Nyquist Reference Manual

Version 3.03

24 February 2009

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. Ning Hu ported Zheng (Geoffrey) Hua and Jim Beauchamp's piano synthesizer to Nyquist and also built NyqIDE, the Nyquist Interactive Development Environment for Windows. Dave Mowatt contributed the original version of jNyqIDE, the cross-platform interactive development environment. Dominic Mazzoni made a special version of Nyquist that runs within the Audacity audio editor, giving Nyquist a new interface and introducing Nyquist to many new users.

Many others have since contributed to Nyquist. Chris Tchou and Morgan Green worked on the Windows port. Eli Brandt contributed a number of filters and other synthesis functions. Pedro J. Morales, Eduardo Reck Miranda, Ann Lewis, and Erich Neuwirth have all contributed nyquist examples found in the demos folder of the Nyquist distribution. Philip Yam ported some synthesis functions from Perry Cook and Gary Scavone's STK to Nyquist. Pedro Morales ported many more STK instruments to Nyquist. Dave Borel wrote the Dolby Pro-Logic encoding library and Adam Hartman wrote stereo and spatialization effects. Stephen Mangiat wrote the MiniMoog emulator. Phil Light recorded the drum samples and wrote drum machine software. The Xmusic library, particularly the pattern specification, was inspired by Rick Taube's Common Music. The functions for generating probability distributions were implemented by Andreas Pfenning.

Starting with Version 3, Nyquist supports a version of SAL, providing an alternative to Lisp syntax. SAL was designed by Rick Taube, and the SAL implementation in Nyquist is based on Taube's original implementation as part of his Common Music system.

The current jNyqIDE includes contributions from many. Chris Yealy and Derek D'Souza implemented early versions of the envelope editor. Daren Makuck and Michael Rivera wrote the original equalizer editor. Priyanka Raghavan implemented the sound browser.

Many others have made contributions, offered suggestions, and found bugs. If you were expecting to find your name here, I apologize for the omission, and please let me know.

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 Linux, Apple Mac OS X, Microsoft Windows NT, 2000, XP, and Vista, and it produces sound files or directly generates audio. Recent versions have also run on AIX, NeXT, SGI, DEC pmax, and Sun Sparc machines. (Makefiles for many of these are included, but out-of-date). 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 IV is the reference manual for XLISP, of which Nyquist is a superset. Starting with Version 3, Nyquist supports a variant of SAL, which is also available in Common Music. Since there are some differences, one should generally call this implementation ‘‘Nyquist SAL;’’ however, in this manual, I will just call it ‘‘SAL.’’ SAL offers most of the capabilities of Lisp, but it uses an Algol-like syntax that may be more familiar to programmers with experience in Java, C, Basic, etc.

1.1. Installation

Nyquist is a C program intended to run under various operating systems including Unix, MacOS, and Windows.

1.1.1. Unix Installation

For Unix systems, Nyquist is distributed as a compressed tar file named `nyqsrc3`*nn*`.zip`, where *nn* is the version number (e.g. v3.01 was in `nyqsrc301.zip`). To install Nyquist, copy `nyqsrc3`*nn*`.zip` to the directory on your machine where you would like to install Nyquist, and type:

```
gunzip nyqsrc3nn.zip
cd nyquist
ln -s sys/unix/linux/Makefile Makefile
setenv XLISPPATH ‘pwd‘/runtime:‘pwd‘/lib
make
```

The first line creates a `nyquist` directory and some subdirectories. The second line (cd) changes directories to the new nyquist directory. The third line makes a link from the top-level directory to the Makefile for your system. In place of `linux` in `sys/unix/linux/Makefile`, you should substitute your system type. Current systems are `next`, `pmax`, `rs6k`, `sgi`, `linux`, 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

‘‘Examples’’ section.

One you establish that Nyquist (ny) is working from the command line, you should try using jNyqIDE, the Java-based Nyquist development environment. First, make `jny` executable (do this only once when you install Nyquist):

```
chmod +x jny
```

Then try running jNyqIDE by typing:

```
./jny
```

If the jNyqIDE window does not appear, make sure you have Java installed (if not, you probably already encountered errors when you ran `make`). You can also try recompiling the Java files:

```
cd jnyqide
javac *.java
cd ..
```

Note: With Linux and the Macintosh OS X, jNyqIDE defines the environment passed to Nyquist. If you set `XLISPPATH` as shown above, it will be passed along to Nyquist under jNyqIDE. If not, a default XLISPPATH will have the `lib` and `runtime` directories only. This does not apply to Windows because even though the environment is there, the Windows version of Nyquist reads the `XLISPPATH` from the Registry.

You can also specify the search path by creating the file `nyquist/xlisppath`, which should have colon-separated paths on a single line of text. This file will override the environment variable `XLISPPATH`.

It is good to have `USER` in the environment with your user ID. This string is used to construct some file names. jNyqIDE will look for it in the environment. You can also specify your user ID using the file `nyquist/user`, but if you have a shared installation of Nyquist, this will not be very useful.

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

1.1.2. Win32 Installation

The Win32 version of Nyquist is packaged as a compiled (runtime) system in an executable installer. A source version is also available (the same source download is for Win32, Mac OS X, and Linux). The source version is intended for developers who want to recompile Nyquist. The contents of the source archive are extracted to the `C:\nyquist` directory, but you can put it anywhere you like. You can then open the workspace file, nyquist.sln, using Microsoft Visual C++. You can build and run the command line version of Nyquist from within Visual C++. There is a batch file, `comp-ide.bat`, for bulding the Nyquist IDE. This requires the Java SDK from Sun Microsystems.

The runtime version contain everything you need to run Nyquist, including the executable, examples, and documentation, packaged as an executable installer program. After executing the installer, just find Nyquist in your Start menu to run it. You may begin typing expressions such as the ones in the following ‘‘Examples’’ section.

Optional: Nyquist needs to know where to find the standard runtime files. The location of runtime files must be stored in the Registry. The installers create a registry entry, but if you move Nyquist or deal with different versions, you can edit the Registry manually as follows:

- Run the Registry editor. Under Windows NT, run `C:\WINNT\system32\regedt32.exe`. Under Windows95, run `C:\WINDOWS\regedit.exe`.
- Find and highlight the `SOFTWARE` key under `HKEY_LOCAL_MACHINE`.
- Choose `Add key ...` from the `Edit` menu, type `CMU`, and click the `OK` button.
- Highlight the new `CMU` key.
- Choose `Add key ...` from the `Edit` menu, type `Nyquist`, and click the `OK` button. (Note that `CMU` and `Nyquist` are case sensitive.)
- Highlight the new `Nyquist` key.
- Choose `Add value ...` from the `Edit` menu, type `XLISPPATH`, and click the `OK` button. (Under WinXP the menu item is `Edit:New:String Value`, after which you need to select the new string name that appears in the right panel, select `Edit:Rename`, and type `XLISPPATH`.)
- In the String Edit box (select the `Edit:Modify` menu item in WinXP), type a list of paths you want Nyquist to search for lisp files. For example, if you installed Nyquist as `C:\nyquist`, then type:

  ```
  C:\nyquist\runtime,C:\nyquist\lib
  ```

  The paths should be separated by a comma or semicolon and no space. The `runtime` path is essential, and the `lib` path may become essential in a future release. You can also add paths to personal libraries of Lisp and Nyquist code.
- Click the `OK` button of the string box and exit from the Registry Editor application.

1.1.2.1. What if Nyquist functions are undefined?

If you do not have administrative privileges for your machine, the installer may fail to set up the Registry entry that Nyquist uses to find initialization files. In this case, Nyquist will run a lisp interpreter, but many Nyquist functions will not be defined. If you can log in as administrator, do it and reinstall Nyquist. If you do not have permission, you can still run Nyquist as follows:

Create a file named `init.lsp` in the same directory as Nyquist.exe (the default location is `C:\Program Files\Nyquist`, but you may have installed it in some other location.) Put the following text in `init.lsp`:

```
(setf *search-path* "C:/Program Files/Nyquist/runtime,C:/Program Files/Nyquist/lib")
(load "C:/Program Files/Nyquist/runtime/init.lsp")
```

Note: in the three places where you see `C:/Program Files/Nyquist`, insert the full path where Nyquist is actually installed. Use forward slashes (`/`) rather than back slashes (`\`) to separate directories. For example, if Nyquist is installed at `D:\rbd\nyquist`, then `init.lsp` should contain:

```
(setf *search-path* "D:/rbd/nyquist/runtime,D:/rbd/nyquist/lib")
(load "d:/rbd/nyquist/runtime/init.lsp")
```

The variable `*search-path*`, if defined, is used in place of the registry to determine search paths for files.

1.1.2.2. SystemRoot

(Ignore this paragraph if you are not planning to use Open Sound Control under Windows.) If Nyquist prints an error message and quits when you enable Open Sound Control (using `osc-enable`), check to see if you have an environment variable `SystemRoot`, e.g. type `set` to a command prompt and look for the value of `SystemRoot`. The normal value is `C:\windows`. If the value is something else, you should put the environment entry, for example:

```
SystemRoot="D:\windows"
```

into a file named `systemroot` (no extension). Put this file in your `nyquist` directory. When you run `jNyqIDE`, it will look for this file and pass the contents as an environment variable to Nyquist. The Nyquist process needs this to open a UDP socket, which is needed for Open Sound Control.

1.1.2.3. The "java is not recognized" Error

Sometimes, Nyquist will run directly from the installer, but then it will not start from the Windows Start menu. You can try running the `nyquist/jnyqide.bat` program from a Windows shell. If that fails, and you see an error similar to "java is not recognized as in internal or external command error", the problem may be that paths are not set up properly to allow the Windows shell to find java. Right click on ‘‘My Computer’’ on the Windows desktop and select ‘‘Properties.’’ Under the ‘‘Advanced’’ tap, press the ‘‘Environment Variables’’ button, and look for PATH under ‘‘System Variables.’’ Make sure the Java bin directory is on the path. If it is not, you will have to find your installation of Java and add the appropriate directory to the PATH variable, e.g. ‘‘C:\Program Files\Java\jdk1.6.0\bin.’’

1.1.3. MacOS X Installation

The OS X version of Nyquist is very similar to the Linux version, but it is developed using Xcode, Apple’s programming environment. With a little work, you can use the Linux installation instructions to compile Nyquist, but it might be simpler to just open the Xcode project that is included in the Nyquist sources.

You can also download a pre-compiled version of Nyquist for the Mac. Just download `nyqosx2xx.tgz` to the desktop and open it to extract the folder <tt>nyqosx2xx</tt>. (Again, "2xx" refers to the current version number, e.g. v2.31 would be named with "231".) Open the folder to find a Mac Application named jNyqIDE and a directory named <tt>nyquist/doc</tt>. Documentation is in the <tt>nyquist/doc</tt> directory.

The file <tt>jNyqIDE.app/Contents/Resources/Java/ny</tt> is the command line executable (if you should need it). To run from the command line, you will need to set the XLISPPATH environment variable as with Linux. On the topic of the `XLISPPATH`, note that this variable is set by jNyqIDE when running with that application, overriding any other value. You can extend the search path by creating the file `xlisppath` in the same directory as the nyquist executable `ny`. The `xlisppath` file should have colon-separated paths on a single line of text.

1.2. Using jNyqIDE

The program named jNyqIDE is an ‘‘integrated development environment’’ for Nyquist. When you run jNyqIDE, it starts the Nyquist program and displays all Nyquist output in a window. jNyqIDE helps you by providing a Lisp and SAL editor, hints for command completion and function parameters, some graphical interfaces for editing envelopes and graphical equalizers, and a panel of buttons for common operations. A more complete description of jNyqIDE is in Chapter 2.

For now, all you really need to know is that you can enter Nyquist commands by typing into the upper left window. When you type return, the expression you typed is sent to Nyquist, and the results appear in the window below. You can edit files by clicking on the New File or Open File buttons. After editing some text, you can load the text into Nyquist by clicking the Load button. jNyqIDE always saves the file first; then it tells Nyquist to load the file. You will be prompted for a file name the first time you load a new file.

1.3. Using SAL

SAL mode means that Nyquist reads and evaluates SAL commands rather than Lisp. The SAL mode prompt is "`SAL> `" while the Lisp mode prompt is "`> `". When Nyquist starts it normally enters SAL mode automatically, but certain errors may exit SAL mode. You can reenter SAL mode by typing the Lisp expression `(sal)`.

In SAL mode, you type commands in the SAL programming language. Nyquist reads the commands, compiles them into Lisp, and evaluates the commands. Commands can be entered manually by typing into the upper left text box in jNyqIDE.

1.4. Helpful Hints

Under Win95 and Win98, the console sometimes locks up. Activating another window and then reactivating the Nyquist window should unlock the output. (We suggest you use JNyqIDE, the interactive development environment rather than a console window.)

You can cut and paste text into Nyquist, but for serious work, you will want to use the Lisp `load` command. To save even more time, write a function to load your working file, e.g. `(defun l () (load "myfile.lsp"))`. Then you can type `(l)` to (re)load your file.

Using SAL, you can type

```
define function l () load "myfile.lsp"
```

and then

```
exec l()
```

to (re)load the file.

The Emacs editor is free GNU software and will help you balance parentheses if you use Lisp mode. In fact, you can run nyquist (without the IDE) as a subprocess to Emacs. A helful discussion is at //http://www.audacity-forum.de/download/edgar/nyquist/nyquist-doc/examples/emacs/main.html. If you use Emacs, there is also a SAL mode (the file is `sal-mode.el`) included with the Common Music distribution, which you can find on the Web at `sourceforge.net`.

The jNyqIDE also runs Nyquist as a subprocess and has built-in Lisp and SAL editors. If your editor does not help you balance parentheses, you may find yourself counting parens and searching for unbalanced expressions. If you are confused or desperate, try the `:print t` option o fthe `load` command. By looking at the expressions printed, you should be able to tell where the last unbalanced expression starts. Alternatively, type `(file-sexprs)` and type the lisp file name at the prompt. This function will read and print expressions from the file, reporting an error when an extra paren or end-of-file is reached unexpectedly.

1.5. Using Lisp

Lisp mode means that Nyquist reads and evaluates Nyquist expressions in Lisp syntax. Since Nyquist is build on a Lisp interpreter, this is the ‘‘native’’ or machine language of Nyquist, and certain errors and functions may break out of the SAL interpreter, leaving you with a prompt for a Lisp expression. Alternatively, you can exit SAL simply by typing `exit` to get a Lisp prompt (`> `). Commands can be entered manually by typing into the upper left text box in jNyqIDE.

1.6. Examples

We will begin with some simple Nyquist programs. Throughout the manual, we will assume SAL mode and give examples in SAL, but it should be emphasized that all of these examples can be performed using Lisp syntax. See Section 6.2 on the relationship between SAL and Lisp.

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/demos/examples.sal`. Corresponding Lisp syntax examples are in the file `nyquist/demos/examples.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` command, which plays the file through the machine’s D/A converters. It also writes a soundfile in case the computation cannot keep up with real time. You can then (re)play the file by typing:

```
exec r()
```

This `(r)` function is a general way to ‘‘replay’’ the last thing written by `play`.

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.

1.6.1. 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 0.5, 0.25, 0.125, and 0.0625. These are scaled and added (using `+`) 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.

```
define function mkwave()
  begin
    set *table* = 0.5 * build-harmonic(1.0, 2048) +
                  0.25 * build-harmonic(2.0, 2048) +
                  0.125 * build-harmonic(3.0, 2048) +
                  0.0625 * build-harmonic(4.0, 2048)
    set *table* = list(*table*, hz-to-step(1.0), #t)
  end
```

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:

```
if ! fboundp(quote(*mkwave*)) then
  begin
    exec mkwave()
    set *mkwave* = #t
  end
```

1.6.2. Wavetables

When Nyquist starts, several waveforms are created and stored in global variables for convenience. They are: `*sine-table*`, `*saw-table*`, and `*tri-table*`, implementing sinusoid, sawtooth, and triangle waves, respectively. The variable `*table*` is initialized to `*sine-table*`, and it is `*table*` that forms the default wave table for many Nyquist oscillator behaviors. If you want a proper, band-limited waveform, you should construct it yourself, but if you do not understand this sentence and/or you do not mind a bit of aliasing, give `*saw-table*` and `*tri-table*` a try.

Note that in Lisp and SAL, global variables often start and end with asterisks (*). These are not special syntax, they just happen to be legal characters for names, and their use is purely a convention.

1.6.3. Sequences

Finally, we define `my-note` to use the waveform, and play several notes in a simple score. Note that the function `my-note` has only one command (a `return` command), so it is not necessary to use `begin` and `end`. These are only necessary when the function body consists of a sequence of statements:

```
define function my-note(pitch, dur)
  return osc(pitch, dur, *table*)

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

Here, `my-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 `my-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 1.7 for a complete description. The result is the sum of all the computed sounds.

Sequences can also be constructed using the `at` transformation to specify time offsets. See `sequence_example.htm`demos, `sequence` for more examples and explanation.

1.6.4. 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.

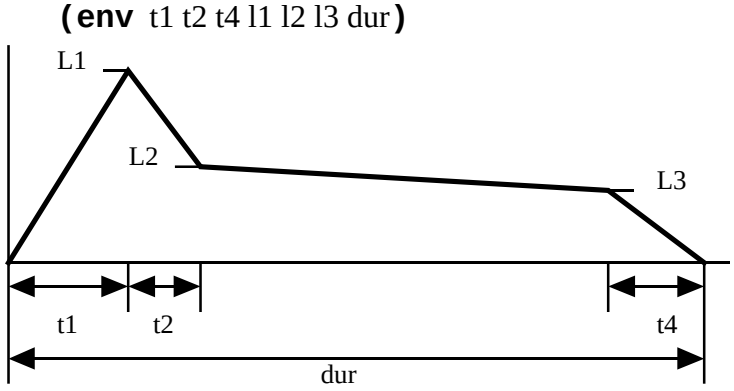

Figure 1: An envelope generated by the `env` function.

```
; env-note produces an enveloped note.  The duration
;   defaults to 1.0, but stretch can be used to change
;   the duration.
;   Uses my-note, defined above.
;
define function env-note(p)
  return my-note(p, 1.0) *
         env(0.05, 0.1, 0.5, 1.0, 0.5, 0.4)

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

While this example shows a smooth envelope multiplied by an audio signal, you can also multiply audio signals to achieve what is often called *ring modulation*. See the code and description in `demos/scratch_tutorial.htm` for an interesting use of ring modulation to create ‘‘scratch’’ sounds.

In the next example, The *stretch* operator (~)is used to modify durations:

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

In addition to *stretch*, there are a number of transformations supported by Nyquist, and transformations of abstract behaviors is perhaps *the* fundamental idea behind Nyquist. Chapter 3 is devoted to explaining this concept, and further elaboration can be found elsewhere [Dannenberg 89].

1.6.5. 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 value (but not the final time) is 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.
;
```

1.7. Predefined 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

- **Miscellaneous**

`ny:all` = ‘‘all the samples’’ (i.e. a big number) = 1000000000

1.8. More Examples

More examples can be found in the directory `demos`, part of the standard Nyquist release. In this directory, you will find the following and more:

- Gong sounds by additive synthesis(`demos/pmorales/b1.lsp` and `demos/mateos/gong.lsp`
- Risset’s spectral analysis of a chord (`demos/pmorales/b2.lsp`)
- Bell sounds (`demos/pmorales/b3.lsp`, `demos/pmorales/e2.lsp`, `demos/pmorales/partial.lsp`, and `demos/mateos/bell.lsp`)
- Drum sounds by Risset (`demos/pmorales/b8.lsp`
- Shepard tones (`demos/shepard.lsp` and `demos/pmorales/b9.lsp`)
- Random signals (`demos/pmorales/c1.lsp`)
- Buzz with formant filters (`demos/pmorales/buzz.lsp`
- Computing samples directly in Lisp (using Karplus-Strong and physical modelling as examples) (`demos/pmorales/d1.lsp`
- FM Synthesis examples, including bell, wood drum, brass sounds, tuba sound (`demos/mateos/tuba.lsp` and clarinet sounds (`demos/pmorales/e2.lsp`
- Rhythmic patterns (`demos/rhythm_tutorial.htm`
- Drum Samples and Drum Machine (`demos/plight/drum.lsp`.

2. The jNyqIDE Program

The jNyqIDE program combines many helpful functions and interfaces to help you get the most out of Nyquist. jNyqIDE is implemented in Java, and you will need the Java runtime system or development system installed on your computer to use jNyqIDE. The best way to learn about jNyqIDE is to just use it. This chapter introduces some of the less obvious features. If you are confused by something and you do not find the information you need here, please contact the author.

2.1. jNyqIDE Overview

The jNyqIDE runs the command-line version of Nyquist as a subtask, so everything that works in Nyquist should work when using the jNyqIDE and vice-versa. Input to Nyquist is usually entered in the top left window of the jNyqIDE. When you type return, if the expression or statement appears to be complete, the expression you typed is sent to Nyquist. Output from Nyquist appears in a window below. You cannot type into or edit the output window text.

The normal way to use the jNyqIDE is to create or open one or more files. You edit these files and then click the Load button. To load a file, jNyqIDE saves the file, sets the current directory of Nyquist to that of the file, then issues a `load` command to Nyquist. In this case and several others, you may notice that jNyqIDE sends expressions to Nyquist automatically for evaluation. You can always see the commands and their results in the output window.

Notice that when you load a selected file window, jNyqIDE uses `setdir` to change Nyquist's current directory. This helps to keep the two programs in sync. Normally, you should keep all the files of a project in the same directory and avoid manually changing Nyquist's current directory (i.e. avoid calling `setdir` in your code).

Arranging windows in the jNyqIDE can be time-consuming, and depending on the operating system, it is possible for a window to get into a position where you cannot drag it to a new position. The Window:Tile menu command can be used to automatically lay out windows in a rational way. There is a preference setting to determine the height of the completion list relative to the height of the output window.

2.2. The Button Bar

There are a number of buttons with frequently-used operations. These are:

- Info — Print information about Nyquist memory utilization, including the number of free cons cells, the number of garbage collections, the total number of cons cells, the total amount of sample buffer memory, and the amount of memory in free sample buffers.
- Break — Send a break character to XLISP. This can be used to enter the debugger (the break loop) while a program is running. Resume by typing `(co)`.
- SAL/Lisp — Switch modes. The button names the mode (SAL or Lisp) you will switch to, not the current mode. For example, if you are in Lisp mode and want to type a SAL command, click the SAL button first.
- Top — Enters `(top)` into Nyquist. If the XLISP prompt is `1>` or some other integer followed by ‘‘`>`’’, clicking the Top button will exit the debug loop and return to the top-level prompt.
- Replay — Enters `(r)` into Nyquist. This command replays the last computed sound.

- F2-F12 — Enters `(f2)` etc. into Nyquist. These commands are not built-in, and allow users to define their own custom actions.
- Browse — Equivalent to the Window:Browse menu item. (See Section 2.4.)
- EQ — Equivalent to the Window:EQ menu item. (See Section 2.6.)
- EnvEdit — Equivalent to the Window:Envelope Edit menu item. (See Section 2.5.)
- NewFile — Equivalent to the File:New menu item. Opens a new file editing window for creating and loading a Lisp or SAL program file.
- OpenFile — Equivalent to the File:Open menu item. Opens an existing Lisp or SAL program file for editing and loading.
- SaveFile — Equivalent to the File:Save menu item (found on the editing window's menu bar). Saves the contents of an editing window to its associated file.
- Load — Equivalent to the File:Load menu item (found on the editing window's menu bar). Performs a Save operation, then sends a command to Nyquist that loads the file as a program.
- Mark — Sends a Control-A to Nyquist. While playing a sound, this displays and records the approximate time in the audio stream. (See Section 7.5 for more detail.)

2.3. Command Completion

To help with programming, jNyqIDE maintains a command-completion window. As you type the first letters of function names, jNyqIDE lists matching functions and their parameters in the Completion List window. If you click on an entry in this window, the displayed expression will replace the incompletely typed function name. A preference allows you to match initial letters or any substring of the complete function name. This is controlled by the ''Use full search for code completion'' preference.

In addition, if you right click (or under Mac OS X, hold down the Alt/Option key and click) on an entry, jNyqIDE will display documentation for the function. Documentation can come from a local copy or from the online copy (determined by the ''Use online manual instead of local copy'' preference). Documentation can be displayed within the jNyqIDE window or in an external browser (determined by the ''Use window in jNyqIDE for help browser'' preference.) Currently, the external browser option does not seem to locate documentation properly, but this should be fixed in the future.

2.4. Browser

If you click on the Browse button or use the Window:Browse menu command, jNyqIDE will display a browser window that is pre-loaded with a number of Nyquist commands to create sounds. You can adjust parameters, audition the sounds, and capture the expression that creates the sound. In many cases, the expression checks to see if necessary functions are defined, loading files if necessary before playing the sound. If you want to use a sound in your own program, you can often simplify things by explicitly loading the required file just once at the beginning of your file.

Since Nyquist now supports a mix of Lisp and SAL, you may find yourself in the position of having code from the browser in one language while you are working in the other. The best way to handle this is to put the code for the sound you want into a function defined in a Lisp (`.lsp`) or SAL (`.sal`) file. Load the file (from Lisp, use the `sal-load` command to load a SAL file), and call the function from the language of your choice.

2.5. Envelope Editor

The envelope editor allows you graphically to design and edit piece-wise linear and exponential envelopes. The editor maintains a list of envelopes and you select the one to edit or delete using the drop down list in the Saved Envelopes List area. The current envelope appears in the Graphical Envelope Editor area. You can click to add or drag points. Alternatively, you can use the Envelope Points window to select and edit any breakpoint by typing coordinates. The duration of the envelope is controlled by the Stop field in the Range area, and the vertical axis is controlled by the Min and Max fields.

When you click the Save button, *all* envelopes are written to Nyquist. You can then use the envelope by treating the envelope name as a function. For example, if you define an envelope named ‘‘fast-attack,’’ then you can create the envelope within a Nyquist SAL program by writing the expression `fast-attack()`.

These edited envelopes are saved to a file named `workspace.lsp` in the current directory. The workspace is Nyquist’s mechanism for saving data of all kinds (see Section 13.4.5). The normal way to work with workspaces is to (1) load the workspace, i.e. `load "workspace"`, as soon as you start Nyquist; (2) invoke the envelope editor to change values in the workspace; and (3) save the workspace at any time, especially before you exit jNyqIDE. If you follow these steps, envelopes will be preserved from session to session, and the entire collection of envelopes will appear in the editor. Be sure to make backups of your `workspace.lsp` file along with your other project files.

The envelope editor can create linear and exponential envelopes. Use the Type pull-down menu to select the type you want. Envelopes can be created using default starting and ending values using `pwl` or `pwe`, or you can specify the initial values using `pwlv` or `pwev`. The envelope editor uses `pwl` or `pwe` if no point is explicitly entered as the initial or final point. To create a `pwlv` or `pwev` function, create a point and drag it to the leftmost or rightmost edge of the graphical editing window. You will see the automatically generated default starting or ending point disappear from the graph.

Exponential envelopes should never decay to zero. If you enter a zero amplitude, you will see that the envelope remains at zero to the next breakpoint. To get an exponential decay to ‘‘silence,’’ try using an amplitude of about 0.001 (about -60dB). To enter small values like this, you can type them into the Amplitude box and click ‘‘Update Point.’’

The Load button refreshes the editor from data saved in the Nyquist process. Normally, there is no need to use this because the editor automatically loads data when you open it.

2.6. Equalizer Editor

The Equalizer Editor provides a graphical EQ interface for creating and adjusting equalizers. Unlike the envelope editor, where you can type any envelope name, equalizers are named `eq-0`, `eq-1`, etc., and you select the equalizer to edit using a pull-down menu. The `Set` button should be use to record changes.

3. 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 the ’’sustain’’ portion of an envelope to obtain a longer note.

3.1. The Environment

The transformation environment consists of a set of special 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 real (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.

3.2. 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(my-note(c4, q), my-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 my-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/SAL scope rules. This is because the `seq` captures the expression in a closure, which retains all of the variable bindings.

3.3. Simultaneous Behavior

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

