*...) — test for less than or equal to
(= *n1 n2*...) — test for equal to
(/= *n1 n2*...) — test for not equal to
(>= *n1 n2*...) — test for greater than or equal to
(> *n1 n2*...) — test for greater than

n1 — the first number to compare
n2 — the second number to compare
returns — `t` if the results of comparing *n1* with *n2*, *n2* with *n3*, etc., are all true.

III.27. Bitwise Logical Functions

(logand *expr*...) — the bitwise and of a list of numbers

expr — the numbers
returns — the result of the and operation

(logior *expr*...) — the bitwise inclusive or of a list of numbers

expr — the numbers
returns — the result of the inclusive or operation

(logxor *expr*...) — the bitwise exclusive or of a list of numbers

expr — the numbers
returns — the result of the exclusive or operation

(lognot *expr*) — the bitwise not of a number

expr — the number
returns — the bitwise inversion of number

III.28. String Functions

(string *expr*) — make a string from an integer ascii value

expr — the integer
returns — a one character string

(string-search[3] *pat str* &key :start :end) — search for pattern in string

pat — a string to search for
str — the string to be searched
:start — the starting offset in str
:end — the ending offset + 1
returns — index of pat in str or NIL if not found

(string-trim *bag str*) — trim both ends of a string

bag — a string containing characters to trim

---

[3] This is not a standard XLisp 2.0 function.

str — the string to trim
returns — a trimed copy of the string

(string-left-trim *bag str*) — trim the left end of a string
bag — a string containing characters to trim
str — the string to trim
returns — a trimed copy of the string

(string-right-trim *bag str*) — trim the right end of a string
bag — a string containing characters to trim
str — the string to trim
returns — a trimed copy of the string

(string-upcase *str* &key :start :end) — convert to uppercase
str — the string
:start — the starting offset
:end — the ending offset + 1
returns — a converted copy of the string

(string-downcase *str* &key :start :end) — convert to lowercase
str — the string
:start — the starting offset
:end — the ending offset + 1
returns — a converted copy of the string

(nstring-upcase *str* &key :start :end) — convert to uppercase
str — the string
:start — the starting offset
:end — the ending offset + 1
returns — the converted string (not a copy)

(nstring-downcase *str* &key :start :end) — convert to lowercase
str — the string
:start — the starting offset
:end — the ending offset + 1
returns — the converted string (not a copy)

(strcat *expr*...) — concatenate strings
expr — the strings to concatenate
returns — the result of concatenating the strings

(subseq *string start* [*end*]) — extract a substring
string — the string
start — the starting position (zero origin)
end — the ending position + 1 (defaults to end)
returns — substring between *start* and *end*

(string< *str1 str2* &key :start1 :end1 :start2 :end2)
(string<= *str1 str2* &key :start1 :end1 :start2 :end2)
(string= *str1 str2* &key :start1 :end1 :start2 :end2)
(string/= *str1 str2* &key :start1 :end1 :start2 :end2)
(string>= *str1 str2* &key :start1 :end1 :start2 :end2)
(string> *str1 str2* &key :start1 :end1 :start2 :end2)
    str1 — the first string to compare
    str2 — the second string to compare
    :start1 — first substring starting offset
    :end1 — first substring ending offset + 1
    :start2 — second substring starting offset
    :end2 — second substring ending offset + 1
    returns — `t` if predicate is true, `nil` otherwise
    Note: case is significant with these comparison functions.

(string-lessp *str1 str2* &key :start1 :end1 :start2 :end2)
(string-not-greaterp *str1 str2* &key :start1 :end1 :start2 :end2)
(string-equalp *str1 str2* &key :start1 :end1 :start2 :end2)
(string-not-equalp *str1 str2* &key :start1 :end1 :start2 :end2)
(string-not-lessp *str1 str2* &key :start1 :end1 :start2 :end2)
(string-greaterp *str1 str2* &key :start1 :end1 :start2 :end2)
    str1 — the first string to compare
    str2 — the second string to compare
    :start1 — first substring starting offset
    :end1 — first substring ending offset + 1
    :start2 — second substring starting offset
    :end2 — second substring ending offset + 1
    returns — `t` if predicate is true, `nil` otherwise
    Note: case is not significant with these comparison functions.