```
play 0.5 * sim(my-note(c4, q), my-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 sum of the results. (In SAL, the `sim` function applied to sounds is equivalent to adding them with the infix `+` operator. The following section illustrates two concepts: first, a *sound* is not a *behavior*, and second, the `sim` operator and and the `at` transformation can be used to place sounds in time.

3.4. 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
;
set a-snd = s-read(strcat(current-path(), "demo-snd.aiff"))

; play it
;
play a-snd
```

One might then be tempted to write the following:

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

Why is this wrong? 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 variable) is always the same sound, regardless of the environment; in this case, the second `a-snd` *should* start at time `0.0`, just like the first. In this case, after the first sound ends, Nyquist is unable to ‘‘back up’’ to time zero, so in fact, this *will* play two sounds in sequence, but that is a result of an implementation detail rather than correct program execution. In fact, a future version of Nyquist might (correctly) stop and report an error when it detects that the second sound in the sequence has a real start time that is before the requested one.

How then do we obtain a sequence of two sounds properly? 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:

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

This example is correct because the second expression will shift the sound stored in `a-snd` to start at the end time of the first expression.

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(cue(a-snd) @ 0.0,
         cue(a-snd) @ 0.7,
         cue(a-snd) @ 1.0,
         cue(a-snd) @ 1.2)
```

3.5. The At Transformation

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

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

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 `@` 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 `cue(a-snd) @ 0.7` is just a sound, so `sim` would ‘‘undo’’ the effect of `@`, making all of the sounds in the previous example start simultaneously, in spite of the `@`! 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.

3.6. The Stretch Transformation

In addition to At (denoted in SAL by the `@` operator, the Stretch transformation is very important. It appeared in the introduction, and it is denoted in SAL by the `~` operator (or in LISP by the `stretch` special form). Stretch also operates on the `*warp*` component of the environment. For example,

```
osc(c4) ~ 3
```

does the following: modify `*warp*`, scaling the degree of "stretch" by 3, and evaluate `osc(c4)`. The `osc` behavior uses the stretch factor to determime the duration, so it will return a sound that is 3 seconds long. Restore `*warp*` to its original value. Like At, Stretch only affects behaviors. `a-snd ~ 10` is equivalent to `a-snd` because `a-snd` is a sound, not a behavior. Behaviors are functions that compute sounds according to the environment and return a sound.

3.7. Nested Transformations

Transformations can be combined using nested expressions. For example,

```
sim(cue(a-snd),
    loud(6.0, cue(a-snd) @ 3))
```

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

Why use `loud` instead of simply multiplying `a-snd` by some scale factor? Using `loud` gives the

behavior the chance to implement the abstract property *loudness* in an appropriate way, e.g. by including timbral changes. In this case, the behavior is `cue`, which implements *loudness* by simple amplitude scaling, so the result is equivalent to multiplication by `db-to-linear(6.0)`.

Transformations can also be applied to groups of behaviors:

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

3.8. Defining Behaviors

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

```
define function snds(dly)
  return sim(cue(a-snd) @ 0.0,
             cue(a-snd) @ 0.7,
             cue(a-snd) @ 1.0,
             cue(a-snd) @ (1.2 + dly))

play snds(0.1)
play loud(0.25, snds(0.3) ~ 0.9)
```

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 the *stretch* (`~`) will apply to the shift (`@`) 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 7.3 describes the full set of transformations.

3.9. Overriding Default Transformations

In Nyquist, behaviors are *the* important abstraction mechanism. A behavior represents a class of related functions or sounds. For example, a behavior can represent a musical note. When a note is stretched, it usually means that the tone sustains for more oscillations, but if the ‘‘note’’ is a drum roll, the note sustains by more repetitions of the component drum strokes. The concept of sustain is so fundamental that we do not really think of different note durations as being different instances of an abstract behavior, but in a music programming language, we need a way to model these abtract behaviors. As the tone and drum roll examples show, there is no one right way to ‘‘stretch,’’ so the language must allow users to define exactly what it means to stretch. By extension, the Nyquist programmer can define how all of the transformations affect different behaviors.

To make programming easier, almost all Nyquist sounds are constructed from primitive behaviors that obey the environment in obvious ways: Stretch transformations make things longer and At transformations shift things in time. But sometimes you have to override the default behaviors. Maybe the attack phase of an envelope should not stretch when the note is stretched, or maybe when you stretch a trill, you should get more notes rather than a slower trill.

To override default behaviors, you almost always follow the same programming pattern: first, capture the environment in a local variable; then, use one of the absolute transformations to ‘‘turn off’’ the environment’s effect and compute the sound as desired. The following example creates a very simple

envelope with a fixed rise time to illustrate the technique.

```
define function two-phase-env(rise-time)
  begin
    with dur = get-duration(1)
    return pwl(rise-time, 1, dur) ~~ 1.0
  end
```

To ‘‘capture the environment in a local variable,’’ a `with` construct is used to create the local variable `dur` and set it to the value of `get-duration(1)`, which answers the question: ‘‘If I apply use the environment to stretch something whose nominal duration is 1, what is the resulting duration?’’ (Since time transformations can involve continuous time deformations, this question is not as simple as it may sound, so please use the provided function rather than peeking inside the `*warp*` structure and trying to do it yourself.) Next, we ‘‘turn off’’ stretching using the `stretch-abs` form, which in SAL is denoted by the `~~` operator. Finally, we are ready to compute the envelope using `pwl`. Here, we use absolute durations. The first breakpoint is at `rise-time`, so the attack time is given by the `rise-time` parameter. The `pwl` decays back to zero at time `dur`, so the overall duration matches the duration expected from the environment encountered by this instance of `two-phase-env`. Note, however, that since the `pwl` is evaluated in a different environment established by `~~`, it is not stretched (or perhaps more accurately, it is stretched by 1.0). This is good because it means `rise-time` will not be stretched, but we must be careful to extend the envelope to `dur` so that it has the expected duration.

3.10. 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 even if the default rate is set to something different.

As with other components of the environment, you should *never* change `*sound-srate*` or `*control-srate*` directly. 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)
```

You can also do this using preferences in jNyqIDE. If you have already started Nyquist and want to change the defaults, the preferences or the following functions can be used:

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

These modify the default values and reinitialize the Nyquist environment.

4. 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.

4.1. 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. The `seqrep` ‘‘function’’ works like `seq` except that it creates copies of a sound by evaluating an expression multiple times. Here, `i` takes on 16 values from 0 to 15, and the expression for the sound could potentially use `i`. Technically, `seqrep` is not really a function but an abbreviation for a special kind of loop construct.

```
define function tone-seq()
  return seqrep(i, 16,
                osc-note(c4) ~ 0.25)
```

Now define a linearly increasing ramp to serve as a transposition function: `define function pitch-rise() return sustain-abs(1.0, 16 * ramp() ~ 4)` 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). The ramp is inside a `sustain-abs` transformation, which prevents a `sustain` transformation from having any effect on the ramp. (One of the drawbacks of behavioral abstraction is that built-in behaviors sometimes do the wrong thing implicitly, requiring some explicit correction to turn off the unwanted transformation.) Now, `pitch-rise` is used to transpose `tone-seq`: `define function chromatic-scale() return 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: `play sustain((0.2 + ramp()) ~ 4, 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 3, 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 obtained a chromatic scale anyway.

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 `partial`, `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`.)

4.2. 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`:

```
define function warper()
  return pwl(0.25, .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:

```
define function warp4()
  return 4 * warper() ~ 4
```

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.

4.3. 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

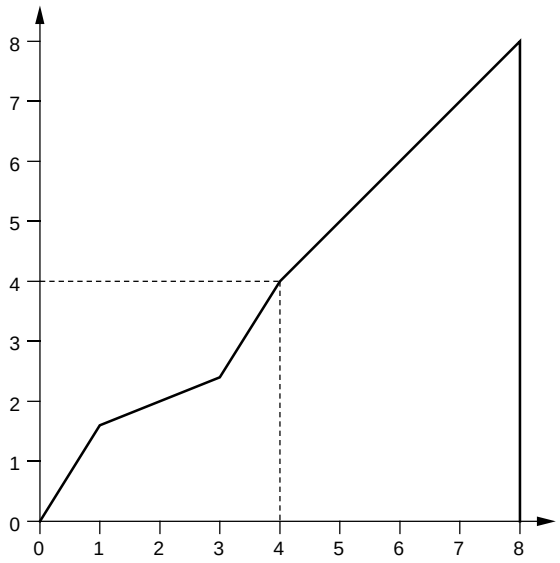

Figure 2: The result of `(warp4)`, intended to map 4 seconds of score time into 4 seconds of real time. The function extends beyond 4 seconds (the dashed lines) to make sure the function is well-defined at location (4, 4). Nyquist sounds are ordinarily open on the right.

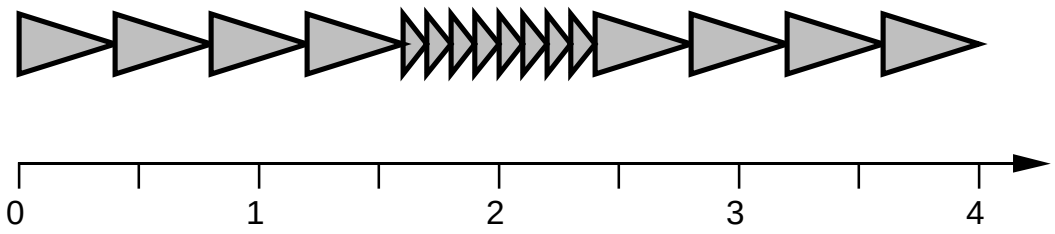

Figure 3: When `(warp4)` is applied to `(tone-seq-2)`, the note onsets and durations are warped.

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:

```
define function tone-seq-2 ()
  return seqrep(i, 16,
                osc-note(c4) ~~ 0.25)

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

Here, we have redefined `tone-seq`, renaming it to `tone-seq-2` and changing the stretch (`~`) to absolute stretch (`~~`). The absolute stretch 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 absolute stretch (`~~`) 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:

```
define function tone-seq-3()
  return seqrep(i, 16,
                set-logical-stop(osc-note(c4) ~~ 0.25, 0.25))

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

Notice the addition of `set-logical-stop` enclosing the absolute stretch (`~~`) 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 `~~`! 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.

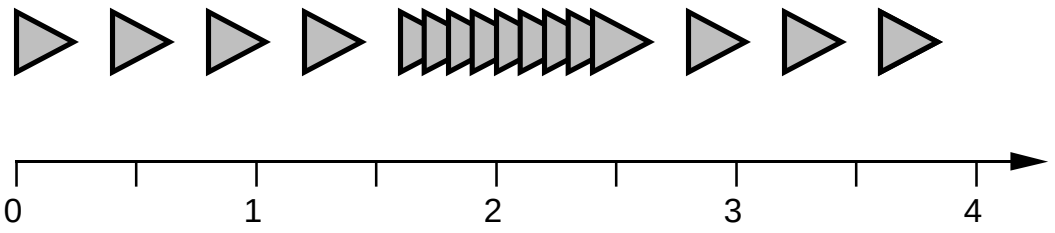

Figure 4: When `(warp4)` is applied to `(tone-seq-3)`, the note onsets are warped, but not the duration, which remains a constant 0.25 seconds. In the fast middle section, this causes notes to overlap. Nyquist will sum (mix) them.

4.4. 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. When the notes are moving faster, they will be closer together in pitch, and the result is not chromatic. What we need is a way to properly compose the warp and ramp functions. If we continuously warp the ramp function in the same way as the note sequence, a chromatic scale should be obtained. 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 desired function to `control-warp`, so we fetch it from the environment with `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.

5. More Examples

This chapter explores Nyquist through additional examples. The reader may wish to browse through these and move on to Chapter 7, which is a reference section describing Nyquist functions.

5.1. Stretching Sampled Sounds

This example illustrates how to stretch a sound, resampling it in the process. Because sounds in Nyquist are *values* that contain the sample rate, start time, etc., use `sound` to convert a sound into a behavior that can be stretched, e.g. `sound(a-snd)`. This behavior stretches a sound according to the stretch factor in the environment, set using `stretch`. For accuracy and efficiency, Nyquist does not resample a stretched sound until absolutely necessary. The `force-srate` function is used to resample the result so that we end up with a ‘‘normal’’ sample rate that is playable on ordinary sound cards.

```
; if a-snd is not loaded, load sound sample:
;
if not(boundp(quote(a-snd))) then
  set a-snd = s-read("demo-snd.aiff")

; the SOUND operator shifts, stretches, clips and scales
; a sound according to the current environment
;
define function ex23()
  play force-srate(*default-sound-srate*,  sound(a-snd) ~ 3.0)

define function down()
  return force-srate(*default-sound-srate*,
                     seq(sound(a-snd) ~ 0.2,
                         sound(a-snd) ~ 0.3,
                         sound(a-snd) ~ 0.4,
                         sound(a-snd) ~ 0.6))
play down()

; that was so much fun, let's go back up:
;
define function up()
  return force-srate(*default-sound-srate*,
                     seq(sound(a-snd) ~ 0.5,
                         sound(a-snd) ~ 0.4,
                         sound(a-snd) ~ 0.3,
                         sound(a-snd) ~ 0.2))

; 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 or changing the effective sample rate 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` may use linear interpolation rather than a high-quality resampling algorithm. In some cases, this may introduce errors audible as noise. Use `resample` (see Section 7.2.2) for high-quality

interpolation.

In the functions `up` and `down`, the `*warp*` is set by *stretch* (`~`), which simply scales time by a constant scale factor. In this case, `sound` can ‘‘stretch’’ the signal simply by changing the sample rate without any further computation. When `seq` tries to add the signals together, it discovers the sample rates do not match and uses linear interpolation to adjust all sample rates to match that of the first sound in the sequence. The result of `seq` is then converted using `force-srate` to convert the sample rate, again using linear interpolation. It would be slightly better, from a computational standpoint, to apply `force-srate` individually to each stretched sound rather than applying `force-srate` after `seq`.

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

5.2. Saving Sound Files

So far, we have used the `play` command to play a sound. The `play` command works by writing a sound to a file while simultaneously playing it. 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*
;
exec s-save(a-snd, 1000000000, "./a-snd-file.snd")

; play a file
; play command normally expects an expression for a sound
; but if you pass it a string, it will open and play a
; sound file
play "./a-snd-file.snd"

; delete the file (do this with care!)
; only works under Unix (not Windows)
exec system("rm ./a-snd-file.snd")

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

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

; play-file is a function to open and play a sound file
exec play-file(my-sound-file)

exec 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.)

5.3. 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. Nyquist has an automated facility to help with normalization. By default, Nyquist computes up to 1 million samples (using about 4MB of memory) looking for the peak. The entire sound is normalized so that this peak will not cause clipping. If the sound has less than 1 million samples, or if the first million samples are a good indication of the overall peak, then the signal will not clip.

With this automated normalization technique, you can choose the desired peak value by setting `*autonorm-target*`, which is initialized to 0.9. The number of samples examined is `*autonorm-max-samples*`, initially 1 million. You can turn this feature off by executing:

```
exec autonorm-off()
```

and turn it back on by typing:

```
exec autonorm-on()
```

This normalization technique is in effect when `*autonorm-type*` is `quote(lookahead)`, which is the default.

An alternative normalization method uses the peak value from the previous call to `play`. After playing a file, Nyquist can adjust an internal scale factor so that if you play the same file again, the peak amplitude will be `*autonorm-target*`, which is initialized to 0.9. This can be useful if you want to carefully normalize a big sound that does not have its peak near the beginning. To select this style of normalization, set `*autonorm-type*` to the (quoted) atom `quote(previous)`.

You can also create your own normalization method in Nyquist. The `peak` function computes the maximum value of a sound. The peak value is also returned from the `play` macro. 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:
set 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:
set mymax = snd-max(mysound, NY:ALL)
display "Computed max", mymax
; now write out and play the sound from memory with a scale factor:
play mysound * (0.9 / mymax)

; if you don't have space in memory, here's how to do it:
define function myscore()
  return sim(osc(c4), osc(c5))
; compute the maximum:
set mymax = snd-max(list(quote(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 myscore() * (0.9 / mymax)
```

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. If a long computation was involved, it should be much faster to scale the saved sound file than to recompute the sound from scratch. 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). You can use Nyquist to rescale a floating point file and convert it to 16-bit samples for playback.

5.4. 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*`)`

Note that pitch is the number of half-steps, e.g. `c4` has the value of 60 which is middle-C, and phase is in degrees. 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.
;
define function sweep(delay, pitch-1, sweep-time,
                      pitch-2, hold-time)
  begin
    with interval = step-to-hz(pitch-2) - step-to-hz(pitch-1)
    return 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)
  end


; 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, 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`.

```
set modulator = pwl(1.0, 1000.0, 1.0005) *
                osc(c4)
; make the sound
play fmosc(c4, modulator)
```

For more simple examples of FM in Nyquist, see `demos/warble_tutorial.htm`. Another interesting FM sound reminiscent of ‘‘scratching’’ can be found with a detailed explanation in `demos/scratch_tutorial.htm`..

5.5. Building a Wavetable

In Section 1.6.1, 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(quote(a-snd))) then
  set a-snd = s-read("demo-snd.aiff")

set *fm-voice* = list(extract(0.110204, 0.13932, cue(a-snd)),
                      24.848422,
                      #T)
```

The file *demos/examples.sal* 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.)

5.6. Filter Examples

Nyquist provides a variety of 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 7.3) to specify the default control sample rate, or use `force-srate` (Section 7.2.2) 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(10.0 * noise(), 500.0, 20.0, 1)
```

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

```
play reson(0.04 * noise(),
           pwl(0.0, 200.0, 1.0, 1000.0, 1.0),
           20.0)
```

For another example with explanations, see `demos/wind_tutorial.htm`.

5.7. DSP in Lisp

In almost any signal processing system, the vast majority of computation takes place in the inner loops of DSP algorithms, and Nyquist is designed so that these time-consuming inner loops are in highly-optimized machine code rather than relatively slow interpreted lisp code. As a result, Nyquist typically spends 95% of its time in these inner loops; the overhead of using a Lisp interpreter is negligible.

The drawback is that Nyquist must provide the DSP operations you need, or you are out of luck. When Nyquist is found lacking, you can either write a new primitive signal operation, or you can perform DSP in Lisp code. Neither option is recommended for inexperienced programmers. Instructions for extending Nyquist are given in Appendix I. This section describes the process of writing a new signal processing function in Lisp.

Before implementing a new DSP function, you should decide which approach is best. First, figure out how much of the new function can be implemented using existing Nyquist functions. For example, you might think that a tapped-delay line would require a new function, but in fact, it can be implemented by composing sound transformations to accomplish delays, scale factors for attenuation, and additions to combine the intermediate results. This can all be packaged into a new Lisp function, making it easy to use. If the function relies on built-in DSP primitives, it will execute very efficiently.

Assuming that built-in functions cannot be used, try to define a new operation that will be both simple and general. Usually, it makes sense to implement only the kernel of what you need, combining it with existing functions to build a complete instrument or operation. For example, if you want to implement a physical model that requires a varying breath pressure with noise and vibrato, plan to use Nyquist functions to add a basic pressure envelope to noise and vibrato signals to come up with a composite pressure signal. Pass that signal into the physical model rather than synthesizing the envelope, noise, and vibrato within the model. This not only simplifies the model, but gives you the flexibility to use all of Nyquist's operations to synthesize a suitable breath pressure signal.

Having designed the new ''kernel'' DSP operation that must be implemented, decide whether to use C or Lisp. (At present, SAL is not a good option because it has no support for object-oriented

programming.) To use C, you must have a C compiler, the full source code for Nyquist, and you must learn about extending Nyquist by reading Appendix I. This is the more complex approach, but the result will be very efficient. A C implementation will deal properly with sounds that are not time-aligned or matched in sample rates. To use Lisp, you must learn something about the XLISP object system, and the result will be about 50 times slower than C. Also, it is more difficult to deal with time alignment and differences in sample rates. The remainder of this section gives an example of a Lisp version of `snd-prod` to illustrate how to write DSP functions for Nyquist in Lisp.

The `snd-prod` function is the low-level multiply routine. It has two sound parameters and returns a sound which is the product of the two. To keep things simple, we will assume that two sounds to be multiplied have a matched sample rate and matching start times. The DSP algorithm for each output sample is simply to fetch a sample from each sound, multiply them, and return the product.

To implement `snd-prod` in Lisp, three components are required:

1. An object is used to store the two parameter sounds. This object will be called upon to yield samples of the result sound;
2. Within the object, the `snd-fetch` routine is used to fetch samples from the two input sounds as needed;
3. The result must be of type `SOUND`, so `snd-fromobject` is used to create the result sound.

The combined solution will work as follows: The result is a value of type `sound` that retains a reference to the object. When Nyquist needs samples from the sound, it invokes the sound's ''fetch'' function, which in turn sends an XLISP message to the object. The object will use `snd-fetch` to get a sample from each stored sound, multiply the samples, and return a result.