III.29. Character Functions

(char *string index*) — extract a character from a string
    string — the string
    index — the string index (zero relative)
    returns — the ascii code of the character

(upper-case-p *chr*) — is this an upper case character?
    chr — the character
    returns — `t` if the character is upper case, `nil` otherwise

(lower-case-p *chr*) — is this a lower case character?
    chr — the character
    returns — `t` if the character is lower case, `nil` otherwise

(both-case-p *chr*) — is this an alphabetic (either case) character?

chr — the character
returns — `t` if the character is alphabetic, `nil` otherwise

(digit-char-p *chr*) — is this a digit character?
chr — the character
returns — the digit weight if character is a digit, `nil` otherwise

(char-code *chr*) — get the ascii code of a character
chr — the character
returns — the ascii character code (integer)

(code-char *code*) — get the character with a specified ascii code
code — the ascii code (integer)
returns — the character with that code or `nil`

(char-upcase *chr*) — convert a character to upper case
chr — the character
returns — the upper case character

(char-downcase *chr*) — convert a character to lower case
chr — the character
returns — the lower case character

(digit-char *n*) — convert a digit weight to a digit
n — the digit weight (integer)
returns — the digit character or `nil`

(char-int *chr*) — convert a character to an integer
chr — the character
returns — the ascii character code

(int-char *int*) — convert an integer to a character
int — the ascii character code
returns — the character with that code

(char< *chr1 chr2*...)
(char<= *chr1 chr2*...)
(char= *chr1 chr2*...)
(char/= *chr1 chr2*...)
(char>= *chr1 chr2*...)
(char> *chr1 chr2*...)
chr1 — the first character to compare
chr2 — the second character(s) to compare
returns — `t` if predicate is true, `nil` otherwise
Note: case is significant with these comparison functions.

(char-lessp *chr1 chr2*...)

(char-not-greaterp *chr1 chr2*...)
(char-equalp *chr1 chr2*...)
(char-not-equalp *chr1 chr2*...)
(char-not-lessp *chr1 chr2*...)
(char-greaterp *chr1 chr2*...)

chr1 — the first string to compare
chr2 — the second string(s) to compare
returns — `t` if predicate is true, `nil` otherwise

Note: case is not significant with these comparison functions.

III.30. Input/Output Functions

(read [*stream* [*eof* [*rflag*]]]) — read an expression

stream — the input stream (default is standard input)
eof — the value to return on end of file (default is `nil`)
rflag — recursive read flag (default is `nil`)
returns — the expression read

(print *expr* [*stream*]) — print an expression on a new line

expr — the expression to be printed
stream — the output stream (default is standard output)
returns — the expression

(prin1 *expr* [*stream*]) — print an expression

expr — the expression to be printed
stream — the output stream (default is standard output)
returns — the expression

(princ *expr* [*stream*]) — print an expression without quoting

expr — the expressions to be printed
stream — the output stream (default is standard output)
returns — the expression

(pprint *expr* [*stream*]) — pretty print an expression

expr — the expressions to be printed
stream — the output stream (default is standard output)
returns — the expression

(terpri [*stream*]) — terminate the current print line

stream — the output stream (default is standard output)
returns — `nil`

(flatsize *expr*) — length of printed representation using prin1

expr — the expression
returns — the length

(flatc *expr*) — length of printed representation using princ
expr — the expression
returns — the length

III.31. The Format Function

(format *stream fmt arg*...) — do formated output
stream — the output stream
fmt — the format string
arg — the format arguments
returns — output string if *stream* is `nil`, `nil` otherwise

The format string can contain characters that should be copied directly to the output and formatting directives. The formatting directives are:

~A — print next argument using princ
~S — print next argument using prin1
~% — start a new line
~~ — print a tilde character

III.32. File I/O Functions

(open *fname* &key :direction) — open a file stream
fname — the file name string or symbol
:direction — :input or :output (default is :input)
returns — a stream

(close *stream*) — close a file stream
stream — the stream
returns — `nil`

(read-char [*stream*]) — read a character from a stream
stream — the input stream (default is standard input)
returns — the character

(peek-char [*flag* [*stream*]]) — peek at the next character
flag — flag for skipping white space (default is `nil`)
stream — the input stream (default is standard input)
returns — the character (integer)

(write-char *ch* [*stream*]) — write a character to a stream
ch — the character to write
stream — the output stream (default is standard output)
returns — the character

(read-line [*stream*]) — read a line from a stream
stream — the input stream (default is standard input)

returns — the string

(read-byte [*stream*]) — read a byte from a stream
stream — the input stream (default is standard input)
returns — the byte (integer)

(write-byte *byte* [*stream*]) — write a byte to a stream
byte — the byte to write (integer)
stream — the output stream (default is standard output)
returns — the byte (integer)

III.33. String Stream Functions

These functions operate on unnamed streams. An unnamed output stream collects characters sent to it when it is used as the destination of any output function. The functions `get-output-stream-string` and string or a list of characters.

An unnamed input stream is setup with the `make-string-input-stream` function and returns each character of the string when it is used as the source of any input function.