Thus the goal is to design an XLISP object that, in response to a `:next` message will return a proper sequence of samples. When the sound reaches the termination time, simply return `NIL`.

The XLISP manual (see Appendix IV describes the object system, but in a very terse style, so this example will include some explanation of how the object system is used. First, we need to define a class for the objects that will compute sound products. Every class is a subclass of class `class`, and you create a subclass by sending `:new` to a class.

```
(setf product-class (send class :new '(s1 s2)))
```

The parameter `'(s1 s2)` says that the new class will have two instance variables, `s1` and `s2`. In other words, every object which is an instance of class `product-class` will have its own copy of these two variables.

Next, we will define the `:next` method for `product-class`:

```
(send product-class :answer :next '()
  '((let ((f1 (snd-fetch s1))
          (f2 (snd-fetch s2)))
      (cond ((and f1 f2)
             (* f1 f2))
            (t nil)))))
```

The `:answer` message is used to insert a new method into our new `product-class`. The method is described in three parts: the name (`:next`), a parameter list (empty in this case), and a list of expressions to be evaluated. In this case, we fetch samples from `s1` and `s2`. If both are numbers, we return their

product. If either is NIL, we terminate the sound by returning nil.

The :next method assumes that s1 and s2 hold the sounds to be multiplied. These must be installed when the object is created. Objects are created by sending :new to a class. A new object is created, and any parameters passed to :new are then sent in a :isnew message to the new object. Here is the :isnew definition for product-class:

```
(send product-class :answer :isnew '(p1 p2)
  '((setf s1 (snd-copy p1))
    (setf s2 (snd-copy p2))))
```

Take careful note of the use of snd-copy in this initialization. The sounds s1 and s2 are modified when accessed by snd-fetch in the :next method defined above, but this destroys the illusion that sounds are immutable values. The solution is to copy the sounds before accessing them; the original sounds are therefore unchanged. (This copy also takes place implicitly in most Nyquist sound functions.)

To make this code safer for general use, we should add checks that s1 and s2 are sounds with identical starting times and sample rates; otherwise, an incorrect result might be computed.

Now we are ready to write snd-product, an approximate replacement for snd-prod:

```
(defun snd-product (s1 s2)
  (let (obj)
    (setf obj (send product-class :new s1 s2))
    (snd-fromobject (snd-t0 s1) (snd-srate s1) obj)))
```

This code first creates obj, an instance of product-class, to hold s1 and s2. Then, it uses obj to create a sound using snd-fromobject. This sound is returned from snd-product. Note that in snd-fromobject, you must also specify the starting time and sample rate as the first two parameters. These are copied from s1, again assuming that s1 and s2 have matching starting times and sample rates.

Note that in more elaborate DSP algorithms we could expect the object to have a number of instance variables to hold things such as previous samples, waveform tables, and other parameters.

6. SAL

Nyquist supports two languages: XLISP and SAL. In some sense, XLISP and SAL are the same language, but with differing syntax. This chapter describes SAL: how it works, SAL syntax and semantics, and the relationship between SAL and XLISP, and differences between Nyquist SAL and Common Music SAL.

Nyquist SAL is based on Rick Taube's SAL language, which is part of Common Music. SAL offers the power of Lisp but features a simple, Algol-like syntax. SAL is implemented in Lisp: Lisp code translates SAL into a Lisp program and uses the underlying Lisp engine to evaluate the program. Aside from the translation time, which is quite fast, SAL programs execute at about the same speed as the corresponding Lisp program. (Nyquist SAL programs run just slightly slower than XLISP because of some runtime debugging support automatically added to user programs by the SAL compiler.)

From the user's perspective, these implementation details are hidden. You can enter SAL mode from XLISP by typing `(SAL)` to the XLISP prompt. The SAL input prompt (`SAL>`) will be displayed. From that point on, you simply type SAL commands, and they will be executed. By setting a preference in the jNyqIDE program, SAL mode will be entered automatically.

It is possible to encounter errors that will take you from the SAL interpreter to an XLISP prompt. In general, the way to get back to SAL is by typing `(top)` to get back to the top level XLISP interpreter and reset the Nyquist environment. Then type `(sal)` to restart the SAL interpreter.

6.1. SAL Syntax and Semantics

The most unusual feature of SAL syntax is that identifiers are Lisp-like, including names such as ‘‘play-file’’ and even ‘‘*warp*.’’ In SAL, most operators must be separated from identifiers by white space. For example, `play-file` is one identifier, but `play - file` is an expression for ‘‘play minus file,’’ where `play` and `file` are two separate identifiers. Fortunately, no spaces are needed around commas and parentheses.

In SAL, whitespace (any sequence of space, newline, or tab characters) is sometimes necessary to separate lexical tokens, but otherwise, spaces and indentation are ignored. To make SAL readable, it is *strongly* advised that you indent SAL programs as in the examples here. The jNyqIDE program is purposely insistent about SAL indentation, so if you use it to edit SAL programs, your indentation should be both beautiful and consistent.

As in Lisp (but very unlike C or Java), comments are indicated by semicolons. Any text from an unquoted semicolon to the end of the line is ignored.

```
; this is a comment
; comments are ignored by the compiler
print "Hello World" ; this is a SAL statement
```

As in Lisp, identifiers are translated to upper-case, making SAL case-insensitive. For example, the function name `autonorm` can be typed in lower case or as `AUTONORM`, `AutoNorm`, or even `AuToNoRm`. All forms denote the same function. The recommended approach is to write programs in all lower case.

SAL is organized around statements, most of which contain expressions. We will begin with expressions and then look at statements.

6.1.1. Expressions

6.1.1.1. Simple Expressions

As in XLISP, simple expressions include:

- integers (FIXNUM's), such as `1215`,
- floats (FLONUM's) such as `12.15`,
- strings (STRING's) such as `"Magna Carta"`, and
- symbols (SYMBOL's) such as `magna-carta`. A symbol with a leading colon (`:`) evaluates to itself as in Lisp. Otherwise, a symbol denotes either a local variable, a formal parameter, or a global variable. As in Lisp, variables do not have data types or type declarations. The type of a variable is determined at runtime by its value.

Additional simple expressions in SAL are:

- lists such as `{c 60 e 64}`. Note that there are no commas to separate list elements, and symbols in lists are not evaluated as variables but stand for themselves. Lists may contain numbers, booleans, symbols, strings, and other lists.
- Booleans: SAL interprets `#t` as true and `#f`as false. (As far as the SAL compiler is concerned, `t` and `nil` are just variables. Since these are the Lisp versions of true and false, they are interchangeable with `#t` and `#f`, respectively.)

A curious property of Lisp and Sal is that *false* and the empty list are the same value. Since SAL is based on Lisp, `#f` and `{}` (the empty list) are equal.

6.1.1.2. Operators

Expressions can be formed with unary and binary operators using infix notation. The operators are:

- `+` - addition, including sounds
- `-` - subtraction, including sounds
- `*` - multiplication, including sounds
- `/` - division (due to divide-by-zero problems, does not operate on sounds)
- `%` - modulus (remainder after division)
- `^` - exponentiation
- `=` - equal (using Lisp `eql`)
- `!=` - not equal
- `>` - greater than
- `<` - less than
- `>=` - greater than or equal
- `<=` - less than or equal
- `~=` - general equality (using Lisp `equal`)
- `&` - logical and
- `|` - logical or
- `!` - logical not (unary)

- @ - time shift
- @@ - time shift to absolute time
- ~ - time stretch
- ~~ - time stretch to absolute stretch factor

Again, remember that operators *must* be delimited from their operands using spaces or parentheses. Operator precedence is based on the following levels of precedence:

```
@ @@ ~ ~~
^
/ *
% - +
~= <= >= > ~= =
!
&
|
```

6.1.1.3. Function Calls

A function call is a function name followed by zero or more comma-delimited argument expressions enclosed within parentheses:

```
list()
piano-note(2.0, c4 + interval, 100)
```

Some functions use named parameters, in which case the name of the argument with a colon precedes the argument expression.

```
s-save(my-snd(), ny:all, "tmp.wav", play: #t, bits: 16)
```

6.1.1.4. Array Notation

An array reference is a variable identifier followed by an index expression in square brackets, e.g.:

```
x[23] + y[i]
```

6.1.1.5. Conditional Values

The special operator `#?` evaluates the first argument expression. If the result is *true*, the second expression is evaluated and its value is returned. If *false*, the third expression is evaluated and returned (or *false* is returned if there is no third expression):

```
#?(random(2) = 0, unison, major-third)
#?(pitch >= c4, pitch - c4) ; returns false if pitch < c4
```

6.1.2. SAL Statements

SAL compiles and evaluates *statements* one at a time. You can type statements at the SAL prompt or load a file containing SAL statements. SAL statements are described below. The syntax is indicated at the beginning of each statement type description: `this font` indicates literal terms such as keywords, *the italic font* indicates a place-holder for some other statement or expression. Bracket [like this] indicate optional (zero or one) syntax elements, while braces with a plus {like this}+ indicate one or more occurrences of a syntax element. Braces with a star {like this}* indicate zero or more occurrences of a syntax element: { *non-terminal* }* is equivalent to [{*non-terminal*}+].

6.1.2.1. begin and end

`begin` [*with-stmt*] {*statement*}+ `end`

A `begin-end` statement consists of a sequence of statements surrounded by the `begin` and `end` keywords. This form is often used for function definitions and after `then` or `else` where the syntax demands a single statement but you want to perform more than one action. Variables may be declared using an optional `with` statement immediately after `begin`. For example:

```
begin
  with db = 12.0,
       linear = db-to-linear(db)
  print db, "dB represents a factor of", linear
  set scale-factor = linear
end
```

6.1.2.2. chdir

`chdir` *expression*

The `chdir` statement changes the working directory. This statement is provided for compatibility with Common Music SAL, but it really should be avoided if you use jNyqIDE. The *expression* following the `chdir` keyword should evaluate to a string that is a directory path name. Note that literal strings themselves are valid expressions.

```
chdir "/Users/rbd/tmp"
```

6.1.2.3. define variable

[`define`] `variable` *name* [= *expression*] {, *name* [= *expression*]}*

Global variables can be declared and initialized. A list of variable names, each with an optional initialization follows the `define variable` keywords. (Since `variable` is a keyword, `define` is redundant and optional in Nyquist SAL, but required in Common Music SAL.) If the initialization part is omitted, the variable is initialized to false. Global variables do not really need to be declared: just using the name implicitly creates the corresponding variable. However, it is an error to use a global variable that has not been initialized; `define variable` is a good way to introduce a variable (or constant) with an initial value into your program.

```
define variable transposition = 2,
                print-debugging-info,  ; initially false
                output-file-name = "salmon.wav"
```

6.1.2.4. define function

[`define`] `function` *name* `(` [*parameter*], {, *parameter*}* `)` *statement*

Before a function be called from an expression (as described above), it must be defined. A function definition gives the function *name*, a list of *parameters*, and a *statement*. When a function is called, the actual parameter expressions are evaluated from left to right and the formal parameters of the function definition are set to these values. Then, *statement* is evaluated.

The formal parameters may be positional parameters that are matched with actual parameters by position from left to right. Syntactically, these are symbols and these symbols are essentially local variables that exist only until *statement* completes or a `return` statement causes the function evaluation to end. As in Lisp, parameters are passed by value, so assigning a new value to a formal parameter has no

effect on the actual value. However, lists and arrays are not copied, so internal changes to a list or array produce observable side effects.

Alternatively, formal parameters may be keyword parameters. Here the *parameter* is actually a pair: a keyword parameter, which is a symbol followed by a colon, and a default value, given by any expression. Within the body of the function, the keyword parameter is named by a symbol whose name matches the keyword parameter except there is no final colon.

```
define function foo(x: 1, y: bar(2, 3))
    display "foo", x, y

exec foo(x: 6, y: 7)
```

In this example, `x` is bound to the value 6 and `y` is bound to the value 7, so the example prints ‘‘`foo : X = 6, Y = 7`’’. Note that while the keyword parameters are `x:` and `y:`, the corresponding variable names in the function body are `x` and `y`, respectively.

The *parameters* are meaningful only within the lexical (static) scope of *statement*. They are not accessible from within other functions even if they are called by this function.

Use a `begin-end` statement if the body of the function should contain more than one statement or you need to define local variables. Use a `return` statement to return a value from the function. If *statement* completes without a `return`, the value false is returned.

6.1.2.5. display

`display` *string* {, *expression*}*

The `display` statement is handy for debugging. At present, it is only implemented in Nyquist SAL. When executed, `display` prints the *string* followed by a colon and then, for each *expression*, the expression and its value are printed, after the last expression, a newline is printed. For example,

```
display "In function foo", bar, baz
```

prints

```
In function foo : bar = 23, baz = 5.3
```

SAL may print the expressions using Lisp syntax, e.g. if the expression is ‘‘bar + baz,’’ do not be surprised if the output is ‘‘`(sum bar baz) = 28.3`.’’

6.1.2.6. exec

`exec` *expression*

Unlike most other programming languages, you cannot simply type an expression as a statement. If you want to evaluate an expression, e.g. call a function, you must use an `exec` statement. The statement simply evaluates the *expression*. For example,

```
exec set-sound-srate(22050.0) ; change default sample rate
```

6.1.2.7. if

`if` *test-expr* `then` *true-stmt* [`else` *false-stmt*]

An `if` statement evaluates the expression *test-expr*. If it is true, it evaluates the statement *true-stmt*. If false, the statement *false-stmt* is evaluated. Use a `begin-end` statement to evaluate more than one

statement in then `then` or `else` parts.

```
if x < 0 then x = -x ; x gets its absoute value

if x > upper-bound then
  begin
    print "x too big, setting to", upper-bound
    x = upper-bound
  end
else
  if x < lower-bound then
    begin
      print "x too small, setting to", lower-bound
      x = lower-bound
    end
```

Notice in this example that the `else` part is another `if` statement. An `if` may also be the `then` part of another `if`, so there could be two possible `if`'s with which to associate an `else`. An `else` clause always associates with the closest previous `if` that does not already have an `else` clause.

6.1.2.8. when

`when` *test statement*

The `when` statement is similar to `if`, but there is no `else` clause.

```
when *debug-flag* print "you are here"
```

6.1.2.9. unless

`unless` *test statement*

The `unless` statement is similar to `when` (and `if`) but the *statement* is executed when the *test* expression is *false*.

```
unless count = 0 set average = sum / count
```

6.1.2.10. load

`load` *expression*

The `load` command loads a file named by *expression*, which must evauate to a string path name for the file. To load a file, SAL interprets each statement in the file, stopping when the end of the file or an error is encountered. If the file ends in `.lsp`, the file is assumed to contain Lisp expressions, which are evaluated by the XLISP interpreter. In general, SAL files should end with the extension `.sal`.

6.1.2.11. loop

`loop` [*with-stmt*] {*stepping*}* {*stopping** *action*+ [*finally*] `end`

The `loop` statement is by far the most complex statement in SAL, but it offers great flexibility for just about any kind of iteration. The basic function of a loop is to repeatedly evaluate a sequence of *action*'s which are statements. Before the loop begins, local variables may be declared in *with-stmt*, a `with` statement.

The *stepping* clauses do several things. They introduce and initialize additional local variables similar to the *with-stmt*. However, these local variables are updated to new values after the *action*'s. In addition,

some *stepping* clauses have associated stopping conditions, which are tested on each iteration *before* evaluating the *action*'s.

There are also *stopping* clauses that provide additional tests to stop the iteration. These are also evaluated and tested on each iteration before evaluating the *action*'s.

When some *stepping* or *stopping* condition causes the iteration to stop, the *finally* clause is evaluated (if present). Local variables and their values can still be accessed in the *finally* clause. After the *finally* clause, the `loop` statement completes.

The *stepping* clauses are the following:

`repeat` *expression*
: Sets the number of iterations to the value of *expression*, which should be an integer (FIXNUM).

`for` *var* = *expression* [`then` *expr2*]
: Introduces a new local variable named *var* and initializes it to *expression*. Before each subsequent iteration, *var* is set to the value of *expr2*. If the `then` part is omitted, *expression* is re-evaluated and assigned to *var* on each subsequent iteration. Note that this differs from a *with-stmt* where expressions are evaluated and variables are only assigned their values once.

`for` *var* `in` *expression*
: Evaluates *expression* to obtain a list and creates a new local variable initialized to the first element of the list. After each iteration, *var* is assigned the next element of the list. Iteration stops when *var* has assumed all values from the list. If the list is initially empty, the loop *action*'s are not evaluated (there are zero iterations).

`for` *var* [`from` *from-expr*] [[`to` | `below` | `downto` | `above`] *to-expr*] [`by` *step-expr*]
: Introduces a new local variable named *var* and intialized to the value of the expression *from-expr* (with a default value of 0). After each iteration of the loop, *var* is incremented by the value of *step-expr* (with a default value of 1). The iteration ends when *var* is greater than the value of *to-expr* if there is a `to` clause, greater than or equal to the value of *to-expr* if there is a `below` clause, less than the value of *to-expr* if there is a `downto` clause, or less than or equal to the value of *to-expr* if there is a `above` clause. (In the cases of *downto* and *above*, the default increment value is -1. If there is no `to`, `below`, `downto`, `above`, or `below` clause, no interation stop test is created for this stepping clause.

The *stopping* clauses are the following:

`while` *expression* The iterations are stopped when *expression* evaluates to *false*. Anything not false is considered to mean true.

`until` *expression* The iterations are stopped when *expression* evaluates to *true*.

The *finally* clause is defined as follows:

`finally` *statement*
: The *statement* is evaluated when one of the *stepping* or *stopping* clauses ends the loop. As always, *statement* may be a `begin-end` statement. If an *action* evaluates a `return` statement, the `finally` statement is not executed.

Loops often fall into common patterns, such as iteratiing a fixed number of times, performing an operation on some range of integers, collecting results in a list, and linearly searching for a solution. These forms are illustrated in the examples below.

```
; iterate 10 times
loop
  repeat 10
  print random(100)
end

; print even numbers from 10 to 20
; note that 20 is printed. On the next iteration,
;   i = 22, so i >= 22, so the loop exits.
loop
  for i from 10 to 22 by 2
  print i
end

; collect even numbers in a list
loop
  with lis
  for i = 0 to 10 by 2
  set lis @= i ; push integers on front of list,
               ; which is much faster than append,
               ; but list is built in reverse
  finally result = reverse(lis)
end
; now, the variable result has a list of evens

; find the first even number in a list
result = #f ; #f means "false"
loop
  for elem in lis
  until evenp(elem)
  finally result = elem
end
; result has first even value in lis (or it is #f)
```

6.1.2.12. print

`print` *expr* {, *expr*}*

The `print` statement prints a newline, then evaluates expressions and prints their values. A blank space is printed after each value.

```
print "The value of x is", x
```

6.1.2.13. return

`return` *expression*

The `return` statement can only be used inside a function. It evaluates *expression* and then the function returns the value of the expression to its caller.

6.1.2.14. set

`set` *var op expression* {, *var op expression*}*

The `set` statement changes the value of a variable *var* according to the operator *op* and the value of the *expression*. The operators are:

`=` The value of *expression* is assigned to *var*.

`+=` The value of *expression* is added to *var*.

`*=` The value of *var* is multiplied by the value of the expression.

`&=` The value of *expression* is inserted as the last element of the list referenced by *var*. If *var* is the empty list (denoted by `#f`), then *var* is assigned a newly constructed list of one element, the value of *expression*.

`^=` The value of *expression*, a list, is appended to the list referenced by *var*. If *var* is the empty list (denoted by `#f`), then *var* is assigned the (list) value of *expression*.

`@=` Pushes the value of *expression* onto the front of the list referenced by *var*. If *var* is empty (denoted by `#f`), then *var* is assigned a newly constructed list of one element, the value of *expression*.

`<=` Sets the new value of *var* to the minimum of the old value of *var* and the value of *expression*.

`>=` Sets the new value of *var* to the maximum of the old value of *var* and the value of *expression*.

; example from Rick Taube's SAL description

```
loop
  with a, b = 0, c = 1, d = {}, e = {}, f = -1, g = 0
  for i below 5
  set a = i, b += 1, c *= 2, d &= i, e @= i, f <= i, g >= i
  finally display "results", a, b, c, d, e, f, g
end
```

6.1.2.15. with

`with` *var* [= *expression*] {, *var* [= *expression*]}*

The `with` statement declares and initializes local variables. It can appear only after `begin` or `loop`. If the *expression* is omitted, the initial value is *false*. The variables are visible only inside the `begin-end` or `loop` statement where the `with` statement appears. Even in `loop`'s the variables are intialized only when the loop is entered, not on each iteration.

6.1.2.16. exit

`exit` [`nyquist`]

The `exit` statement is unique to Nyquist SAL. It returns from SAL mode to the XLISP interpreter. (Return to SAL mode by typing ‘‘`(sal)`’’). If `nyquist` is included in the statement, then the entire Nyquist process will exit.

6.2. Interoperability of SAL and XLISP

When SAL evaluatas command or loads files, it translates SAL into XLISP. You can think of SAL as a program that translates everything you write into XLISP and entering it for you. Thus, when you define a SAL function, the function actually exists as an XLISP function (created using Lisp's `defun` special form). When you set or evaluate global variables in SAL, these are exactly the same Lisp global variables. Thus, XLISP functions can call SAL functions and vice-versa. At run time, everything is Lisp.

6.2.1. Function Calls

In general, there is a very simple translation from SAL to Lisp syntax and back. A function call is SAL, for example,

```
osc(g4, 2.0)
```

is translated to Lisp by moving the open parenthesis in front of the function name and removing the commas:

```
(osc g4 2.0)
```

Similarly, if you want to translate a Lisp function call to SAL, just reverse the translation.

6.2.2. Symbols and Functions

SAL translates keywords with trailing colons (such as `foo:`) into Lisp keywords with leading colons (such as `:foo`), but SAL keywords are not treated as expressions as they are in Lisp. You cannot write `open("myfile.txt", direction: output:)` because SAL expects an expression after direction. A special form `keyword` is defined to generate a Lisp keyword as an expression. The argument is the keyword *without* a colon, e.g. `open("myfile.txt", direction: keyword(output))`. Alternatively, you can write the Lisp-style keyword with the leading colon, e.g. `open("myfile.txt", direction: :output)`.

In Nyquist SAL, the hash character (#), can be used as a prefix to a Lisp function name. For example, the following command is not legal because `print` is a SAL command name, not a legal function name: `set v = append(print(a), print(b))`. (Here the intent is to print arguments to append). However, you can use the hash character to access the Lisp `print` function: `set v = append(#print(a), #print(b))`.

6.2.3. Playing Tricks On the SAL Compiler

In many cases, the close coupling between SAL and XLISP gives SAL unexpected expressive power. A good example is `seqrep`. This is a special looping construct in Nyquist, implemented as a macro in XLISP. In Lisp, you would write something like:

```
(seqrep (i 10) (pluck c4))
```

One might expect SAL would have to define a special `seqrep` statement to express this, but since statements do not return values, this approach would be problematic. The solution (which is already fully implemented in Nyquist) is to define a new macro `sal-seqrep` that is equivalent to `seqrep` except that it is called as follows:

```
(sal-seqrep i 10 (pluck c4))
```

The SAL compiler automatically translates the identifier `seqrep` to `sal-seqrep`. Now, in SAL, you can just write

```
seqrep(i, 10, pluck(c4))
```

which is translated in a pretty much semantics-unaware fashion to

```
(sal-seqrep i 10 (pluck c4))
```

and viola!, we have Nyquist control constructs in SAL even though SAL is completely unaware that `seqrep` is actually a special form.

7. 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 1.7).

7.1. 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.

7.1.1. 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.

7.1.2. 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.

7.1.3. Accessing and Creating Sound

Several functions display information concerning a sound and can be used to query the components of a sound. There are functions that access samples in a sound and functions that construct sounds from samples.

`sref(`*sound*`, `*time*`)`

Accesses *sound* at the point *time*, which is a local 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*): $t=(n/srate)+t0$ Thus, the lisp code to access the n^{th} sample of a sound would look like: `(sref sound (global-to-local (+ (/ n (snd-srate sound)) (snd-t0 sound))))` Here is why `sref` interprets its time argument as a local time:

```
> (sref (ramp 1) 0.5) ; evaluate a ramp at time 0.5
0.5
> (at 2.0 (sref (ramp 1) 0.5)) ; ramp is shifted to start at 2.0
                               ; the time, 0.5, is shifted to 2.5
0.5
```

If you were to use `snd-sref`, which treats time as global, instead of `sref`, which treats time as local, then the first example above would return the same answer (0.5), but the second example would return 0. Why? Because the `(ramp 1)` behavior would be shifted to start at time 2.0, but the resulting sound would be evaluated at global time 0.5. By definition, sounds have a value of zero before their start time.