(make-string-input-stream *str* [*start* [*end*]])
str — the string
start — the starting offset
end — the ending offset + 1
returns — an unnamed stream that reads from the string

(make-string-output-stream)
returns — an unnamed output stream

(get-output-stream-string *stream*)
stream — the output stream
returns — the output so far as a string
Note: the output stream is emptied by this function

(get-output-stream-list *stream*)
stream — the output stream
returns — the output so far as a list
Note: the output stream is emptied by this function

III.34. System Functions

(load *fname* &key :verbose :print) — load a source file
fname — the filename string or symbol
:verbose — the verbose flag (default is t)

:print — the print flag (default is `nil`)
returns — the filename

(save *fname*) — save workspace to a file
fname — the filename string or symbol
returns — `t` if workspace was written, `nil` otherwise

(restore *fname*) — restore workspace from a file
fname — the filename string or symbol
returns — `nil` on failure, otherwise never returns

(dribble [*fname*]) — create a file with a transcript of a session
fname — file name string or symbol (if missing, close current transcript)
returns — `t` if the transcript is opened, `nil` if it is closed

(gc) — force garbage collection
returns — `nil`

(expand *num*) — expand memory by adding segments
num — the number of segments to add
returns — the number of segments added

(alloc *num*) — change number of nodes to allocate in each segment
num — the number of nodes to allocate
returns — the old number of nodes to allocate

(room) — show memory allocation statistics
returns — `nil`

(type-of *expr*) — returns the type of the expression
expr — the expression to return the type of
returns — `nil` if the value is `nil` otherwise one of the symbols:
SYMBOL — for symbols
OBJECT — for objects
CONS — for conses
SUBR — for built-in functions
FSUBR — for special forms
CLOSURE — for defined functions
STRING — for strings
FIXNUM — for integers
FLONUM — for floating point numbers
CHARACTER — for characters
FILE-STREAM — for file pointers
UNNAMED-STREAM — for unnamed streams
ARRAY — for arrays

(peek *addrs*) — peek at a location in memory

addrs — the address to peek at (integer)
returns — the value at the specified address (integer)

(poke *addrs value*) — poke a value into memory
addrs — the address to poke (integer)
value — the value to poke into the address (integer)
returns — the value

(address-of *expr*) — get the address of an xlisp node
expr — the node
returns — the address of the node (integer)

(exit) — exit xlisp
returns — never returns

III.35. File I/O Functions

III.35.1. Input from a File

To open a file for input, use the `open` function with the keyword argument `:direction` set to `:input`. To open a file for output, use the `open` function with the keyword argument `:direction` set to `:output`. The `open` function takes a single required argument which is the name of the file to be opened. This name can be in the form of a string or a symbol. The `open` function returns an object of type `FILE-STREAM` if it succeeds in opening the specified file. It returns the value `nil` if it fails. In order to manipulate the file, it is necessary to save the value returned by the `open` function. This is usually done by assigning it to a variable with the `setq` special form or by binding it using `let` or `let*`. Here is an example:

```
(setq fp (open "init.lsp" :direction :input))
```

Evaluating this expression will result in the file `init.lsp` being opened. The file object that will be returned by the `open` function will be assigned to the variable `fp`.

It is now possible to use the file for input. To read an expression from the file, just supply the value of the `fp` variable as the optional *stream* argument to `read`.

```
(read fp)
```

Evaluating this expression will result in reading the first expression from the file `init.lsp`. The expression will be returned as the result of the `read` function. More expressions can be read from the file using further calls to the `read` function. When there are no more expressions to read, the `read` function will return `nil` (or whatever value was supplied as the second argument to `read`).

Once you are done reading from the file, you should close it. To close the file, use the following expression:

```
(close fp)
```

Evaluating this expression will cause the file to be closed.

III.35.2. Output to a File

Writing to a file is pretty much the same as reading from one. You need to open the file first. This time you should use the `open` function to indicate that you will do output to the file. For example:

```
(setq fp (open "test.dat" :direction :output))
```

Evaluating this expression will open the file `test.dat` for output. If the file already exists, its current contents will be discarded. If it doesn't already exist, it will be created. In any case, a `FILE-STREAM` object will be returned by the `OPEN` function. This file object will be assigned to the `fp` variable.

It is now possible to write to this file by supplying the value of the `fp` variable as the optional *stream* parameter in the `print` function.

```
(print "Hello there" fp)
```

Evaluating this expression will result in the string ‘‘Hello there’’ being written to the file `test.dat`. More data can be written to the file using the same technique.

Once you are done writing to the file, you should close it. Closing an output file is just like closing an input file.

```
(close fp)
```

Evaluating this expression will close the output file and make it permanent.

III.35.3. A Slightly More Complicated File Example

This example shows how to open a file, read each Lisp expression from the file and print it. It demonstrates the use of files and the use of the optional *stream* argument to the `read` function.

```
(do* ((fp (open "test.dat" :direction :input))
      (ex (read fp) (read fp)))
     ((null ex) nil)
  (print ex))
```

References

[Dannenberg 89] Dannenberg, R. B. and C. L. Fraley. Fugue: Composition and Sound Synthesis With Lazy Evaluation and Behavioral Abstraction. In T. Wells and D. Butler (editor), *Proceedings of the 1989 International Computer Music Conference*, pages 76-79. International Computer Music Association, San Francisco, 1989.

[Touretzky 84] Touretzky, David S. *LISP: a gentle introduction to symbolic computation.* Harper & Row, New York, 1984.

Index

Table of Contents

List of Figures