`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 `FLONUM`s 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-fromarraystream(`*t0*`, `*sr*`, `*object*`)`

Creates a sound for which samples come from *object*. The starting time is *t0* (a `FLONUM`), and the

[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.

sample rate is *sr*. The *object* is an XLISP object (see Section IV.11 for information on objects.) A sound is returned. When the sound needs samples, they are generated by sending the message `:next` to *object*. If *object* returns `NIL`, the sound terminates. Otherwise, *object* must return an array of `FLONUM`s. The values in these arrays are concatenated to form the samples of the resulting sound. There is no provision for *object* to specify the logical stop time of the sound, so the logical stop time is the termination time.

`snd-fromobject(`*t0*`, `*sr*`, `*object*`)`
Creates a sound for which samples come from *object*. The starting time is *t0* (a `FLONUM`), and the sample rate is *sr*. The *object* is an XLISP object (see Section IV.11 for information on objects. A sound is returned. When the sound needs samples, they are generated by sending the message `:next` to *object*. If *object* returns `NIL`, the sound terminates. Otherwise, *object* must return a `FLONUM`. There is no provision for *object* to specify the logical stop time of the sound, so the logical stop time is the termination time.

`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-fetch(`*sound*`)`
Reads samples sequentially from *sound*. This returns a `FLONUM` after each call, or `NIL` when *sound* terminates. ***Note:*** `snd-fetch` modifies *sound*; it is strongly recommended to copy *sound* using `snd-copy` and access only the copy with `snd-fetch`.

`snd-fetch-array(`*sound*`, `*len*`, `*step*`)`
Reads sequential arrays of samples from *sound*, returning either an array of `FLONUM`s or `NIL` when the sound terminates. The *len* parameter, a `FIXNUM`, indicates how many samples should be returned in the result array. After the array is returned, *sound* is modified by skipping over *step* (a `FIXNUM`) samples. If *step* equals *len*, then every sample is returned once. If *step* is less than *len*, each returned array will overlap the previous one, so some samples will be returned more than once. If *step* is greater than *len*, then some samples will be skipped and not returned in any array. The *step* and *len* may change at each call, but in the current implementation, an internal buffer is allocated for *sound* on the first call, so subsequent calls may not specify a greater *len* than the first. ***Note:*** `snd-fetch-array` modifies *sound*; it is strongly recommended to copy *sound* using `snd-copy` and access only the copy with `snd-fetch-array`.

`snd-flatten(`*sound*`, `*maxlen*`)`
This function is identical to `snd-length`. You would use this function to force samples to be computed in memory. Normally, this is not a good thing to do, but here is one appropriate use: In the case of sounds intended for wavetables, the unevaluated sound may be larger than the evaluated (and typically short) one. Calling `snd-flatten` will compute the samples and allow the unit generators to be freed in the next garbage collection. ***Note:*** If a sound is computed from many instances of table-lookup oscillators, calling `snd-flatten` will free the oscillators and their tables. Calling `(stats)` will print how many total bytes have been allocated to tables.

`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 `peak`, a replacement (page 82).

`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. Originally, this was a placeholder for a facility to play samples directly to an audio output device, but playback is now accomplished by `s-save`. 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 7.5) for saving samples to a file, and `play` (Section 7.5) 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-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-sref(`*sound*`, `*time*`)`
Evaluates *sound* at the global time given by *time*. Safe for ordinary use, but normally, you should call `sref` 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. This is the only way to find out how much memory is being used by table-lookup oscillator instances.

7.1.4. 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*.

`follow(`*sound*`, `*floor*`, `*risetime*`, `*falltime*`, `*lookahead*`)`

An envelope follower intended as a commponent for compressor and limiter functions. The basic goal of this function is to generate a smooth signal that rides on the peaks of the input signal. The usual objective is to produce an amplitude envelope given a low-sample rate (control rate) signal representing local RMS measurements. The first argument is the input signal. The *floor* is the minimum output value. The *risetime* is the time (in seconds) it takes for the output to rise (exponentially) from *floor* to unity (1.0) and the *falltime* is the time it takes for the output to fall (exponentially) from unity to *floor*. The algorithm looks ahead for peaks and will begin to increase the output signal according to *risetime* in anticipation of a peak. The amount of anticipation (in seconds) is given by *lookahead*. The algorithm is as follows: the output value is allowed to increase according to *risetime* or decrease according to *falltime*. If the next input sample is in this range, that sample is simply output as the next output sample. If the next input sample is too large, the algorithm goes back in time as far as necessary to compute an envelope that rises according to *risetime* to meet the new value. The algorithm will only work backward as far as *lookahead*. If that is not far enough, then there is a final forward pass computing a rising signal from the earliest output sample. In this case, the output signal will be at least momentarily less than the input signal and will continue to rise exponentially until it intersects the input signal. If the input signal falls faster than indicated by *falltime*, the output fall rate will be limited by *falltime*, and the fall in output will stop when the output reaches *floor*. This algorithm can make two passes througth the buffer on sharply rising inputs, so it is not particularly fast. With short buffers and low sample rates this should not matter. See `snd-avg` for a function that can help to generate a low-sample-rate input for `follow`. See `snd-chase` in Section 7.6.3 for a related filter.

`gate(`*sound*`, `*floor*`, `*risetime*`, `*falltime*`, `*lookahead*`, `*threshold*`)`

Generate an exponential rise and decay intended for noise gate implementation. The decay starts when the signal drops below *threshold* and stays there for longer than *lookahead* (a `FLONUM` in seconds). (The signal begins to drop when the signal crosses *threshold*, not after *lookahead*.) Decay continues until the value reaches *floor* (a `FLONUM`), at which point the decay stops and the output value is held constant. Either during the decay or after the floor is reached, if the signal goes above *threshold*, then the ouptut value will rise to unity (1.0) at the point the signal crosses the threshold. Because of internal lookahead, the signal actually begins to rise before the signal crosses *threshold*. The rise is a constant-rate exponential and set so that a rise from *floor* to unity occurs in *risetime*. Similary, the fall is a constant-rate exponential such that a fall from unity to *floor* takes *falltime*.

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

Returns a step number for *freq* (in hz), which can be either a number of a `SOUND`. The result has the same type as the argument. See also `step-to-hz` (below).

`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 7.3).

`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 7.3).

`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 step number or a `SOUND` type representing a time-varying step number. The result is a `FLONUM` if *pitch* is a number, and a `SOUND` if *pitch* is a `SOUND`. See also `hz-to-step` (above).

`get-duration(`*dur*`)`
Gets the actual duration of of something starting at a local time of 0 and ending at a local time of *dur* times the current sustain. For convenience, `*rslt*` is set to the global time corresponding to local time zero.

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

`osc-enable(`*flag*`)`
Enable or disable Open Sound Control. (See Appendix II.) Enabling creates a socket and a

service that listens for UDP packets on port 7770. Currently, only two messages are accepted by Nyquist. The first is of the form `/slider` with an integer index and a floating point value. These set internal slider values accessed by the `snd-slider` function. The second is of the form `/wii/orientation` with two floating point values. This message is a special case to support the DarwiinRemoteOsc program which can relay data from a Nintendo WiiMote device to Nyquist via OSC. The two orientation values control sliders 0 and 1. Disabling terminates the service (polling for messages) and closes the socket. The *previous* state of enablement is returned, e.g. if OSC is enabled and *flag* is *nil*, OSC is disabled and `T` (true) is returned because OSC was enabled at the time of the call. This function only exists if Nyquist is compiled with the compiler flag `OSC`. Otherwise, the function exists but always returns the symbol `DISABLED`. Consider lowering the audio latency using `snd-set-latency`. *Warning:* there is the potential for network-based attacks using OSC. It is tempting to add the ability to evaluate XLISP expressions sent via OSC, but this would create unlimited and unprotected access to OSC clients. For now, it is unlikely that an attacker could do more than manipulate slider values.

`snd-set-latency(`*latency*`)`
Set the latency requested when Nyquist plays sound to *latency*, a `FLONUM`. The previous value is returned. The default is 0.3 seconds. To avoid glitches, the latency should be greater than the time required for garbage collection and message printing and any other system activity external to Nyquist.

7.2. Behaviors

7.2.1. 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, samples from which are coerced to the current default `*sound-srate*` sample rate.

`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.

7.2.2. Sound Synthesis

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

`const(`*value*`[ ,`*duration*`])`
Creates a constant function at the `*control-srate*`. Every sample has the given *value*, and the default *duration* is 1.0. See also `s-rest`, which is equivalent to calling `const` with zero, and note that you can pass scalar constants (numbers) to `sim`, `sum`, and `mult` where they are handled more efficiently than constant functions.

`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 $t_1 + t_2 + 2\text{ms} + t_4$ 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 `*control-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.

`exp-dec(`*hold*`, `*halfdec*`, `*length*`)`
This convenient envelope shape is a special case of `pwev` (see Section 7.2.2.2). The envelope starts at 1 and is constant for *hold* seconds. It then decays with a half life of *halfdec* seconds until *length*. (The total duration is *length*.) In other words, the amplitude falls by half each *halfdec* seconds. When stretched, this envelope scales linearly, which means the hold time increases and the half decay time increases.

`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. See also `resample`.

`lfo(`*freq*`[ ,`*duration*`, `*table*`, `*phase*`])`
Just like `osc` (below) except this computes at the `*control-srate*` and frequency is specified in Hz. Phase is specified in degrees. 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.

`fmlfo(`*freq*`[ ,`*table*`, `*phase*`])`
A low-frequency oscillator that computes at the `*control-srate*` using a sound to specify a time-varying frequency in Hz. Phase is a `FLONUM` in degrees. The duration of the result is determined by *freq*.

`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 7 for an example.

`control-warp(`*warp-fn*`, `*signal*`, [`*wrate*`])`
Applies a warp function *warp-fn* to *signal* using function composition. If *wrate* is omitted, 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 `sound-warp` for an explanation of *wrate* and high-quality warping.

`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.

`prod(`*beh*$_1$`,` *beh*$_2$`, ...)`
Same as `mult`.

`resample(`*sound*`,` *srate*`)`
Similar to `force-srate`, except high-quality interpolation is used to prefilter and reconstruct the signal at the new sample rate. Also, the result is scaled by 0.95 to reduce problems with clipping. (See also `sound-warp`.)

`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 7.6.2).

`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 7.6.2).

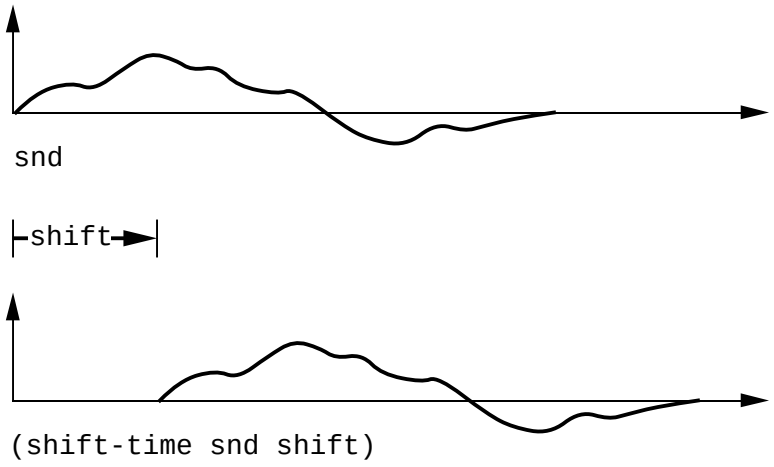

Figure 5: The `shift-time` function shifts a sound in time according to its *shift* argument.

`sound-warp(`*warp-fn*`,` *signal*`[ ,`*wrate*`])`
Applies a warp function *warp-fn* to *signal* using function composition. If the optional parameter *wrate* is omitted or NIL, linear interpolation is used. Otherwise, high-quality sample interpolation is used, and the result is scaled by 0.95 to reduce problems with clipping

(interpolated samples can exceed the peak values of the input samples.) *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`.

If *wrate* is not NIL, it must be a number. The parameter indicates that high-quality resampling should be used and specifies the sample rate for the inverse of *warp-fn*. Use the lowest number you can. (See below for details.) Note that high-quality resampling is much slower than linear interpolation.

To perform high-quality resampling by a fixed ratio, as opposed to a variable ratio allowed in `sound-warp`, use `scale-srate` to stretch or shrink the sound, and then `resample` to restore the original sample rate.

`Sound-warp` and `control-warp` both take the inverse of *warp-fn* to get a function from real time to score time. Each sample of this inverse is thus a score time; *signal* is evaluated at each of these score times to yield a value, which is the desired result. The sample rate of the inverse warp function is somewhat arbitrary. With linear interpolation, the inverse warp function sample rate is taken to be the output sample rate. Note, however, that the samples of the inverse warp function are stored as 32-bit floats, so they have limited precision. Since these floats represent sample times, rounding can be a problem. Rounding in this case is equivalent to adding jitter to the sample times. Nyquist ignores this problem for ordinary warping, but for high-quality warping, the jitter cannot be ignored.

The solution is to use a rather low sample rate for the inverse warp function. `Sound-warp` can then linearly interpolate this signal using double-precision floats to minimize jitter between samples. The sample rate is a compromise: a low sample rate minimizes jitter, while a high sample rate does a better job of capturing detail (e.g. rapid fluctuations) in the warp function. A good rule of thumb is to use at most 1,000 to 10,000 samples for the inverse warp function. For example, if the result will be 1 minute of sound, use a sample rate of 3000 samples / 60 seconds = 50 samples/second. Because Nyquist has no advance information about the warp function, the inverse warp function sample rate must be provided as a parameter. When in doubt, just try something and let your ears be the judge.

`integrate(`*signal*`)`
Computes the integral of *signal*. The start time, sample rate, etc. are taken from *signal*.

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

7.2.2.1. Oscillators

`osc(`*pitch*`[, `*duration*`, `*table*`, `*phase*`])`
Returns a sound which is the *table* oscillated at *pitch* for the given *duration*, starting with the *phase* (in degrees). Defaults are: *duration* `1.0` (second), *table* `*table*`, *phase* `0.0`. The default value of `*table*` is a sinusoid. 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. ***Note 3:*** When `osc` is called, memory is allocated for the table, and samples are copied from the sound (the first element of the list which is the *table* parameter) to the memory. Every instance of `osc` has a private copy of the table, so the total storage can become large in some cases, for example in granular synthesis with many instances of `osc`. In some cases, it may make sense to use `snd-flatten` (see Section 7.1.3) to cause the sound to be fully realized, after which the `osc` and its table memory can be reclaimed by garbage collection. The `partial` function (see below) does not need a private table and does not use much space.

`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`.

`hzosc(`*hz*`[ ,`*table*`, `*phase*`])`
Returns a sound which is the *table* oscillated at *hz* starting at *phase* degrees. The default *table* is `*table*` and the default *phase* is *0.0*. The default duration is `1.0`, but this is stretched as in `osc` (see above). The *hz* parameter may be a `SOUND`, in which case the duration of the result is the duration of *hz*. The sample rate is `*sound-srate*`.

`osc-saw(`*hz*`)`
Returns a sawtooth waveshape at the indicated frequency (in Hertz). The sample rate is `*sound-srate*`. The *hz* parameter may be a sound as in *hzosc* (see above).

`osc-tri(`*hz*`)`
Returns a triangle waveshape at the indicated frequency (in Hertz). The sample rate is `*sound-srate*`. The *hz* parameter may be a sound as in *hzosc* (see above).

`osc-pulse(`*hz*`, `*bias*`[ ,`*compare-shape*`])`
Returns a square pulse with variable width at the indicated frequency (in Hertz). The *bias* parameter controls the pulse width and should be between `-1` and `+1`, giving a pulse width from 0% (always at `-1`) to 100% (always at `+1`). When bias is zero, a square wave is generated. Bias may be a `SOUND` to create varying pulse width. If bias changes rapidly, strange effects may occur. The optional *compare-shape* defaults to a hard step at zero, but other shapes may be used to achieve non-square pulses. The `osc-pulse` behavior is written in terms of other behaviors and defined in the file `nyquist.lsp` using just a few lines of code. Read the code for the complete story.

`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 degrees) 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 degrees) 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*`.

`fmfb(`*pitch*`, `*index*`[ ,`*dur*`])`

Returns a sound generated by feedback FM synthesis. The *pitch* parameter (given in the usual half-step units) controls the fundamental frequency. The *index* is the amount of feedback, which may be a SOUND or a FLONUM. If *index* is a FLONUM, *dur* must be provided (a FLONUM) to specify the duration. Otherwise, *dur* is ignored if present and the duration is determined by that of *index*. The sample rate is *sound-srate*. A sinusoid table is used. If *index* is below 1.1, this generates a sawtooth-like waveform.

buzz(*n, pitch, modulation*)
Returns a sound with *n* harmonics of equal amplitude and a total amplitude of 1.0, using a well-known function of two cosines. If *n* (an integer) is less than 1, it is set to 1. Aliasing will occur if *n* is too large. The duration is determined by the duration of the sound *modulation*, which is a frequency modulation term expressed in Hz (see Section 7.2.2.1). Negative frequencies are correctly handled. The sample rate is *sound-srate*.

pluck(*pitch*[,*duration*] [, *final-amplitude*])
Returns a sound at the given *pitch* created using a modified Karplus-Strong plucked string algorithm. The tone decays from an amplitude of about 1.0 to about *final-amplitude* in *duration* seconds. The default values are to decay to 0.001 (-60dB) in 1 second. The sample rate is *sound-srate*.

siosc(*pitch, modulation, tables*)
Returns a sound constructed by interpolating through a succession of periodic waveforms. The frequency is given (in half steps) by *pitch* to which a *modulation* signal (in hz) is added, exactly as in fmosc. The *tables* specify a list of waveforms as follows: (*table0 time1 table2 ... timeN tableN*), where each *table* is a sound representing one period. Each *time* is a time interval measured from the starting time. The time is scaled by the nominal duration (computed using (local-to-global (get-sustain))) to get the actual time. Note that this implies linear stretching rather than continuous timewarping of the interpolation or the breakpoints. The waveform is *table0* at the starting time, *table1* after *time1* (scaled as described), and so on. The duration and logical stop time is given by *modulation*. If *modulation* is shorter than *timeN*, then the full sequence of waveforms is not used. If *modulation* is longer than *timeN*, *tableN* is used after *timeN* without further interpolation.

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. 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. If the loop point is not in the bounds of the sound, it is set to zero. 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. It is an error to modulate such that the frequency is negative. Note also that the loop point may be fractional. The sample rate is *sound-srate*.

7.2.2.2. 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.

7.2.2.3. Filter Behaviors

`alpass(`*sound*`,` *decay*`,` *hz*`[ ,`*minhz*`])`

Applies an all-pass filter to *sound*. This all-pass filter creates a delay effect without the resonances of a comb filter. The decay time of the filter is given by *decay*. The *hz* parameter must be a number or sound 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, 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*. If *hz* is of type `SOUND`, the delay may be time-varying. Linear interpolation is then used for fractional sample delay, but it should be noted that linear interpolation implies a low-pass transfer function. Thus, this filter may behave differently with a constant `SOUND` than it does with a `FLONUM` value for *hz*. In addition, if *hz* is of type `SOUND`, then *minhz* is required. The *hz* parameter will be clipped to be greater than *minhz*, placing an upper bound on the delay buffer length.

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

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*.

`congen(`*gate*`,` *risetime*`,` *falltime*`)`

Implements an analog synthesizer-style contour generator. The input *gate* normally goes from 0.0 to 1.0 to create an attack and from 1.0 to 0.0 to start a release. During the attack (output is increasing), the output converges half-way to *gate* in *risetime* (a `FLONUM`) seconds. During the decay, the half-time is *falltime* seconds. The sample rate, start time, logical stop, and terminate time all come from *gate*. If you want a nice decay, be sure that the *gate* goes to zero and stays there for awhile before *gate* terminates, because `congen` (and all Nyquist sounds) go immediately to zero at termination time. For example, you can use `pwl` to build a pulse followed by some zero time:

```
(pwl 0 1 duty 1 duty 0 1)
```

Assuming *duty* is less than 1.0, this will be a pulse of duration *duty* followed by zero for a total duration of 1.0.

```
(congen (pwl 0 1 duty 1 duty 0 1) 0.01 0.05)
```

will have a duration of 1.0 because that is the termination time of the `pwl` input. The decaying

release of the resulting envelope will be truncated to zero at time 1.0. (Since the decay is theoretically infinite, there is no way to avoid truncation, although you could multiply by another envelope that smoothly truncates to zero in the last millisecond or two to get both an exponential decay and a smooth final transition to zero.)

`convolve(`*sound, response*`)`
Convolves two signals. The first can be any length, but the computation time per sample and the total space required are proportional to the length of *response*.

`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*`)`
No longer defined; use `lp` instead, or define it by adding `(setfn tone lp)` to your program.

`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*`)`
No longer defined; use `hp` instead, or define it by adding `(setfn atone hp)` to your program.

`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*.

One application of `reson` is to simulate resonances in the human vocal tract. See `demos/voice_synthesis.htm`for sample code and documentation.

`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* + clip(*signal*(*t*))

where 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.

Further discussion and examples can be found in `demos/distortion.htm`. The `shape` function is also used to map frequency to amplitude to achieve a spectral envelope for Shepard tones in `demos/shepard.lsp`.

`biquad(`*signal*`, `*b0*`, `*b1*`, `*b2*`, `*a0*`, `*a1*`, `*a2*`)`
A fixed-parameter biquad filter. All filter coefficients are `FLONUM`s. See also `lowpass2`, `highpass2`, `bandpass2`, `notch2`, `allpass2`, `eq-lowshelf`, `eq-highshelf`, `eq-band`, `lowpass4`, `lowpass6`, `highpass4`, and `highpass8` in this section for convenient variations based on the same filter. The equations for the filter are: $z_n = s_n + a1 * z_{n-1} + a2 * z_{n-2}$, and $y_n = z_n * b0 + z_{n-1} * b1 + z_{n-2} * b2$.

`biquad-m(`*signal*`, `*b0*`, `*b1*`, `*b2*`, `*a0*`, `*a1*`, `*a2*`)`
A fixed-parameter biquad filter with Matlab sign conventions for *a0*, *a1*, and *a2*. All filter coefficients are `FLONUM`s.

`lowpass2(`*signal*`, `*hz*`[ ,`*q*`])`
A fixed-parameter, second-order lowpass filter based on `snd-biquad`. The cutoff frequency is given by *hz* (a `FLONUM`) and an optional Q factor is given by *q* (a `FLONUM`).

`highpass2(`*signal*`, `*hz*`[ ,`*q*`])`
A fixed-parameter, second-order highpass filter based on `snd-biquad`. The cutoff frequency is given by *hz* (a `FLONUM`) and an optional Q factor is given by *q* (a `FLONUM`).

`bandpass2(`*signal*`, `*hz*`[ ,`*q*`])`
A fixed-parameter, second-order bandpass filter based on `snd-biquad`. The center frequency is given by *hz* (a `FLONUM`) and an optional Q factor is given by *q* (a `FLONUM`).

`notch2(`*signal*`, `*hz*`[ ,`*q*`])`
A fixed-parameter, second-order notch filter based on `snd-biquad`. The center frequency is given by *hz* (a `FLONUM`) and an optional Q factor is given by *q* (a `FLONUM`).

`allpass2(`*signal*`, `*hz*`[ ,`*q*`])`
A fixed-parameter, second-order allpass filter based on `snd-biquad`. The frequency is given by *hz* (a `FLONUM`) and an optional Q factor is given by *q* (a `FLONUM`).

`eq-lowshelf(`*signal*`, `*hz*`, `*gain*`[ ,`*slope*`])`
A fixed-parameter, second-order bass shelving equalization (EQ) filter based on `snd-biquad`. The *hz* parameter (a `FLONUM`)is the halfway point in the transition, and *gain* (a `FLONUM`) is the bass boost (or cut) in dB. The optional *slope* (a `FLONUM`) is 1.0 by default, and response becomes peaky at values greater than 1.0.

`eq-highshelf(`*signal*`, `*hz*`, `*gain*`[ ,`*slope*`])`
A fixed-parameter, second-order treble shelving equalization (EQ) filter based on `snd-biquad`. The *hz* parameter (a `FLONUM`)is the halfway point in the transition, and *gain* (a `FLONUM`) is the treble boost (or cut) in dB. The optional *slope* (a `FLONUM`) is 1.0 by default, and response becomes peaky at values greater than 1.0.

`eq-band(`*signal*`, `*hz*`, `*gain*`, `*width*`)`
A fixed- or variable-parameter, second-order midrange equalization (EQ) filter based on `snd-biquad`, `snd-eqbandcv` and `snd-eqbandvvv`. The *hz* parameter (a `FLONUM`) is the center frequency, *gain* (a `FLONUM`) is the boost (or cut) in dB, and *width* (a `FLONUM`) is the half-gain width in octaves. Alternatively, *hz*, *gain*, and *width* may be `SOUND`s, but they must all have the same sample rate, e.g. they should all run at the control rate or at the sample rate.

`lowpass4(`*signal*`, `*hz*`)`

A four-pole Butterworth lowpass filter. The cutoff frequency is *hz* (a FLONUM).

lowpass6(*signal*, *hz*)
A six-pole Butterworth lowpass filter. The cutoff frequency is *hz* (a FLONUM).

lowpass8(*signal*, *hz*)
An eight-pole Butterworth lowpass filter. The cutoff frequency is *hz* (a FLONUM).

highpass4(*signal*, *hz*)
A four-pole Butterworth highpass filter. The cutoff frequency is *hz* (a FLONUM).

highpass6(*signal*, *hz*)
A six-pole Butterworth highpass filter. The cutoff frequency is *hz* (a FLONUM).

highpass8(*signal*, *hz*)
An eight-pole Butterworth highpass filter. The cutoff frequency is *hz* (a FLONUM).

tapv(*sound*, *offset*, *vardelay*, *maxdelay*)
A delay line with a variable position tap. Identical to snd-tapv. See it for details (7.6.2).

7.2.2.4. Effects

nrev(*sound*, *decay*, *mix*)

jcrev(*sound*, *decay*, *mix*)

prcrev(*sound*, *decay*, *mix*) These reverbs (nrev, jcrev, and prcrev) are implemented in STK (running within Nyquist). nrev derives from Common Music's NRev, which consists of 6 comb filters followed by 3 allpass filters, a lowpass filter, and another allpass in series followed by two allpass filters in parallel. jcrev is the John Chowning reverberator which is based on the use of networks of simple allpass and comb delay filters. This reverb implements three series allpass units, followed by four parallel comb filters, and two decorrelation delay lines in parallel at the output. prcrev is a Perry Cook's reverberator which is based on the Chowning/Moorer/Schroeder reverberators using networks of simple allpass and comb delay filters. This one implements two series allpass units and two parallel comb filters. The *sound* input may be single or multi-channel. The *decay* time is in seconds, and *mix* sets the mixture of input sound reverb sound, where 0.0 means input only (dry) and 1.0 means reverb only (wet).

stkchorus(*sound*, *depth*, *freq*, *mix*[,*delay*])
Chorus implemented in STK. The input *sound* can be single or multi-channel. The FLONUM parameters *depth* and *freq* set the modulation depth from 0 to 1 and modulation frequency (in Hz), and *mix* sets the mixture of input sound and chorused sound, where 0.0 means input sound only (dry) and 1.0 means chorused sound only (wet). The parameter *delay* is a FIXNUM representing the median desired delay length in samples.

pitshift(*sound*, *shift*, *mix*)
A pitch shifter implemented in STK. The input *sound*, a single-channel or multi-channel SOUND is pitch-shifted by *shift*, a FLONUM ratio. A value of 1.0 means no shift. The parameter *mix* sets the mixture of input and shifted sounds. A value of 0.0 means input only (dry) and a value of 1.0 means shifted sound only (wet).

7.2.2.5. Physical Models

clarinet(*step*, *breath-env*)
A physical model of a clarinet from STK. The *step* parameter is a FLONUM that controls the tube length, and the *breath-env* (a SOUND) controls the air pressure and also determines the length of the resulting sound. The *breath-env* signal should range from zero to one.

clarinet-freq(*step*, *breath-env*, *freq-env*)
A variation of clarinet that includes a variable frequency control, *freq-env*, which specifies

frequency deviation in Hz. The duration of the resulting sound is the minimum duration of *breath-env* and *freq-env*. These parameters may be of type FLONUM or SOUND. FLONUMs are coerced into SOUNDs with a nominal duration arbitrarily set to 30.

clarinet-all(*step*, *breath-env*, *freq-env*, *vibrato-freq*, *vibrato-gain*, *reed-stiffness*, *noise*)
A variation of clarinet-freq that includes controls *vibrato-freq* (a FLONUM for vibrato frequency in Hertz), *vibrato-gain* (a FLONUM for the amount of amplitude vibrato), *reed-stiffness* (a FLONUM or SOUND controlling reed stiffness in the clarinet model), and *noise* (a FLONUM or SOUND controlling noise amplitude in the input air pressure). The *vibrato-gain* is a number from zero to one, where zero indicates no vibrato, and one indicates a plus/minus 50% change in breath envelope values. Similarly, the *noise* parameter ranges from zero to one where zero means no noise and one means white noise with a peak amplitude of plus/minus 40% of the *breath-env*. The *reed-stiffness* parameter varies from zero to one. The duration of the resulting sound is the minimum duration of *breath-env*, *freq-env*, *reed-stiffness*, and *noise*. As with clarinet-freq, these parameters may be either FLONUMs or SOUNDs, and FLONUMs are coerced to sounds with a nominal duration of 30.

sax(*step*, *breath-env*)
A physical model of a sax from STK. The *step* parameter is a FLONUM that controls the tube length, and the *breath-env* controls the air pressure and also determines the length of the resulting sound. The *breath-env* signal should range from zero to one.

sax-freq(*step*, *breath-env*, *freq-env*)
A variation of sax that includes a variable frequency control, *freq-env*, which specifies frequency deviation in Hz. The duration of the resulting sound is the minimum duration of *breath-env* and *freq-env*. These parameters may be of type FLONUM or SOUND. FLONUMs are coerced into SOUNDs with a nominal duration arbitrarily set to 30.

sax-all(*step*, *breath-env*, *freq-env*, *vibrato-freq*, *vibrato-gain*, *reed-stiffness*, *noise*, *blow-pos*, *reed-table-offset*)
A variation of sax-freq that includes controls *vibrato-freq* (a FLONUM for vibrato frequency in Hertz), *vibrato-gain* (a FLONUM for the amount of amplitude vibrato), *reed-stiffness* (a SOUND controlling reed stiffness in the sax model), *noise* (a SOUND controlling noise amplitude in the input air pressure), *blow-pos* (a SOUND controlling the point of excitation of the air column), and *reed-table-offset* (a SOUND controlling a parameter of the reed model). The *vibrato-gain* is a number from zero to one, where zero indicates no vibrato, and one indicates a plus/minus 50% change in breath envelope values. Similarly, the *noise* parameter ranges from zero to one where zero means no noise and one means white noise with a peak amplitude of plus/minus 40% of the *breath-env*. The *reed-stiffness*, *blow-pos*, and *reed-table-offset* parameters all vary from zero to one. The duration of the resulting sound is the minimum duration of *breath-env*, *freq-env*, *reed-stiffness*, *noise*, *breath-env*, *blow-pos*, and *reed-table-offset*. As with sax-freq, these parameters may be either FLONUMs or SOUNDs, and FLONUMs are coerced to sounds with a nominal duration of 30.

flute(*step*, *breath-env*)
A physical model of a flute from STK. The *step* parameter is a FLONUM that controls the tube length, and the *breath-env* controls the air pressure and also determines the starting time and length of the resulting sound. The *breath-env* signal should range from zero to one.

flute-freq(*step*, *breath-env*, *freq-env*)
A variation of flute that includes a variable frequency control, *freq-env*, which specifies frequency deviation in Hz. The duration of the resulting sound is the minimum duration of *breath-env* and *freq-env*. These parameters may be of type FLONUM or SOUND. FLONUMs are coerced into SOUNDs with a nominal duration arbitrary set to 30.

flute-all(*step*, *breath-env*, *freq-env*, *vibrato-freq*, *vibrato-gain*, *jet-delay*, *noise*)
A variation of clarinet-freq that includes controls *vibrato-freq* (a FLONUM for vibrato frequency in Hz), *vibrato-gain* (a FLONUM for the amount of amplitude vibrato), *jet-delay* (a

FLONUM or SOUND controlling jet delay in the flute model), and noise (a FLONUM or SOUND controlling noise amplitude in the input air pressure). The *vibrato-gain* is a number from zero to one where zero means no vibrato, and one indicates a plus/minus 50% change in breath envelope values. Similarly, the *noise* parameter ranges from zero to one, where zero means no noise and one means white noise with a peak amplitude of plus/minus 40% of the *breath-env*. The *jet-delay* is a ratio that controls a delay length from the flute model, and therefore it changes the pitch of the resulting sound. A value of 0.5 will maintain the pitch indicated by the step parameter. The duration of the resulting sound is the minimum duration of *breath-env*, *freq-env*, *jet-delay*, and *noise*. These parameters may be either FLONUMs or SOUNDs, and FLONUMs are coerced to sounds with a nominal duration of 30.

bowed(*step*, *bowpress-env*)
: A physical model of a bowed string instrument from STK. The *step* parameter is a FLONUM that controls the string length, and the *bowpress-env* controls the bow pressure and also determines the duration of the resulting sound. The *bowpress-env* signal should range from zero to one.

bowed-freq(*step*, *bowpress-env*, *freq-env*)
: A variation of bowed that includes a variable frequency control, *freq-env*, which specifies frequency deviation in Hz. The duration of the resulting sound is the minimum duration of *bowpress-env* and *freq-env*. These parameters may be of type FLONUM or SOUND. FLONUMs are coerced into SOUNDs with a nominal duration arbitrarily set to 30s.

mandolin(*step*, *dur*, &optional *detune*)
: A physical model of a plucked double-string instrument from STK. The *step* parameter is a FLONUM wich specifies the desired pitch, *dur* means the duration of the resulting sound and detune is a FLONUM that controls the relative detune of the two strings. A value of 1.0 means unison. The default value is 4.0. Note: *body-size* (see snd-mandolin does not seem to work correctly, so a default value is always used by mandolin.

wg-uniform-bar(*step*, *bowpress-env*)

wg-tuned-bar(*step*, *bowpress-env*)

wg-glass-harm(*step*, *bowpress-env*)

wg-tibetan-bowl(*step*, *bowpress-env*)
: These sounds are presets for a Banded Wave Guide Percussion instrument implemented in STK. The parameter *step* is a FLONUM that controls the resultant pitch, and *bowpress-env* is a SOUND ranging from zero to one that controls a parameter of the model. In addition, *bowpress-env* determines the duration of the resulting sound. (Note: The *bowpress-env* does not seems influence the timbral quality of the resulting sound).

modalbar(*preset*, *step*, *dur*)
: A physical model of a struck bar instrument implemented in STK. The parameter *preset* is one of the symbols MARIMBA, VIBRAPHONE, AGOGO, WOOD1, RESO, WOOD2, BEATS, TWO-FIXED, or CLUMP. The symbol must be quoted, e.g. for SAL syntax use quote(marimba), and for Lisp syntax use 'marimba. The parameter *step* is a FLONUM that sets the pitch (in steps), and *dur* is the duration in seconds.

sitar(*step*, *dur*)
: A sitar physical model implemented in STK. The parameter *step* is a FLONUM that sets the pitch, and *dur* is the duration.

7.2.2.6. More Behaviors

clip(*sound*, *peak*)
: Hard limit *sound* to the given *peak*, a positive number. The samples of *sound* are constrained between an upper value of *peak* and a lower value of –()*peak*. If *sound* is a number, clip will return *sound* limited by *peak*. If *sound* is a multichannel sound, clip returns a multichannel

sound where each channel is clipped. The result has the type, sample rate, starting time, etc. of *sound*.

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

`s-sqrt(`*sound*`)`
A generalized square root function. If *sound* is a `SOUND`, compute the square root of each sample. If *sound* is a number, just compute the square root. If *sound* is a multichannel sound, return a multichannel sound with `s-sqrt` applied to each element. The result has the type, sample rate, starting time, etc. of *sound*. In taking square roots, if an input sample is less than zero, the corresponding output sample is zero. This is done because the square root of a negative number is undefined.

`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.

`s-max(`*sound1*`, `*sound2*`)`
Compute the maximum of two functions, *sound1* and *sound2*. This function also accepts numbers and multichannel sounds and returns the corresponding data type. The start time of the result is the maximum of the start times of *sound1* and *sound2*. The logical stop time and physical stop time of the result is the minimum of the logical stop and physical stop times respectively of *sound1* and *sound2*. Note, therefore, that the result value is zero except within the bounds of *both* input sounds.

`s-min(`*sound1*`, `*sound2*`)`
Compute the minimum of two functions, *sound1* and *sound2*. This function also accepts numbers and multichannel sounds and returns the corresponding data type. The start time of the result is the maximum of the start times of *sound1* and *sound2*. The logical stop time and physical stop time of the result is the minimum of the logical stop and physical stop times respectively of *sound1* and *sound2*. Note, therefore, that the result value is zero except within the bounds of *both* input sounds.

`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*`.

`quantize(`*sound*`, `*steps*`)`
Quantizes *sound* as follows: *sound* is multiplied by *steps* and rounded to the nearest integer. The result is then divided by *steps*. For example, if *steps* is 127, then a signal that ranges from -1 to +1 will be quantized to 255 levels (127 less than zero, 127 greater than zero, and zero itself). This would match the quantization Nyquist performs when writing a signal to an 8-bit audio file. The *sound* may be multi-channel.

`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*`.

`rms(`*sound*`[ ,`*rate*`, `*window-size*`])`
Computes the RMS of *sound* using a square window of size *window-size*. The result has a sample rate of *rate*. The default value of *rate* is 100 Hz, and the default window size is 1/rate seconds (converted to samples). The *rate* is a `FLONUM` and *window-size* is a `FIXNUM`.

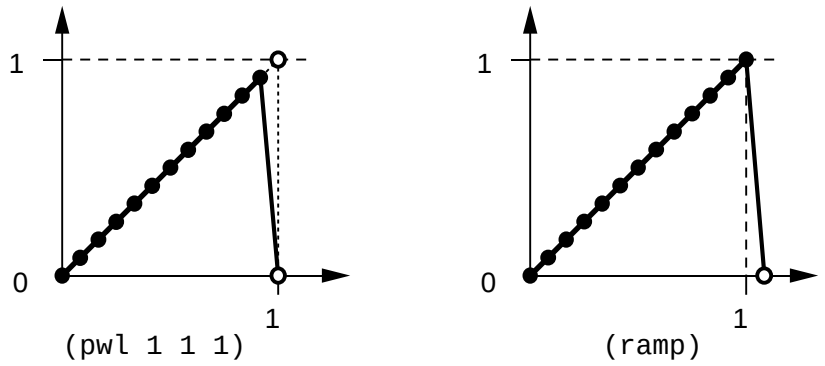

Figure 6: Ramps generated by `pwl` and `ramp` functions. The `pwl` version ramps toward the breakpoint (1, 1), but in order to ramp back to zero at breakpoint (1, 0), the function never reaches an amplitude of 1. If used at the beginning of a `seq` construct, the next sound will begin at time 1. The `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.

`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*`])`
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*`.

`yin(`*sound*`, `*minstep*`, `*maxstep*`, `*stepsize*`)`
Fundamental frequency estimation (pitch detection. Use the YIN algorithm to estimate the fundamental frequency of *sound*, which must be a `SOUND`. The *minstep*, a `FLONUM`, is the minimum frequency considered (in steps), *maxstep*, a `FLONUM`, is the maximum frequency considered (in steps), and *stepsize*, a `FIXNUM`, is the desired hop size. The result is a ‘‘stereo’’

signal, i.e. an array of two SOUNDs, both at the same sample rate, which is approximately the sample rate of *sound* divided by *stepsize*. The first SOUND consists of frequency estimates. The second sound consists of values that measure the confidence or reliability of the frequency estimate. A small value (less than 0.1) indicates fairly high confidence. A larger value indicates lower confidence. This number can also be thought of as a ratio of non-periodic power to periodic power. When the number is low, it means the signal is highly periodic at that point in time, so the period estimate will be reliable. Hint #1: See Alain de Cheveigne and Hideki Kawahara's article "YIN, a Fundamental Frequency Estimator for Speech and Music" in the Journal of the Acoustic Society of America, April 2002 for details on the yin algorithm. Hint #2: Typically, the *stepsize* should be at least the expected number of samples in one period so that the fundamental frequency estimates are calculated at a rate far below the sample rate of the signal. Frequency does not change rapidly and the yin algorithm is fairly slow. To optimize speed, you may want to use less than 44.1 kHz sample rates for input sounds. Yin uses interpolation to achieve potentially fractional-sample-accurate estimates, so higher sample rates do not necessarily help the algorithm and definitely slow it down. The computation time is $O(n^2)$ per estimate, where n is the number of samples in the longest period considered. Therefore, each increase of *minstep* by 12 (an octave) gives you a factor of 4 speedup, and each decrease of the sample rate of *sound* by a factor of two gives you another factor of 4 speedup. Finally, the number of estimates is inversely proportional to *stepsize*. Hint #3: Use `snd-srate` (see Section 7.1.3) to get the exact sample rate of the result, which will be the sample rate of *sound* divided by *stepsize*. E.g. `(snd-srate (aref yin-output 0))`, where `yin-output` is a result returned by `yin`, will be the sample rate of the estimates.

7.3. 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. Or in SAL, we would say `abs-env(my-beh()) @ 10` is equivalent to `abs-env(my-beh())`.

`at(`*time*`, `*beh*`)`
: Evaluate *beh* with `*warp*` shifted by *time*. In SAL, you can use the infix operator `@` as in *beh* `@` *time*. To discover how the environment is shifting time, use `local-to-global(`*time*`)`. Most commonly, you call `local-to-global(0)` to find when a sound created in the current environment will start, expressed in absolute (global) terms. This can be regarded as the ''current time.''

`at-abs(`*time*`, `*beh*`)`
: Evaluate *beh* with `*warp*` shifted so that local time 0 maps to *time*. In SAL, you can use the infix operator `@@` as in *beh* `@@` *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 4 for more information. Use `get-duration(`*dur*`)` to get the nominal actual duration of a behavior that locally has a duration of *dur*. Here, ‘‘nominal’’ means what would be expected if the behavior obeys the shift, stretch, and warp components of the environment. (Any behavior is free to deviate from the nominal timing. For example, a percussion sound might have a fixed duration independent of the stretch factor.) Also, ‘‘actual’’ means global or absolute time, and ‘‘locally’’ means within the environment where `get-duration` is called. `get-duration` works by mapping the current time (local time 0) using `local-to-global` to obtain an actual start time, and mapping *dur* to obtain an actual end time. The difference is returned.

`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 4 for more information.

`sustain(`*factor*`, `*beh*`)`
Evaluates *beh* with `*sustain*` scaled by *factor*. The effect is to ‘‘stretch’’ the result of *beh* (under the current environment) by *factor*; however, the logical stop times are not stretched. Therefore, the overall duration of a sequence is not changed, and sounds will tend to overlap if `*sustain*` is greater than one (legato) and be separated by silence if `*sustain*` is less than one.

`sustain-abs(`*factor*`, `*beh*`)`
Evaluates *beh* with `*sustain*` set to *factor*. (See `sustain`, above.)

`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*`.

7.4. 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 placed sequentially in time as if by `seq`. The symbol *var* is a *read-only* local variable to *beh*. Assignments are not restricted or detected, but may cause a run-time error or crash. In LISP, the syntax is `(seqrep (`*var limit*`) `*beh*`)`.

`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. An important limitation of `sim` is that it cannot handle hundreds of behaviors due to a stack size limitation in XLISP. To compute hundreds of sounds (e.g. notes) at specified times, see `timed-seq`, below. 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`. In LISP, the syntax is `(seqrep (`*var limit*`) `*beh*`)`.

`trigger(`*s*`, `*beh*`)`
Returns a sound which is the sum of instances of the behavior *beh*. One instance is created each

time SOUND *s* makes a transition from less than or equal to zero to greater than zero. (If the first sample of *s* is greater than zero, an instance is created immediately.) The sample rate of *s* and all behaviors must be the same, and the behaviors must be (monophonic) SOUNDs. This function is particularly designed to allow behaviors to be invoked in real time by making *s* a function of a Nyquist slider, which can be controlled by a graphical interface or by OSC messages. See `snd-slider` in Section 7.6.1.

`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*.

`mult(`*a*`[ ,`*b*`, `*c*`, ...])`
Returns the product of *a*, *b*, *c*, ..., allowing mixed multiplication of sounds, multichannel sounds and numbers.

`diff(`*a, b*`)`
Returns the difference between *a* and *b*. This function is defined as `(sum a (prod -1 b))`.

`timed-seq(`*score*`)`
Computes sounds from a note list or ‘‘score.’’ The *score* is of the form: `‘((`*time1 stretch1 beh1*`) (`*time2 stretch2 beh2*`) ...)`, where *timeN* is the starting time, *stretchN* is the stretch factor, and *behN* is the behavior. Note that *score* is normally a *quoted list*! The times must be in increasing order, and each *behN* is evaluated using lisp’s `eval`, so the *behN* behaviors cannot refer to local parameters or local variables. The advantage of this form over `seq` is that the behaviors are evaluated one-at-a-time which can take much less stack space and overall memory. One special ‘‘behavior’’ expression is interpreted directly by `timed-seq`: `(SCORE-BEGIN-END)` is ignored, not evaluated as a function. Normally, this special behavior is placed at time 0 and has two parameters: the score start time and the score end time. These are used in Xmusic functions. If the behavior has a `:pitch` keyword parameter which is a list, the list represents a chord, and the expression is replaced by a set of behaviors, one for each note in the chord. It follows that if `:pitch` is `nil`, the behavior represents a rest and is ignored.

7.5. Sound File Input and Output

`play` *sound*
Play the sound through the DAC. Note that `play` is a command in SAL. In XLISP, it is a function, so the syntax is `(play` *sound*`)`, and in SAL, you can call the XLISP function as `#play(`*sound*`)`. The `play` command or function writes a file and plays it. The details of this are system-dependent, but `play` is defined in the file `system.lsp`. The variable `*default-sf-dir*` names a directory into which to save a sound file. Be careful not to call `play` or `sound-play` within a function and then invoke that function from another `play` command.

By default, Nyquist will try to normalize sounds using the method named by `*autonorm-type*`, which is `’lookahead` by default. The *lookahead* method precomputes and buffers `*autonorm-max-samples*` samples, finds the peak value, and normalizes accordingly. The `’previous` method bases the normalization of the current sound on the peak value of the (entire) previous sound. This might be good if you are working with long sounds that start rather softly. See Section 5.3 for more details.

If you want precise control over output levels, you should turn this feature off by typing:

```
autonorm-off()
```

Reenable the automatic normalization feature by typing:

```
autonorm-on()
```

Play normally produces real-time output. The default is to send audio data to the DAC as it is computed in addition to saving samples in a file. If computation is slower than real-time, output will be choppy, but since the samples end up in a file, you can type `(r)` to replay the stored sound. Real-time playback can be disabled by:

```
sound-off()
```

and reenabled by:

```
sound-on()
```

Disabling real-time playback has no effect on `(play-file)` or `(r)`.

While sounds are playing, typing control-A to Nyquist will push the estimated elapsed audio time onto the head of the list stored in `*audio-markers*`. Because samples are computed in blocks and because there is latency between sample computation and sample playback, the elapsed time may not be too accurate, and the computed elapsed time may not advance after all samples have been computed but the sound is still playing.

`play-file(`*filename*`)`
Play the contents of a sound file named by *filename*. The `s-read` function is used to read the file, and unless *filename* specifies an absolute path or starts with ‘‘.’’, it will be read from `*default-sf-dir*`.

`autonorm-on()`
Enable automatic adjustment of a scale factor applied to sounds computed using the `play` command.

`autonorm-off()`
Disable automatic adjustment of a scale factor applied to sounds computed using the `play` command.

`sound-on()`
Enable real-time audio output when sound is computed by the the `play` command.

`sound-off()`
Disable real-time audio output when sound is computed by the the `play` command.

`s-save(`*expression*`, `*maxlen*`, `*filename*`[ ,format: `*format*`] [, mode: `*mode*`] [, bits: `*bits*`] [, swap: `*flag*`] [, play: `*play*`])`
Evaluates the *expression*, which should result in a sound or an array of sounds, and writes the result to the given *filename*. A `FLONUM` is returned giving the maximum absolute value of all samples written. (This is useful for normalizing sounds and detecting sample overflow.) If *play* is not `NIL`, the sound will be output through the computer’s audio output system. (*:play* is not implemented on all systems; if it is implemented, and *filename* is `NIL`, then this will play the file without also writing a file.) The latency (length of audio buffering) used to play the sound is 0.3s by default, but see `snd-set-latency`. 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, and bytes are swapped if *swap* is not NIL. Defaults for these are `*default-sf-format*`, `*default-sf-mode*`, and `*default-sf-bits*`. The default for *swap* is NIL. The *bits* parameter may be 8, 16, or 32. The values for the *format* and *mode* options are described below:

Format

`snd-head-none`	The format is unknown and should be determined by reading the file.
`snd-head-raw`	A raw format file has 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.
`snd-head-*`	See sndfnint.lsp for more formats.

Mode

`snd-mode-adpcm`	ADPCM mode (not supported).
`snd-mode-pcm`	signed binary PCM mode.
`snd-mode-ulaw`	8-bit U-Law mode.
`snd-mode-alaw`	8-bit A-Law mode (not supported).
`snd-mode-float`	32-bit floating point mode.
`snd-mode-upcm`	unsigned binary PCM mode.
`snd-mode-*`	See sndfnint.lsp for more modes.

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

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

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

Reads a sound from *filename*. The global `*default-sf-dir*` applies. 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*`.

- `:swap` — (T or NIL) swap byte order of each sample. Default is NIL.

If there is an error, for example if `:time-offset` is greater than the length of the file, then `NIL` is returned rather than a sound. Information about the sound is also returned by `s-read` through `*rslt*`[2]. The list assigned to `*rslt*` is of the form: (*format channels mode bits samplerate duration flags byte-offset*), which are defined as follows:

- *format* — the header format. See `s-save` for details.
- *channels* — the number of channels.
- *mode* — the sample representation, e.g. PCM or float. See `s-save` for details.
- *bits* — the number of bits per sample.
- *samplerate* — the sample rate, expressed as a `FLONUM`.
- *duration* — the duration of the sound, in seconds.
- *flags* — The values for *format*, *channels*, *mode*, *bits*, *samplerate*, and *duration* are initially just the values passed in as parameters or default values to `s-read`. If a value is actually read from the sound file header, a flag is set. The flags are: `snd-head-format`, `snd-head-channels`, `snd-head-mode`, `snd-head-bits`, `snd-head-srate`, and `snd-head-dur`. For example,

```
(let ((flags (caddr (cddddr  *rslt*))))
  (not (zerop (logand flags snd-head-srate))))
```

  tells whether the sample rate was specified in the file. See also `sf-info` below.

- *byte-offset* — the byte offset into the file of the first sample to be read (this is used by the `s-overwrite` and `s-add-to` functions).

`s-add-to(`*expression*`, `*maxlen*`, `*filename*`[ ,`*offset*`])`

Evaluates the *expression*, which should result in a sound or an array of sounds, and adds the result to the given *filename*. The global `*default-sf-dir*` applies. A `FLONUM` is returned, giving the maximum absolute value of all samples written. The sample rate(s) of *expression* must match those of the file. 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. If *offset* is specified, the new sound is added to the file beginning at an *offset* from the beginning (in seconds). The file is extended if necessary to accommodate the new addition, but if *offset* falls outside of the original file, the file is not modified. (If necessary, use `s-add-to` to extend the file with zeros.) The file must be a recognized sound file with a header (not a raw sound file).

`s-overwrite(`*expression*`, `*maxlen*`, `*filename*`[ ,`*offset*`])`

Evaluates the *expression*, which should result in a sound or an array of sounds, and replaces samples in the given *filename*. The global `*default-sf-dir*` applies. A `FLONUM` is returned, giving the maximum absolute value of all samples written. The sample rate(s) of *expression* must match those of the file. 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. If *offset* is specified, the new sound is written to the file beginning at an *offset* from the beginning (in seconds). The file is extended if necessary to accommodate the new insert, but if *offset* falls outside of the original file, the file is not modified. (If necessary, use `s-add-to` to extend the file with zeros.) The file must be a recognized sound file with a header (not a raw sound file).

`sf-info(`*filename*`)`

Prints information about a sound file. The parameter *filename* is a string. The file is assumed to

---

[2]Since XLISP does not support multiple value returns, multiple value returns are simulated by having the function assign additional return values in a list to the global variable `*rslt*`. Since this is a global, it should be inspected or copied immediately after the function return to insure that return values are not overwritten by another function.

be in *default-sf-dir* (see `soundfilename` below) unless the filename begins with ‘‘.’’ or ‘‘/’’. The source for this function is in the `runtime` and provides an example of how to determine sound file parameters.

`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*`[, `*dur*`, `*n*`])`
Plots sound in a window. This function was designed to run a `plot` program on a Unix workstation, but now is primarily used with `jNyqIDE`, which has self-contained plotting. Normally, time/value pairs in ascii are written to `points.dat` and system-dependent code (or the `jNyqIDE` program) takes it from there. If the *sound* is longer than the optional *dur* (default is 2 seconds), only the first *dur* seconds are plotted. If there are more than *n* samples to be plotted, the signal is interpolated to have *n* samples before plotting. The data file used is:

`*default-plot-file*` The file containing the data points, defaults to "points.dat".

`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`.

7.6. Low-level Functions

Nyquist includes many low-level functions that are used to implement the functions and behaviors described in previous sections. For completeness, these functions 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, you can skip this section.

7.6.1. Creating Sounds

The basic operations that create sounds are described here.

`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 7.2.2.2) instead.

`snd-read(`*filename*`, `*offset*`, `*t0*`, `*format*`, `*channels*`, `*mode*`, `*bits*`, `*swap*`, `*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 *swap* flag is 0 or 1, where 1 means to swap sample bytes. 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 7.5) instead of `snd-read`. Also, see Section 7.5 for information on the *mode* and *format* parameters.

`snd-save`(*expression*, *maxlen*, *filename*, *format*, *mode*, *bits*, *swap*, *play*)
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, and swapping bytes if *swap* is 1 (otherwise it should be 0). If *play* is not null, the audio is played in real time (to the extent possible) as it is computed. The peak value of the sound is returned. In addition, the symbol `*RSLT*` is bound to a list containing the sample rate, number of channels, and duration (in that order) of the saved sound. ***Note:*** you probably want to call `s-save` (see Section 7.5) instead. The *format* and *mode* parameters are described in Section 7.5.

`snd-overwrite`(*expression*, *maxlen*, *filename*, *offset*, *format*, *mode*, *bits*, *swap*)
Evaluates the *expression*, which should result in a sound or an array of sounds, and replaces samples in the given *filename*, writing the first frame at a time of *offset* seconds. The *offset* must be less than or equal to the duration of the existing file. The duration of the written samples may be greater than that of the file, in which case the file is extended as necessary. The sample rate(s) of *expression* and the number of channels must match those of the file. If *format* is `SND-HEAD-RAW`, then the file format is given by *mode* (see `snd-save`, *bits* (per channel), *swap* (1 means to swap bytes and 0 means write them in the native byte order), and the number of channels and sample rate of the sound returned by evaluating *expression*. If the file is a known audio file format, *format* should be `SND-HEAD-NONE`, and the other parameters are ignored. Up to a maximum of *maxlen* samples will be written per channel. The peak value of the sound is returned. In addition, the symbol `*RSLT*` is bound to a list containing the duration of the written sound (which may not be the duration of the sound file). Use `s-add-to` (in Section 7.5 or `s-overwrite` (in Section 7.5 instead of this function.

`snd-coterm`(*s1*, *s2*)
Returns a copy of *s1*, except the start time is the maximum of the start times of *s1* and *s2*, and the termination time is the minimum of *s1* and *s2*. (After the termination time, the sound is zero as if *s1* is gated by *s2*.) Some rationale follows: In order to implement `s-add-to`, we need to read from the target sound file, add the sounds to a new sound, and overwrite the result back into the file. We only want to write as many samples into the file as there are samples in the new sound. However, if we are adding in samples read from the file, the result of a `snd-add` in Nyquist will have the maximum duration of either sound. Therefore, we may read to the end of the file. What we need is a way to truncate the read, but we cannot easily do that, because we do not know in advance how long the new sound will be. The solution is to use `snd-coterm`, which will allow us to truncate the sound that is read from the file (*s1*) according to the duration of the new sound (*s2*). When this truncated sound is added to the new sound, the result will have only the duration of the new sound, and this can be used to overwrite the file. This function is used in the implementation of `s-add-to`, which is defined in `runtime/fileio.lsp`.

`(snd-from-array ...)`
See page 50.

`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 7.2.2.6).

`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 7.2.2.2) instead.

`get-slider-value(`*index*`)`

Return the current value of the slider named by *index* (an integer index into the array of sliders). Note that this ‘‘slider’’ is just a floating point value in an array. Sliders can be changed by OSC messages (see `osc-enable`) and by sending character sequences to Nyquist’s standard input. (Normally, these character sequences would not be typed but generated by the jNyqIDE interactive development environment, which runs Nyquist as a sub-process, and which present the user with graphical sliders.)

`snd-slider(`*index, t0, srate, duration*`)`

Create a sound controlled by the slider named by *index* (an integer index into the array of sliders; see `get-slider-value` for more information). The function returns a sound. Since Nyquist sounds are computed in blocks of samples, and each block is computed at once, each block will contain copies of the current slider value. To obtain reasonable responsiveness, slider sounds should have high (audio) sample rates so that the block rate will be reasonably high. Also, consider lowering the audio latency using `snd-set-latency`. To ‘‘trigger’’ a Nyquist behavior using slider input, see the `trigger` function in Section 7.4.

7.6.2. Signal Operations

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

`snd-abs(`*sound*`)`

Computes a new sound where each sample is the absolute value of the corresponding sample in *sound*. You should probably use `s-abs` instead. (See Section 7.2.2.6.)

`snd-sqrt(`*sound*`)`

Computes a new sound where each sample is the square root of the corresponding sample in *sound*. If a sample is negative, it is taken to be zero to avoid raising a floating point error. You should probably use `s-sqrt` instead. (See Section 7.2.2.6.)

`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 7.4).

`snd-offset(`*sound, offset*`)`

Add an offset to a sound. The resulting start time, logical stop time, stop time, and sample rate are those of *sound*. Use `sum` instead (see Section 7.4).

`snd-avg(`*sound, blocksize, stepsize, operation*`)`

Computes the averages or peak values of blocks of samples. Each output sample is an average or peak of *blocksize* (a fixnum) adjacent samples from the input *sound*. After each average or peak is taken, the input is advanced by *stepsize*, a fixnum which may be greater or less than *blocksize*. The output sample rate is the *sound* (input) sample rate divided by *stepsize*. This function is useful for computing low-sample-rate rms or peak amplitude signals for input to `snd-gate` or `snd-follow`. To select the operation, *operation* should be one of `OP-AVERAGE` or `OP-PEAK`. (These are global lisp variables; the actual *operation* parameter is an integer.) For RMS computation, see `rms` in Section 7.2.2.6.

`snd-clip(`*sound, peak*`)`

Hard limit *sound* to the given *peak*, a positive number. The samples of *sound* are constrained between an upper value of *peak* and a lower value of –()*peak*. Use `clip` instead (see Section 7.2.2.6).

`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`, `shape`, and `snd-resample`.

For an extended example that uses `snd-compose` for variable pitch shifting, see `demos/pitch_change.htm`.

`snd-tapv(`*sound*`,` *offset*`,` *vardelay*`,` *maxdelay*`)`
A variable delay: *sound* is delayed by the sum of *offset* (a `FIXNUM` or `FLONUM`) and *vardelay* (a `SOUND`). The specified delay is adjusted to lie in the range of zero to *maxdelay* seconds to yield the actual delay, and the delay is implemented using linear interpolation. This function was designed specifically for use in a chorus effect: the *offset* is set to half of *maxdelay*, and the *vardelay* input is a slow sinusoid. The maximum delay is limited to *maxdelay*, which determines the length of a fixed-sized buffer. The function `tapv` is equivalent and preferred (see Section 7.2.2.3).

`snd-tapf(`*sound*`,` *offset*`,` *vardelay*`,` *maxdelay*`)`
A variable delay like `snd-tapv` except there is no linear interpolation. By eliminating interpolation, the output is an exact copy of the input with no filtering or distortion. On the other hand, delays jump by samples causing samples to double or skip even when the delay is changed smoothly.

`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 7.2.2).

`snd-exp(`*sound*`)`
Compute the exponential of each sample of *sound*. Use `s-exp` instead (see Section 7.2.2.6).

`snd-follow(`*sound*`,` *floor*`,` *risetime*`,` *falltime*`,` *lookahead*`)`
An envelope follower. The basic goal of this function is to generate a smooth signal that rides on the peaks of the input signal. The usual objective is to produce an amplitude envelope given a low-sample rate (control rate) signal representing local RMS measurements. The first argument is the input signal. The *floor* is the minimum output value. The *risetime* is the time (in seconds) it takes for the output to rise (exponentially) from *floor* to unity (1.0) and the *falltime* is the time it takes for the output to fall (exponentially) from unity to *floor*. The algorithm looks ahead for peaks and will begin to increase the output signal according to *risetime* in anticipation of a peak. The amount of anticipation (in sampless) is given by *lookahead*. The algorithm is as follows: the output value is allowed to increase according to *risetime* or decrease according to *falltime*. If the next input sample is in this range, that sample is simply output as the next output sample. If the next input sample is too large, the algorithm goes back in time as far as necessary to compute an envelope that rises according to *risetime* to meet the new value. The algorithm will only work backward as far as *lookahead*. If that is not far enough, then there is a final forward pass computing a rising signal from the earliest output sample. In this case, the output signal will be at least momentarily less than the input signal and will continue to rise exponentially until it intersects the input signal. If the input signal falls faster than indicated by *falltime*, the output fall rate will be limited by *falltime*, and the fall in output will stop when the output reaches *floor*. This algorithm can make two passes througth the buffer on sharply rising inputs, so it is not particularly fast. With short buffers and low sample rates this should not matter. See `snd-avg`

above for a function that can help to generate a low-sample-rate input for `snd-follow`. See `snd-chase` in Section 7.6.3 for a related filter.

`snd-gate(`*sound*`, `*lookahead*`, `*risetime*`, `*falltime*`, `*floor*`, `*threshold*`)`
This function generates an exponential rise and decay intended for noise gate implementation. The decay starts when the signal drops below threshold and stays there for longer than lookahead. Decay continues until the value reaches floor, at which point the decay stops and the output value is held constant. Either during the decay or after the floor is reached, if the signal goes above threshold, then the output value will rise to unity (1.0) at the point the signal crosses the threshold. Again, look-ahead is used, so the rise actually starts before the signal crosses the threshold. The rise is a constant-rate exponential and set so that a rise from *floor* to unity occurs in *risetime*. Similarly, the fall is a constant-rate exponential such that a fall from unity to *floor* takes *falltime*. The result is delayed by *lookahead*, so the output is not actually synchronized with the input. To compensate, you should drop the initial *lookahead* of samples. Thus, `snd-gate` is not recommended for direct use. Use `gate` instead (see Section 7.1.4).

`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 7.2.2.6).

`peak(`*expression*`, `*maxlen*`)`
Compute the maximum absolute value of the 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 (`peak` 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 5.3 on page 31 for examples.

`snd-max(`*expression*`, `*maxlen*`)`
Compute the maximum absolute value of the 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 `peak` (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 `peak`.

`snd-maxv(`*sound1*`, `*sound2*`)`
Compute the maximum of *sound1* and *sound2* on a sample-by-sample basis. 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. The physical stop time is the minimum of the physical stop times of the two sounds. *Note that this violates the ‘‘normal’’ interpretation that sounds are zero outside their start and stop times. For example, even if* sound1 *extends beyond* sound2 *and is greater than zero, the result value in this extension will be zero because it will be after the physical stop time, whereas if we simply treated* sound2 *as zero in this region and took the maximum, we would get a non-zero result.* Use `s-max` instead (see Section 7.2.2.6).

`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-oneshot(`*sound*`,` *threshold*`,` *ontime*`)`
Computes a new sound that is zero except where *sound* exceeds threshold. From these points, the result is 1.0 until *sound* remains below *threshold* for *ontime* (in seconds). The result has the same sample rate, start time, logical stop time, and duration as *sound*.

`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 7.2.2). 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 7.2.2.2).

`snd-quantize(`*sound*`,` *steps*`)`
Quantizes a sound. See Section 7.2.2.6 for details.

`snd-recip(`*sound*`)`
Compute the reciprocal of each sample of *sound*. Use `recip` instead (see Section 7.2.2.6).

`snd-resample(`*f*`,` *rate*`)`
Resample sound *f* using high-quality interpolation, yielding a new sound with the specified *rate*. The result is scaled by 0.95 because often, in resampling, interpolated values exceed the original sample values, and this could lead to clipping. The resulting start time, etc. are taken from *f*. Use `resample` instead.

`snd-resamplev(`*f*`,` *rate*`,` *g*`)`
Compose two signals, i.e. compute *f*(*g*(*t*)), where *f* and *g* are sounds. The result has sample rate given by *rate*. At each time *t* (according to the *rate*), *g* is linearly interpolated to yield an increasing sequence of high-precision score-time values. *f* is then interpolated at each value to yield a result sample. 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. The result is scaled by 0.95 because often, in resampling, interpolated values exceed the original sample values, and this could lead to clipping. Note that if *g* has a high sample rate, this may introduce unwanted jitter into sample times. See `sound-warp` for a detailed discussion. See `snd-compose` for a fast, low-quality alternative to this function. Normally, you should use `sound-warp` instead of this function.

`snd-scale(`*scale*`,` *sound*`)`
Scales the amplitude of *sound* by the factor *scale*. Use `scale` instead (see Section 7.2.2).

`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 7.2.2.3).

`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

7.2.2).

`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 7.2.1).

`snd-yin`(*sound*, *minstep*, *maxstep*, *rate*)
Identical to `yin`. See Section 7.2.2.6.

7.6.3. 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-alpass`(*sound*, *delay*, *feedback*)
An all-pass filter. This produces a repeating echo effect without the resonances of `snd-delay`. The *feedback* should be less than one to avoid exponential amplitude blowup. Delay is rounded to the nearest sample. You should use `alpass` instead (see Section 7.2.2.3).

`snd-alpasscv`(*sound*, *delay*, *feedback*)
An all-pass filter with variable *feedback*. This is just like *snd-alpass* except *feedback* is a sound. You should use `alpass` instead (see Section 7.2.2.3).

`snd-alpassvv`(*sound*, *delay*, *feedback*, *maxdelay*)
An all-pass filter with variable *feedback* and *delay*. This is just like *snd-alpass* except *feedback* and *delay* are sounds, and there is an additional `FLONUM` parameter, *maxdelay*, that gives an upper bound on the value of *delay*. ***Note:*** *delay* must remain between zero and *maxdelay*. If not, results are undefined, and Nyquist may crash. You should use `alpass` instead (see Section 7.2.2.3).

`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 7.2.2.3).

`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 7.2.2.3).

`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 7.2.2.3).

`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 7.2.2.3).

`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 7.2.2.3).

`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 7.2.2.3).

`snd-biquad`(*sound*, *b0*, *b1*, *b2*, *a1*, *a2*, *z1init*, *z2init*)
A general second order IIR filter, where *a0* is assumed to be unity. For *a1* and *a2*, the sign convention is opposite to that of Matlab. All parameters except the input *sound* are of type `FLONUM`. You should probably use one of `lowpass2`, `highpass2`, `bandpass2`, `notch2`, `allpass2`, `eq-lowshelf`, `eq-highshelf`, `eq-band`, `lowpass4`, `lowpass6`, `lowpass8`, `highpass4`, `highpass6`, or `highpass8`, which are all based on `snd-biquad` and described in Section 7.2.2.3. For completeness, you will also find `biquad` and `biquad-m` described in that section.

`snd-chase`(*sound*, *risetime*, *falltime*)
A slew rate limiter. The output ‘‘chases’’ the input at rates determined by *risetime* and *falltime*. If the input changes too fast, the output will lag behind the input. This is a form of lowpass filter, but it was created to turn hard-switching square waves into smoother control signals that could be used for linear crossfades. If the input switches from 0 to 1, the output will linearly rise to 1 in *risetime* seconds. If the input switches from 1 to 0, the output will linearly fall to 0 in *falltime* seconds. The generated slope is constant; the transition is linear; this is not an exponential rise or fall. The *risetime* and *falltime* must be scalar constants; complain to the author if this is not adequate. The `snd-chase` function is safe for ordinary use. See `snd-follow` in Section 7.6.2 for a related function.

`snd-congen`(*gate*, *risetime*, *falltime*)
A simple ‘‘contour generator’’ based on analog synthesizers. The *gate* is a sound that normally steps from 0.0 to 1.0 at the start of an envelop and goes from 1.0 back to 0.0 at the beginning of the release. At each sample, the output converges to the input exponentially. If *gate* is greater than the output, e.g. the attack, then the output converges half-way to the output in *risetime*. If the *gate* is less than the output, the half-time is *falltime*. The sample rate, starting time, logical-stop-time, and terminate time are taken from *gate*. You should use `congen` instead (see Section 7.2.2.3.

`snd-convolve`(*sound*, *response*)
Convolves *sound* by *response* using a simple O(N x M) algorithm. The *sound* can be any length, but the *response* is computed and stored in a table. The required compuation time per sample and total space are proportional to the length of *response*. Use `convolve` instead (see Section 7.2.2.3).

`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 7.2.2.3)

`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 7.2.2.3).

`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 7.2.2.3).

`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 7.2.2.3).

`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 7.2.2.3).

`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 7.2.2.3).

`snd-stkchorus`(*sound*, *delay*, *depth*, *freq*, *mix*, *sr*)
A chorus implemented in STK. The parameter *delay* is a `FIXNUM` representing the median desired delay length in samples. A typical value is 6000. The `FLONUM` parameters *depth* and *freq* set the modulation depth (from 0 to 1) and modulation frequency (in Hz), *mix* sets the mixture of input sound and chorused sound, where a value of 0.0 means input sound only (dry) and a value of 1.0 means chorused sound only (wet). The parameter *sr* is the desired sample rate of the resulting sound[3] You should use `pitshift` instead (see Section 7.2.2.4).

`snd-stkpitshift`(*sound*, *shift*, *mix*, *sr*)
A pitch shifter implemented in STK. The *sound* is shifted in pitch by *shift*, a `FLONUM` representing the shift factor. A value of 1.0 means no shift. The parameter *mix* sets the mixture of input and shifted sounds. A value of 0.0 means input only (dry) and a value of 1.0 means shifted sound only (wet). The *sr* is the desired sampling frequency.[4] You should use `pitshift` instead (see Section 7.2.2.4).

`snd-stkrev`(*rev-type*, *sound*, *decay*, *mix*, *sr*)
A reverb implemented in STK. The parameter rev-type is a `FIXNUM` ranging from zero to two and selects the type of reverb. Zero selects NRev type, one selects JCRev, and two selects PRCRev. The input *sound* is processed by the reverb with a *decay* time in seconds (a `FLONUM`). The *mix*, a `FLONUM`, sets the mixture of dry input and reverb output. A value of 0.0 means input only (dry) and a value of 1.0 means reverb only (wet). The sample rate is *sr*[5] You should use `nrev`, `jcrev` or `prcrev` instead (see Section 7.2.2.4).

`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 7.2.2.3).

`snd-tonev`(*sound*, *hz*)

---

[3]This is probably a mistake since sample rate is implied by *sound*. This parameter may be removed in a future release.

[4]This is probably a mistake since sample rate is implied by *sound*. This parameter may be removed in a future release.

[5]This is probably a mistake since sample rate is implied by *sound*. This parameter may be removed in a future release.

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 7.2.2.3).

7.6.4. 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 7.2.2.1).

`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 7.2.2.1).

`snd-fmfb(`*t0*`,` *hz*`,` *sr*`,` *index*`,` *dur*`)`
A Feedback FM oscillator. The resulting sound starts at *t0*, has a fundamental frequency of *hz*, a sample rate of *sr*, and a duration of *dur* seconds. The *index* is a `FLONUM` that specifies the amount of feedback. You should use `fmfb` instead (see Section 7.2.2.1).

`snd-fmfbv(`*t0*`, `*hz*`, `*sr*`, `*index*`)`
A Feedback FM oscillator. The resulting sound starts at *t0*, has a fundamental frequency of *hz*, and a sample rate of *sr*. The *index* is a `SOUND` that specifies the amount of feedback and determines the duration. You should use `fmfb` instead (see Section 7.2.2.1).

`snd-buzz(`*n*`, `*sr*`, `*hz*`, `*t0*`, `*fm*`)`
A buzz oscillator, which generates *n* harmonics of equal amplitude. The *fm* specifies frequency deviation (in Hertz) from *hz*. You should use `buzz` instead (see Section 7.2.2.1).

`snd-pluck(`*sr*`, `*hz*`, `*t0*`, `*d*`, `*final-amp*`)`
A Karplus-Strong plucked string oscillator with sample rate *sr*, fundamental frequency *hz*, starting time *t0*, duration *d*, initial amplitude approximately 1.0 (not exact because the string is initialized with random values) and final amplitude approximately *final-amp*. You should use `pluck` instead (see Section 7.2.2.1).

`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 7.2.2.1).

`snd-partial(`*sr*`, `*hz*`, `*t0*`, `*env*`)`
This is a special case of `snd-amosc` that generates a sinusoid starting at phase 0 degrees. The *env* parameter gives the envelope or any other amplitude modulation. You should use `partial` instead (see Section 7.2.2.1).

`snd-sine(`*t0*`, `*hz*`, `*sr*`, `*d*`)`
This is a special case of `snd-osc` that always generates a sinusoid with initial phase of 0 degrees. You should use `sine` instead (see Section 7.2.2.1).

`snd-siosc(`*tables*`, `*sr*`, `*hz*`, `*t0*`, `*fm*`)`
A Spectral Interpolation Oscillator with frequency modulation. The *tables* is a list of sounds and sample counts as follows: (*table0 count1 table1 ... countN tableN*). The initial waveform is given by *table0*, which is interpolated linearly to *table1* over the first *count1* samples. From *count1* to *count2* samples, the waveform is interpolated from *table1* to *table2*, and so on. If more than *countN* samples are generated, *tableN* is used for the remainder of the sound. The duration and logical stop time of the sound is taken from *fm*, which specified frequency modulation (deviation) in Hertz. You should use `siosc` instead (see Section 7.2.2.1).

7.6.5. Physical Model Functions

These functions perform some sort of physically-based modeling synthesis.

`(snd-bandedwg` *freq bowpress-env preset sr*`)`
A Banded Wave Guide Percussion instrument implemented in STK. The parameter *freq* is a `FLONUM` in Hz, *bowpress-env* is a `SOUND` that ranges from zero to one, *preset* is a `FIXNUM`, and *sr* is the desired sample rate in Hz. Currently, there are four presets: uniform-bar (0), tuned-bar (1), glass-harmonica (2), and tibetan-bowl (3). You should use `wg-uniform-bar`, `wg-tuned-bar`, `wg-glass-harm`, or `wg-tibetan-bowl` instead (see Section 7.2.2.5).

`snd-bowed(`*freq*`, `*bowpress-env*`, `*sr*`)`
A bowed string instrument implemented in STK. The freq is a `FLONUM` in Hertz, bowpress-env is a `SOUND` that ranges from z ero to one, and sr is the desired sample rate (a `FLONUM`). You should use bowed instead (see Section 7.2.2.5).

`snd-bowed-freq(`*freq*`, `*bowpress-env*`, `*freq-env*`, `*sr*`)`
A bowed model just like `snd-bowed` but with an additional parameter for continuous frequency control. You should use `bowed-freq` instead (see Section 7.2.2.5).

`snd-clarinet(`*freq*`, `*breath-env*`, `*sr*`)`
A clarinet model implemented in STK. The *freq* is a `FLONUM` in Hertz, *breath-env* is a `SOUND`

that ranges from zero to one, and *sr* is the desired sample rate (a FLONUM). You should use clarinet instead (see Section 7.2.2.5).

snd-clarinet-freq(*freq*, *breath-env*, *freq-env*, *sr*)
A clarinet model just like snd-clarinet but with an additional parameter for continuous frequency control. You should use clarinet-freq instead (see Section 7.2.2.5).

snd-clarinet-all(*freq*, *vibrato-freq*, *vibrato-gain*, *freq-env*, *breath-env*, *reed-stiffness*, *noise*, *sr*)
A clarinet model just like snd-clarinet-freq but with additional parameters for vibrato generation and continuous control of reed stiffness and breath noise. You should use clarinet-all instead (see Section 7.2.2.5).

snd-flute(*freq*, *breath-env*, *sr*)
A flute implemented in STK. The *freq* is a FLONUM in Hertz, *breath-env* is a SOUND that ranges from zero to one, and *sr* is the desired sample rate (a FLONUM). You should use flute instead (see Section 7.2.2.5).

snd-flute-freq(*freq*, *breath-env*, *freq-env*, *sr*)
A flute model just like snd-flute but with an additional parameter for continuous frequency control. You should use flute-freq instead (see Section 7.2.2.5).

snd-flute-all(*freq*, *vibrato-freq*, *vibrato-gain*, *freq-env*, *breath-env*, *jet-delay*, *noise*, *sr*)
A flute model just like snd-flute-freq but with additional parameters for vibrato generation and continuous control of breath noise. You should use flute-all instead (see Section 7.2.2.5).

snd-mandolin(*t0*, *freq*, *dur*, *body-size*, *detune*, *sr*)
A plucked double-string instrument model implemented in STK. The *t0* parameter is the starting time (in seconds), *freq* is a FLONUM in Hz, *body-size* and *detune* are FLONUMs, and sr is the desired sample rate. You should use mandolin instead (see Section 7.2.2.5).

snd-modalbar(*t0*, *freq*, *preset*, *dur*, *sr*)
Struck bar instrument model implemented in STK. The parameter *t0* is the starting time (in seconds), *freq* is a FLONUM in Hz, preset is a FIXNUM ranging from 0 to 8, *dur* is a FLONUM that sets the duration (in seconds) and *sr* is the desired sample rate. You should use modalbar instead (see Section 7.2.2.5).

snd-sax(*freq*, *breath-env*, *sr*)
A sax model implemented in STK. The *freq* is a FLONUM in Hertz, *breath-env* is a SOUND that ranges from zero to one, and *sr* is the desired sample rate (a FLONUM). You should use sax instead (see Section 7.2.2.5).

snd-sax-freq(*freq*, *freq-env*, *breath-env*, *sr*)
A sax model just like snd-sax but with an additional parameter for continuous frequency control. You should use sax-freq instead (see Section 7.2.2.5).

snd-sax-all(*freq*, *vibrato-freq*, *vibrato-gain*, *freq-env*, *breath-env*, *reed-stiffness*, *noise*, *blow-pos*, *reed-table-offset*, *sr*)
A sax model just like snd-sax-freq but with additional parameters for vibrato generation and continuous control of reed stiffness, breath noise, excitation position, and reed table offset. You should use sax-all instead (see Section 7.2.2.5).

snd-sitar(*t0*, *freq*, *dur*, *sr*)
A sitar model implemented in STK. The parameter *t0* is the starting time, *freq* is a FLONUM (in Hz), E *dur* sets the duration and *sr* is the sample rate (in Hz) of the resulting sound. You should use sitar instead (see Section 7.2.2.5).

7.6.6. 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 7.4.

`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 7.4.

`snd-trigger(`*s*`, `*closure*`)`

This is one of the only ways in which a behavior instance can be created by changes in a signal. When *s* (a `SOUND`) makes a transition from less than or equal to zero to greater than zero, the closure, which takes a starting time parameter, is evaluated. The closure must return a `SOUND`. The sum of all these sounds is returned. If there are no sounds, the result will be zero. The stop time of the result is the maximum stop time of *s* and all sounds returned by the closure. The sample rate of the return value is the sample rate of *s*, and the sounds returned by the closure must all have that same sample rate. Do not call this function. See `trigger` in Section 7.4.

An implementation note: There is no way to have `snd-trigger` return a multichannel sound. An alternative implementation would be a built-in function to scan ahead in a sound to find the time of the next zero crossing. This could be combined with some LISP code similar to `seq` to sum up instances of the closure. However, this would force arbitrary look-ahead and therefore would not work with real-time inputs, which was the motivation for `snd-trigger` in the first place.

8. Nyquist Globals

There are many global variables in Nyquist. A convention in Lisp is to place asterisks (*) around global variables, e.g. `*table*`. This is only a convention, and the asterisks are just like any other letter as far as variable names are concerned. Here are some globals users should know about:

`*table*` Default table used by `osc` and other oscillators.

`*A4-Hertz*` Frequency of A4 in Hertz.. Note: you must call `(set-pitch-names)` to recompute pitches after changing `*A4-Hertz*`.

`*autonorm*` The normalization factor to be applied to the next sound when `*autonorm-type*` is `'previous`. See Sections 5.3 and 7.5.

`*autonormflag*` Enables the automatic normalization feature of the `play` command. You should use `(autonorm-on)` and `(autonorm-off)` rather than setting `*autonormflag*` directly. See Sections 5.3 and 7.5.

`*autonorm-max-samples*` Specifies how many samples will be computed searching for a peak value when `*autonorm-type*` is `'lookahead`. See Sections 5.3 and 7.5.

`*autonorm-previous-peak*` The peak of the previous sound generated by `play`. This is used to compute the scale factor for the next sound when `*autonorm-type*` is `'previous`. See Sections 5.3 and 7.5.

`*autonorm-target*` The target peak amplitude for the autonorm feature. The default value is 0.9. See Sections 5.3 and 7.5.

`*autonorm-type*` Determines how the autonorm feature is implemented. Valid values are `'lookahead` (the default) and `'previous`. See Sections 5.3 and 7.5.

`*breakenable*` Controls whether XLISP enters a break loop when an error is encountered. See Section IV.14.

`*control-srate*` Part of the environment, establishes the control sample rate. See Section 3.1 for details.

`*default-sf-bits*` The default bits-per-sample for sound files. Typically 16.

`*default-sf-dir*` The default sound file directory. Unless you give a full path for a file, audio files are assumed to be in this directory. (Applies to many functions that deal with sound files. Check the function description to see if `*default-sf-dir*` applies.)

`*default-sf-format*` The default sound file format. When you write a file, this will be the default format: AIFF for Mac and most Unix systems, NeXT for NeXT systems, and WAV for Win32.

`*default-sf-srate*` The default sample rate for sound files. Typically 44100.0, but often set to 22050.0 for speed in non-critical tasks.

`*default-control-srate*` Default value for `*control-srate*`. This value is restored when you execute `(top)` to pop out of a debugging session. Change it by calling `(set-control-srate` *value*`)`.

`*default-sound-srate*` Default value for `*sound-srate*`. This value is restored when you execute `(top)` to pop out of a debugging session. Change it by calling `(set-sound-srate` *value*`)`.

`*file-separator*` The character that separates directories in a path, e.g. ‘‘`/`’’ for Unix,

	‘‘:’’ for Mac, and ‘‘\’’ for Win32. This is normally set in `system.lsp`.
`*rslt*`	When a function returns more than one value, `*rslt*` is set to a list of the ‘‘extra’’ values. This provides a make-shift version of the `multiple-value-return` facility in Common Lisp.
`*sound-srate*`	Part of the environment, establishes the audio sample rate. See Section 3.1 for details.
`*soundenable*`	Controls whether writes to a sound file will also be played as audio. Set this variable by calling `(sound-on)` or `(sound-off)`.
`*tracenable*`	Controls whether XLISP prints a backtrace when an error is encountered.
XLISP variables	See Section IV.14 for a list of global variables defined by XLISP.
Environment variables	See Section 3.1 for definitions of variables used in the environment for behaviors. In general, you should never set or access these variables directly.
Various constants	See Section 1.7 for definitions of predefined constants for loudness, duration, and pitch.

9. Time/Frequency Transformation

Nyquist provides functions for FFT and inverse FFT operations on streams of audio data. Because sounds can be of any length, but an FFT operates on a fixed amount of data, FFT processing is typically done in short blocks or windows that move through the audio. Thus, a stream of samples is converted in to a sequence of FFT frames representing short-term spectra.

Nyquist does not have a special data type corresponding to a sequence of FFT frames. This would be nice, but it would require creating a large set of operations suitable for processing frame sequences. Another approach, and perhaps the most ‘‘pure’’ would be to convert a single sound into a multichannel sound, with one channel per bin of the FFT.

Instead, Nyquist violates its ‘‘pure’’ functional model and resorts to objects for FFT processing. A sequence of frames is represented by an XLISP object. Whenever you send the selector `:next` to the object, you get back either NIL, indicating the end of the sequence, or you get an array of FFT coefficients.

The Nyquist function `snd-fft` (mnemonic, isn’t it?) returns one of the frame sequence generating objects. You can pass any frame sequence generating object to another function, `snd-ifft`, and turn the sequence back into audio.

With `snd-fft` and `snd-ifft`, you can create all sorts of interesting processes. The main idea is to create intermediate objects that both accept and generate sequences of frames. These objects can operate on the frames to implement the desired spectral-domain processes. Examples of this can be found in the file `fft_tutorial.htm`, which is part of the standard Nyquist release. The documentation for `snd-fft` and `snd-ifft` follows.

`snd-fft(`*sound*`,` *length*`,` *skip*`,` *window*`)`

This function performs an FFT on the first samples in *sound* and returns a Lisp array of `FLONUM`s. The function modifies the *sound*, violating the normal rule that sounds are immutable in Nyquist, so it is advised that you copy the sound using `snd-copy` if there are any other references to *sound*. The length of the FFT is specified by *length*, a `FIXNUM` (integer). After each FFT, the sound is advanced by *skip* samples, also of type `FIXNUM`. Overlapping FFTs, where *skip* is less than *length*, are allowed. If *window* is not `NIL`, it must be a sound. The first *length* samples of *window* are multiplied by *length* samples of *sound* before performing the FFT. When there are no more samples in *sound* to transform, this function returns `NIL`. The coefficients in the returned array, in order, are the DC coefficient, the first real, the first imaginary, the second real, the second imaginary, etc. If the length is even, the last array element corresponds to the real coefficient at the Nyquist frequency.

`snd-ifft(`*time*`,` *srate*`,` *iterator*`,` *skip*`,` *window*`)`

This function performs an IFFT on a sequence of spectral frames obtained from *iterator* and returns a sound. The start time of the sound is given by *time*. Typically, this would be computed by calling `(local-to-global 0)`. The sample rate is given by *srate*. Typically, this would be `*sound-srate*`, but it might also depend upon the sample rate of the sound from which the spectral frames were derived. To obtain each frame, the function sends the message `:next` to the *iterator* object, using XLISP’s primitives for objects and message passing. The object should return an array in the same format as obtained from `snd-fft`, and the object should return `NIL` when the end of the sound is reached. After each frame is inverse transformed into the time domain, it is added to the resulting sound. Each successive frame is added with a sample offset specified by *skip* relative to the previous frame. This must be an integer greater than zero. If *window* is not `NIL`, it must be a sound. This window signal is multiplied by the inverse transformed frame before the frame is added to the output sound. The length of each frame should

be the same. The length is implied by the array returned by *iterator*, so it does not appear as a parameter. This length is also the number of samples used from *window*. Extra samples are ignored, and window is padded with zeros if necessary, so be sure *window* is the right length. The resulting sound is computed on demand as with other Nyquist sounds, so `:next` messages are sent to *iterator* only when new frames are needed. One should be careful not to reuse or modify *iterator* once it is passed to `snd-ifft`.

10. MIDI, Adagio, and Sequences

Nyquist includes facilities to read and write MIDI files as well as an ASCII text-based score representation language, Adagio. XLISP and Nyquist can be used to generate MIDI files using compositional algorithms. (See also Section 13.) A tutorial on using the Adadio representation and MIDI can be found in `demos/midi_tutorial.htm`. The Adagio language is described below. Adagio was originally developed as part of the CMU MIDI Toolkit, which included a program to record and play MIDI using the Adagio representation. Some of the MIDI features of Adagio may not be useful within Nyquist.

Nyquist offers a number of different score representations, and you may find this confusing. In general, MIDI files are a common way to exchange music performance data, especially with sequencers and score notation systems. The `demos/midi_tutorial.htm` examples show how to get the most precise control when generating MIDI data. Adagio is most useful as a text-based score entry language, and it is certainly more compact than Lisp expressions for MIDI-like data. The Xmusic library (Chapter 13) is best for algorithmic generation of music and score manipulation. There are functions to convert between the Adagio, MIDI sequence data, and Xmusic score representations.

Adagio is an easy-to-use, non-procedural notation for scores. In Adagio, text commands are used to specify each note. If you are new to Adagio, you may want to glance at the examples in Section 10.3 starting on page 101 before reading any further.

A note is described in Adagio by a set of attributes, and any attribute not specified is ‘‘inherited’’ from the previous line. Attributes may appear in any order and must be separated by one or more blanks. An attribute may not contain any blanks. The attributes are: time, pitch, loudness, voice number, duration, and articulation.

Adagio has been used to program a variety of hardware and software synthesizers, and the Adagio compiler can be easily adapted to new environments. Although not originally intended for MIDI, Adagio works quite well as a representation for MIDI scores. Adagio has been extended to allow MIDI controller data such as modulation wheels, pitch bend, and volume, MIDI program commands to change timbre, and System Exclusive messages.

A note command in Adagio must be separated from other notes. Usually, notes are distinguished by writing each one on a separate line. Notes can also be separated by using a comma or semicolon as will be described below.

Besides notes, there are several other types of commands:

1. An asterisk (`*`) in column one (or immediately after a comma, semicolon, or space) indicates that the rest of the line is a comment. The line is ignored by Adagio, and is therefore a good way to insert text to be read by people. Here are some examples:

```
* This is a comment.
T150 G4  * This is a comment too!
T150 G4  ;* So is this.
```

2. An empty command (a blank line, for example) is ignored as if it were a comment[6].

[6]To be consistent, a blank line ought to specify zero attributes and generate a note that inherits all of its attributes from the previous one. Adagio is intentionally inconsistent in this respect.

3. An exclamation point (!) in column one (or immediately after a comma or semicolon) indicates a special command. A special command does not generate a note. Special commands follow the ‘‘!’’ with no intervening spaces and extend to the end of the line, for example:

```
!TEMPO 100
```

4. Control change commands are used to control parameters like pitch bend, modulation, and program (timbre). Control change commands can be specified along with notes or by themselves. A command that specifies control changes without specifying a pitch will not produce a note.

Adagio is insensitive to case, thus ‘‘A’’ is equivalent to ‘‘a’’, and you can mix upper and lower case letters freely.

10.1. Specifying Attributes

A note is indicated by a set of attributes. Attributes are indicated by a string of characters with no intervening spaces because spaces separate attributes. The attributes are described below.

The default unit of time is a centisecond (100^{th}’s), but this can be changed to a millisecond (1000^{th}’s) using the `!MSEC` command and reset to centiseconds with `!CSEC` (see Section 10.4.1). In the descriptions below, the term ‘‘time unit’’ will be used to mean whichever convention is currently in effect.

10.1.1. Time

The time attribute specifies when to start the note. A time is specified by a ‘‘T’’ followed by a number representing time units or by a duration (durations are described below). Examples:

```
T150    ** 1.5 sec (or .15 sec)
TQ3     ** 3 quarter note's duration
```

If no time is specified, the default time is the sum of the time and duration attributes of the previous note. (But see Section 10.1.4.) Time is measured relative to the time of the most recent Tempo or Rate command. (See the examples in Section 10.3 for some clarification of this point.)

10.1.2. Pitch

The pitch attribute specifies what frequency to produce. Standard scale pitches are named by name, using `S` for sharp, `F` for flat, and (optionally) `N` for natural. For example, `C` and `CN` represent the same pitch, as do `FS` and `GF` (F sharp and G flat). Note that there are no bar lines, and accidentals to not carry forward to any other notes as in common practice notation.

Octaves are specified by number. `C4` is middle C, and `B3` is a half step lower. `F5` is the top line of the treble clef, etc. (Adagio octave numbering follows the ISO standard, but note that this is not universal. In particular, Yamaha refers to middle C as C3.) Accidentals can go before or after the octave number, so `FS3` and `F3S` have the same meaning.

An alternate notation for pitch is `P`*n*, where *n* is an integer representing the pitch.Middle C (C4) is equivalent to `P60`, `CS4` is `P61`, etc.

If you do not specify an octave, Adagio will choose one for you. This is done by picking the octave

that will make the current pitch as close to the previous pitch as possible. In the case of augmented fourths or diminished fifths, there are two equally good choices. Adagio chooses the lower octave.

10.1.3. Duration

Duration is specified by a letter indicating a number of beats, followed by one or several modifiers. The basic duration codes are:

`W` (whole, 4 beats),
`H` (half, 2 beats),
`Q` (quarter, 1 beat),
`I` (eighth, 1/2 beat),
`S` (sixteenth, 1/4 beat),
`%` (thirtysecond, 1/8 beat), and
`^` (sixtyfourth, 1/16 beat).

Note that `E` is a pitch, so eighth-notes use the duration code `I`. The default tempo is 100 beats per minute (see Section 10.1.10). These codes may be followed by a `T` (triplet), indicating a duration of 2/3 the normal. A dot (`.`) after a duration code extends it by half to 3/2 the normal. An integer after a note multiplies its duration by the indicated value (the result is still just one note). Finally, a slash followed by an integer divides the duration by the integer. Like all attributes, duration attributes may not have embedded spaces. Examples:

`Q`	1 beat (quarter note)
`QT`	2/3 beat (quarter triplet)
`W.`	6 beats(dotted whole note)
`ST6`	1 beat (6 sixteenth triplets)
`H5`	10 beats(5 half notes)
`Q3/7`	3/7 beats

A duration may be noted by `U`*n*, where *n* is an integer indicating 100th's of a second (or 1000th's), see Section 10.4.1. For example, `U25` is twenty-five time units.

Durations may be combined using a plus sign:

```
Q+IT        ** a quarter tied to an eighth triplet
Q/7+W+Q2/7  ** a 7th beat tied to a whole tied to 2/7th beat
Q+U10       ** a quarter plus 10 time units
```

10.1.4. Next Time

The time of the next command (the next command in the Adagio program text) is normally the time of the current note command plus the duration of the current note. This can be overridden by a field consisting of the letter `N` followed by a number indicating time units, or followed by a duration as described above. The next note will then start at the time of the current note plus the duration specified after `N`. If the next note has an explicit time attribute (`T`), then the specified time will override the one based on the previous note. Examples:

```
N0      ** start the next note at the same time as this one
N50     ** start the next note 0.5 seconds after this one
NQT     ** start the next note 2/3 beat after the current one
NU10+Q  ** start after 0.1 seconds plus a quarter
```

A comma has an effect similar to `N0` and is explained in Section 10.4.2. Articulation effects such as *staccato* can be produced using `N`, but it is more convenient to use the articulation attribute described in Section 10.1.6.

10.1.5. Rest

Rests are obtained by including the field R in a note command. The effect of an R field is to omit the note that would otherwise occur as the result of the current note command. In all other respects, the command is processed just like any other line. This means that attributes such as duration, loudness, and pitch can be specified, and anything specified will be inherited by the note in the next command. Normally, a rest will include just R and a duration. The fact that a note command specifies a rest is not inherited. For example:

```
R H      ** a half (two beat) rest
RH       ** illegal, R must be separated from H by space(s)
```

Because some synthesizers (e.g. a DX7) cannot change programs (presets) rapidly, it may be desirable to change programs in a rest so that the synthesizer will be ready to play by the end of the rest. See Section 10.1.9 for an example.

10.1.6. Articulation

Articulation in Adagio refers to the percentage of time a note is on relative to the indicated duration. For example, to play a note *staccato*, you would normally play the note about half of its indicated duration. In Adagio, articulation is indicated by # followed by an integer number indicating a percentage. The articulation attribute does not affect the time of the next command. This example plays two *staccato* quarter notes:

```
C Q #50
D
```

To produce overlapping notes, the articulation may be greater than 100.

Be aware that overlapping notes on the same pitch can be a problem for some synthesizers. The following example illustrates this potential problem:

```
!TEMPO 60
C Q #160   * starts at time 0,   ends at 1.6 sec
D I        * starts at time 1,   ends at 1.8 sec
C Q        * starts at time 1.5, ends at 3.1 sec?
```

At one beat per second (tempo 60), these three notes will start at times 0, 1, and 1.5 seconds, respectively. Since these notes have an articulation of 160, each will be on 160% of its nominal duration, so the first note (C) will remain on until 1.6 seconds. But the third note (another C) will start at time 1.5 seconds. Thus, the second C will be started before the first one ends. Depending on the synthesizer, this may cancel the first C or play a second C in unison. In either case, a note-off message will be sent at time 1.6 seconds. If this cancels the second C, its actual duration will be 0.1 rather than 1.6 seconds as intended. A final note-off will be sent at time 3.1 seconds.

10.1.7. Loudness

Loudness is indicated by an L followed by a dynamic marking from the following: PPP, PP, P, MP, MF, F, FF, FFF. Alternatively, a number from 1 to 127 may be used. The loudness attribute is the MIDI note velocity. (Note that a MIDI velocity of 0 means ‘‘note-off,’’ so the minimum loudness is 1.) The dynamicmarkings are translated into numbers as follows:

Lppp	20	Lmf	58
Lpp	26	Lf	75
Lp	34	Lff	98
Lmp	44	Lfff	127

10.1.8. Voice

The voice attribute tells which of the 16 MIDI channels to use for the note. The voice attribute consists of a V followed by an integer from 1 (the default) to 16.

There is a limit to how many notes can be played at the same time on a given voice (MIDI channel). Since the limit depends upon the synthesizer, Adagio cannot tell you when you exceed the limit. Similarly, Adagio cannot tell whether your synthesizer is set up to respond to a given channel, so there is no guarantee that what you write will actually be heard.

10.1.9. Timbre (MIDI Program)

A MIDI program (synthesizer preset) can be selected using the attribute Z*n*, where *n* is the program number (from 1 to 128). Notice that in MIDI, changing the program on a given channel will affect *all* notes on that channel and possibly others. Adagio treats MIDI program changes as a form of control change.

For many synthesizers, you will not be able to change programs at the start of a note or during a note. Change the program during a rest instead. For example:

```
R I Z23 V4      ** change MIDI channel 4 to program 23 during rest
A4              ** play a note on channel 4
```

Check how your synthesizer interprets program numbers. For example, the cartridge programs on a DX7 can be accessed by adding 32 to the cartridge program number. Cartridge program number 10 is specified by Z42.

As in MIDI, the Adagio timbre is a property of the voice (MIDI channel), so the timbre will not be inherited by notes on a different channel; to change the timbre on multiple voices (channels), you must explicitly notate each change.

10.1.10. Tempo

The length of a beat may be changed using a Tempo command:

```
!TEMPO n
```

where *n* indicates beats per minute. The exclamation mark tells Adagio that this is a special command line rather than a note definition. A special command takes the place of a note specification. No other attributes should be written on a line with a special command. The !TEMPO command is associated with a time, computed as if the !TEMPO command were a note. The time attribute (T) of all succeeding notes is now measured relative to the time of the !TEMPO command. The new tempo starts at the !TEMPO command time and affects all succeeding notes. Durations specified in time units (for example U58, N15) are not affected by the !TEMPO command, and numerical times (for example T851) are computed relative to the time of the last !TEMPO command.

The !TEMPO command is fairly clever about default durations. If the last duration specified before the !TEMPO command is symbolic (using one of ^,%, S, I, Q, H, or W), then the default duration for the node after the !TEMPO command will be modified according to the tempo change. Consider the following tempo change:

```
!TEMPO 60
A4 H
!TEMPO 120
G
```

In this example, the first note will last 2 seconds (2 beats at 60 beats per minute). The second note inherits the duration (H) from the first note, but at 120 beats per minute, the second note will last only 1 second. If the duration had been specified U200 (also a duration of 2 seconds), the second note would also last 2 seconds because the !TEMPO command does not affect times or durations specified numerically in time units. If the duration is the sum of a symbolic and a numeric specification, the

inherited duration after a !TEMPO command is undefined.

10.1.11. Rate

The !RATE command scales all times including those specified in hundredths of seconds. A rate of 100 means no change, 200 means twice as fast, and 50 means half as fast. For example, to make a piece play 10% faster, you can add the following command at the beginning of the score:

```
!RATE 110
```

!RATE and !TEMPO commands combine, so

```
!RATE 200
!TEMPO 70
```

will play 70 beats per minute at double the normal speed, or 140 beats per minute. Like !TEMPO, the time of the !RATE command is added to the time attribute of all following notes up to the next !TEMPO or !RATE command.

Two !RATE commands do not combine, so a !RATE command only affects the rate until the next !RATE command.

Although !TEMPO and !RATE can occur in the middle of a note (using N, T, etc.) they do not affect a note already specified. This property allows multiple tempi to exist simultaneously (see Section 10.4.4).

10.2. Default Attributes

If an attribute is omitted, the previous one is used by default (with the exception of the time attribute). The default values for the first note, which are inherited by succeeding notes until something else is specified, are given below in Adagio notation:

Time	T0
Pitch	C4
Duration	Q
Articulation	#100
Loudness	LFFF
Voice	V1
Tempo	!TEMPO 100
Rate	!RATE 100

Control changes (including timbre or MIDI program, specified by Z) have no default value and are only sent as specified in the score.

Important: the rules for determining when a command will play a note are as follows (and this has changed slightly from previous versions):

1. If a special (!) command or nothing is specified, e.g. a blank line, do *not* play a note.
2. If R (for ‘‘rest’’) is specified, do *not* play a note.
3. Otherwise, if a pitch is specified, *do* play a note.
4. Otherwise, if no control changes (or program changes) are specified (so this is a command with non-pitch attributes and no control changes), *do* play a note.

Another way to say this is ‘‘Special commands and commands with rests (R) do not play notes. Otherwise, play a note if a pitch is specified or if no control is specified.’’

10.3. Examples

The following plays the first two bars of ‘‘Happy Birthday’’. Note that Adagio knows nothing of bar lines, so the fact that the first note occurs on beat 3 or that the meter is three-four is of no consequence:

```
*Example 1 ** Happy Birthday tune (C major)
!TEMPO 120
G4 I. LF
G4 S
A4 Q
G4
C5
B4 H
```

The time attribute for the first note is zero (`0`). The second note will occur a dotted eighth later, etc. Notice that no timbre or rate was specified. Adagio will provide reasonable default values of 1 and 100, respectively.

The following example plays the first four bars of an exercise from Bartok’s Mikrokosmos (Vol. 1, No. 12). An extra quarter note is inserted at the beginning of each voice in order to allow time to change MIDI programs. The right hand part is played on voice (MIDI channel) 1 and the left hand part on voice 2. Notice the specification of the time attribute to indicate that voice 2 starts at time 0. Also, default octaves are used to reduce typing.

```
*Example 2 ** Bartok
*voice 1, right hand
R Q Z10 V1   ** extra rest for program change
A4 H
B Q
C
D H
C
D Q
C
B
A
B
C
D
R

*voice 2, left hand
T0 R Q Z15 V2   ** extra rest for program change
G3 H
F Q
E
D H
E
D Q
E
F
G
F
E
D
R
```

The next example is the same piece expressed in a different manner, illustrating the interaction between

the !TEMPO command and the time attribute. Recall that the time attribute is measured relative to the time of the last !TEMPO command:

```
*Example 3 ** 4 measures in 2 sections
!Tempo 100
*Voice 1, Measures 1 & 2
R Q Z10 V1
A4 H
B Q
C
D H
C

*Voice 2, Measures 1 & 2
T0 R Q Z15 V2
G3 H
F Q
E
D H
E H

!TEMPO 100
*Voice 1, Measures 3 & 4
* note that Z10 is still in effect for V1
V1 D4 Q
C
B
A
B
C
D
R

*Voice 2, Measures 3 & 4
T0 V2 D3 Q
E
F
G
F
E
D
R
```

The piece is written in 4 sections. The first plays a rest followed by two measures, starting at time 0. The next section changes the time back to zero and plays two measures of the left hand part (voice 2). The next command (!TEMPO 100) sets the tempo to 100 (it already is) *and* sets the reference time to be two measures into the piece. Therefore, the next note (D4) will begin measure 3. The D3 that begins the last group of notes has a T0 attribute, so it will also start at measure 3. Notice how the !TEMPO command can serve to divide a piece into sections.

The last example will show yet another way to express the same piece of music using the ‘‘Next’’ attribute. Only the first bar of music is given.

```
*Example 4 ** use of the Next attribute
!Tempo 100
R Q Z10 V1 N0
R Q Z15 V2

A4 H V1 N0
G3   V2

B4 Q V1 N0
F3   V2

C4 Q V1 N0
E3   V2
```

Here, each pair of lines represents two simultaneous notes. The `N0` attribute forces the second line to start at the same time as the first line of each pair. Because of the large intervals, octave numbers (3 and 4) are necessary to override the default octave for these pitches.

10.4. Advanced Features

Beyond the simple notation described above, Adagio supports a number of features. (See also the next chapter.)

10.4.1. Time Units and Resolution

The default time unit is 10ms (ten milliseconds or one centisecond or 100^{th} of a second), but it is possible to change the basic unit to 1ms, or 1000^{th} of a second. The time unit can be specified by:

`!CSEC`	centisecond time units = 100^{th}
`!MSEC`	millisecond time units = 1000^{th}

The time unit remains in effect until the next `!CSEC` or `!MSEC` command.

10.4.2. Multiple Notes Per Line

Notes can be separated by commas or semicolons as well as by starting a new line. A comma is equivalent to typing `N0` and starting a new line. In other words, the next note after a comma will start at the same time as the note before the comma. In general, *use commas to separate the notes of a chord.*

A semicolon is equivalent to starting a new line. In general, *use semicolons to group notes in a melody*. Here is yet another rendition of the Bartok:

```
*Example 5 ** use of semicolons
!Tempo 100
R Q Z10 V1
A4 H; B Q; C; D H; C; D Q; C; B; A; B; C; D; R

T0 R Q Z15 V2
G3 H; F Q; E; D H; E; D Q; E; F; G; F; E; D; R
```

This example is similar to Example 2, except semicolons are used. Note how semicolons make the two lines of music stand out. The next example is similar to Example 4, except commas are used and four bars are notated. The music below is treated as a sequence of 2-note chords, with each chord on a separate line:

```
*Example 6 ** use of commas
!Tempo 100
R Q Z10 V1, R Q Z15 V2
A4 H V1, G3 V2
B4 Q V1, F3 V2
C4   V1, E3 V2
D4 H V1, D3 V2
C4   V1, E3 V2
D4 Q V1, D3 V2
C4   V1, E3 V2
B4   V1, F3 V2
A4   V1, G3 V2
B4   V1, F3 V2
C4   V1, E3 V2
D4   V1, D3 V2
R
```

10.4.3. Control Change Commands

Any control change can be specified using the syntax ‘‘~*n*(*v*)’’, where *n* is the controller number (0 - 127), and *v* is the value. In addition, Adagio has some special syntax for some of the commonly used control changes (note that Pitch bend, Aftertouch, and MIDI Program Change are technically not MIDI control changes but have their own special message format and status bytes):

K Portamento switch

M Modulation wheel

O Aftertouch

X Volume

Y Pitch bend

Z Program Change

The letter listed beside each control function is the Adagio command letter. For example, `M23` is the command for setting the modulation wheel to 23. Except for pitch bend, the portamento switch, and MIDI Program Change, all values range from 0 to 127. Pitch bend is ‘‘off’’ or centered at 128, and has a range from 0 to 255 (MIDI allows for more precision, but Adagio does not). Turn on portamento with `K127` and off with `K0`. Programs are numbered 1 to 128 to correspond to synthesizer displays.

About volume: Midi volume is just a control, and the Midi standard does not say what it means. Typically it does what the volume pedal does; that is, it scales the amplitude in a continuously changeable fashion. In contrast, Midi velocity, which is controlled by the `L` (loudness) attribute, is part of a Midi note-on command and is fixed for the duration of the note. Typically, these two ways of controlling loudness and amplitude operate independently. In some low-cost synthesizers the numbers seem to be added together internally and volume changes are ignored after the note starts.

About pitch bend: Midi pitch bend is a number from 0 to 16383, where 8192 is the center position. To convert to Midi, Adagio simply multiplies your number by 64, giving values from 0 to 16320. Note that `Y128` translates exactly to 8192. The *meaning* of pitch bend depends upon your synthesizer and its setting. Most synthesizers let you specify a ‘‘pitch bend range.’’ A range of one semitone means that

Y255 will produce a bend of approximately one semitone up, and Y0 will bend one semitone down. If the range is 12 semitones, then the same Y255 will bend an octave. Typically, pitch bend is exponential, so each increment in the pitch bend value will bend an equal number of cents in pitch.

Control changes can be part of a note specification or independent. In the following example, a middle C is played with a modulation wheel setting of 50 and a pitch bend of 120. Then, at 10 unit intervals, the pitch bend is decreased by 10. The last line sets the portamento time (controller 5) to 80:

```
*Example 7
C4 LMF M50 Y120 U100 N10
Y110 N10; Y100 N10; Y90 N10; Y80 N10
Y70 N10; Y60 N10; Y50 N10
~5(80)
```

See Section 10.2 on page 100 for rules on whether or not a command will play a note.

10.4.4. Multiple Tempi

Writing a piece with multiple tempi requires no new commands; you just have to be clever in the use of Tempo and Time. The following plays a 7 note diatonic scale on voice 1, and a 12 note chromatic scale on voice 2:

```
*Example 8 ** multiple tempi
!TEMPO 70
V1 C4; D; E; F; G; A; B
T0 R N0

!TEMPO 120
V2 C4; CS; D; DS; E; F; FS; G; GS; A; AS; B

!TEMPO 100
V1 C5, V2 C5
```

The third line plays the 7-note diatonic scale on voice 1. The next line contains the tricky part: notice that the time is set back to zero, there is a rest, and a next (N) attribute is used to specify that the next default time will be at the same time as the current one. This is tricky because a !TEMPO command cannot have a time (T0) attribute, and a T0 by itself would create a note with a duration. T0 R N0 says: ‘‘go to time 0, do not play a note, and do not advance the time before the next command’’. Thus, the time of the !TEMPO 120 command is zero. After the 12 note scale, the tempo is changed to 100 and a final note is played on each voice. A little arithmetic will show that 7 notes at tempo 70 and 12 notes at tempo 120 each take 6 seconds, so the final notes (C5) of each scale will happen at the same time.

10.4.5. MIDI Synchronization

The Adagio program (but not Nyquist) can synchronize with external devices using MIDI real time messages. Thus, Adagio has a !CLOCK command. This command is currently of no use to Nyquist users but is documented here for completeness (it’s part of the language syntax even if it does not do anything).

Since Adagio supports multiple tempi, and Midi clock is based on beats, it is necessary to be explicit in the score about where the clock should start and what is the duration of a quarter note. The !CLOCK command in Adagio turns on a 24 pulse-per-quarter (PPQ) clock at the current tempo and time:

```
!TEMPO 100
!CLOCK
```

A `!CLOCK` command must also be inserted for each tempo change that is to be reflected in the Midi clock. Typically, each !TEMPO command will be followed by a !CLOCK command.
Clock commands and thus tempo changes can take place at arbitrary times. It is assumed that tempo changes on an exact 24th of a beat subdivision (for example, exactly on a beat). If not, the tempo change will take place on the nearest exact 24th of a beat subdivision. This may be earlier or later than the requested time.

10.4.6. System Exclusive Messages

Adagio has a definition facility that makes it possible to send system exclusive parameters. Often, there are parameters on Midi synthesizers that can only be controlled by system exclusive messages. Examples include the FM ratio and LFO rate on a DX7 synthesizer. The following example defines a macro for the DX7 LFO rate and then shows how the macro is used to set the LFO rate for a B-flat whole note in the score. The macro definition is given in hexadecimal, except `v` is replaced by the channel (voice) and `%1` is replaced by the first parameter. A macro is invoked by writing ‘‘~’’ followed by the macro name and a list of parameters:

```
!DEF LFO F0 43 0v 01 09 %1 F7
Bf5 W ~LFO(25)
```

In general, the `!DEF` command can define any single MIDI message including a system exclusive message. The message must be complete (including the status byte), and each `!DEF` must correspond to just one message. The symbol following `!DEF` can be any name consisting of alphanumeric characters. Following the name is a hexadecimal string (with optional spaces), all on one line. Embedded in the string may be the following special characters:

`v` — Insert the 4-bit voice (MIDI channel) number. If `v` occurs in the place of a high-order hexadecimal digit, replace `v` with `0v` so that the channel number is always placed in the low-order 4 bits of a data byte. In other words, `v` is padded if necessary to fall into the low-order bits.

`%`*n* — Insert a data byte with the low-order 7 bits of parameter number *n*. Parameters are numbered 1 through 9. If the parameter value is greater than 127, the high-order bits are discarded.

`^`*n* — Insert a data byte with bits 7 through 13 of parameter number *n*. In other words, shift the value right 7 places then clear all but the first 7 bits. Note that 14-bit numbers can be encoded by referencing the same parameter twice; for example, `%4^4` will insert the low-order followed by the high-order parts of parameter 4 into two successive data bytes.

Parameters are separated by commas, but there may be no spaces. The maximum number of parameters allowed is 9. Here is an example of definitions to send a full-resolution pitch bend command and to send a system exclusive command to change a DX7 parameter[7].

[7]My TX816 Owner’s Manual gives an incorrect format for the change parameter sysex command (according to the manual, there is no data in the message!) I am assuming that the data should be the last byte before the EOX and that there is no byte count. If you are reading this, assume that I have not tested this guess, nor have I tested this example.

```
* Define macro for pitch bend commands:
!DEF bend Ev %1 ^1

A ~bend(8192)  ** 8192 is "pitch bend off"

* Change the LFO SPEED:
*  SYSEX = F0, Yamaha = 43, Substatus/Channel = 1v,
*  Group# = 01, Parameter# = 9, Data = 0-99, EOX = F7
!DEF lfospeed F0 43 1v 01 09 %1 F7

* now use the definitions:
G4 ~bend(7567) N40
~lfospeed(30) N35
```

10.4.7. Control Ramps

The `!RAMP` command can specify a smooth control change from one value to another. It consists of a specification of the starting and ending values of some control change, a duration specifying how often to send a new value, and a duration specifying the total length of the ramp.

```
!RAMP X10 X100 Q W2
!RAMP ~23(10) ~23(50) U20 W
!RAMP ~lfo(15) ~lfo(35) U10
```

The first line says to ramp the volume control (controller number 7) from 10 to 100, changing at each quarter note for the duration of two whole notes. The second line says to ramp controller number 23 from value 10 to value 50, sending a new control change message every 20 time units. The overall duration of the ramp should be equivalent to a whole note (`W`). As shown in the third line, even system exclusive messages controlled by parameters can be specified. If the system exclusive message has more than one parameter, only one parameter may be ‘‘ramped’’; the others must remain the same. For example, the following would ramp the second parameter:

```
!RAMP ~mysysex(4,23,75) ~mysysex(4,100,75) U10 W
```

A rather curious and extreme use of macros and ramps is illustrated in the following example. The `noteon` macro starts a note, and `noteoff` ends it. Ramps can now be used to emit a series of notes with changing pitches or velocities. Since Adagio has no idea that these macros are turning on notes, it is up to the programmer to turn them off!

```
!DEF noteon 9v %1 %2
!DEF noteoff 8v %1 %2
~noteon(48,125)
~noteoff(48,126)
* turn on some notes
!RAMP ~noteon(36,125) ~noteon(60,125) Q W NW
* turn them off
!RAMP ~noteoff(60,50) ~noteoff(36,50) Q W NW
```

10.4.8. The !End Command

The special command `!END` marks the end of a score. Everything beyond that is ignored, for example:

```
* this is a score
C; D; E; F; G W
!END
since the score has ended, this text will be ignored
```

10.4.9. Calling C Routines

It is possible to call C routines from within Adagio scores when using specially linked versions, but this feature is disabled in Nyquist. The syntax is described here for completeness.

The `!CALL` command calls a C routine that can in turn invoke a complex sequence of operations. Below is a call to a trill routine, which is a standard routine in Adagio. The parameters are the base pitch of the trill, the total duration of the trill, the interval in semitones, the duration of each note of the trill, and the loudness. Notice that both numbers and Adagio notation can be used as parameters:

```
!CALL trill(A5,W,2,S,Lmf)  T278 V1
```

The parameter list should have no spaces, and parameters are separated by commas. Following the close parenthesis, you may specify other attributes such as the starting time and voice as shown in the example above.

A parameter may be an Adagio pitch specification, an Adagio duration, an Adagio loudness, a number, or an ASCII character within single quotes, e.g. `'a'` is equivalent to `97` because 97 is the decimal encoding of ‘‘a’’ in ASCII.

The `!CALL` may be followed by a limited set of attributes. These are time (`T`), voice (`V`), and next time (`N`). The `!CALL` is made at the current time if no time is specified, and the time of the next adagio command is the time of the `!CALL` unless a next time is specified. In other words, the default is `N0`.

10.4.10. Setting C Variables

In addition to calling C routines, there is another way in which scores can communicate with C. As with `!CALL`, specific C code must be linked before these commands can be used, and this is not supported in Nyquist. The `!SETI` command sets an integer variable to a value, and the `!SETV` command sets an element of an integer array. For example, the next line sets the variable `delay` to 200 and sets `transposition[5]` to -4 at time 200:

```
!SETI delay 200
!SETV transposition 5 -4  T200
```

As with the `!CALL` command, these commands perform their operations at particular times according to their place in the Adagio score. This makes it very easy to implement time-varying parameters that control various aspects of an interactive music system.

11. Linear Prediction Analysis and Synthesis

Nyquist provides functions to perform Linear Prediction Coding (LPC) analysis and synthesis. In simple terms, LPC analysis assumes that a sound is the result of an all-pole filter applied to a source with a flat spectrum. LPC is good for characterizing the general spectral shape of a signal, which may be time-varying as in speech sounds. For synthesis, any source can be filtered, allowing the general spectral shape of one signal (used in analysis) to be applied to any source (used in synthesis). A popular effect is to give vowel-like spectra to musical tones, creating an artificial (or sometimes natural) singing voice.

Examples of LPC analysis and synthesis can be found in the file `lpc_tutorial.htm`, which is part of the standard Nyquist release.

As with FFT processing, LPC analysis takes a sound as input and returns a stream of frames. Frames are returned from an object using the `:next` selector just as with FFT frames. An LPC frame is a list consisting of: *RMS1*, the energy of the input signal, *RMS2*, the energy of the residual signal, *ERR*, the square root of *RMS1*/*RMS2*, and *FILTER-COEFS*, an array of filter coefficients. To make code more readable and to avoid code dependence on the exact format of a frame, the functions `lpc-frame-rms1`, `lpc-frame-rms2`, `lpc-frame-err`, and `lpc-frame-filter-coefs` can be applied to a frame to obtain the respective fields.

The *z* transform of the filter is $H(z) = 1/A(z)$, where $A(z)$ is a polynomial of the form $A(z) = 1 + a_1 z + a_2 z + ... + a_p z$. The *FILTER-COEFS* array has the form `#(`$a_p\ a_{p-1}\ ...\ a_3\ a_2\ a_1$`)`.

The file `lpc.lsp` defines some useful classes and functions. The file is *not* automatically loaded with Nyquist, so you must execute `(load "lpc")` before using them.

11.1. LPC Classes and Functions

`make-lpanal-iterator(`*sound*`,` *framedur*`,` *skiptime*`,` *npoles*`)`
: Makes an iterator object, an instance of `lpanal-class`, that returns LPC frames from successive frames of samples in *sound*. The duration (in seconds) of each frame is given by *framedur*, a `FLONUM`. The skip size (in seconds) between successive frames is given by *skiptime*, a `FLONUM`. Typical values for *framedur* and *skiptime* are 0.08 and 0.04, giving 25 frames per second and a 50% frame overlap. The number of poles is given by *npoles*, a `FIXNUM`. The result is an object that responds to the `:next` selector by returning a frame as described above. `NIL` is returned when *sound* terminates. (Note that one or more of the last analysis windows may be padded with zeros. `NIL` is only returned when the corresponding window would begin after the termination time of the sound.)

`make-lpc-file-iterator(`*filename*`)`
: Another way to get LPC frames is to read them from a file. This function opens an ASCII file containing LPC frames and creates an iterator object, an instance of class `lpc-file-class` to access them. Create a file using `save-lpc-file` (see below).

`save-lpc-file(`*lpc-iterator*`,` *filename*`)`
: Create a file containing LPC frames. This file can be read by `make-lpc-file-iterator` (see above).

`show-lpc-data(`*lpc-iterator*`,` *iniframe*`,` *endframe*`[ ,`*poles?*`])`
: Print values of LPC frames from an LPC iterator object. The object is *lpc-iterator*, which is typically an instance of `lpanal-class` or `lpc-file-class`. Frames are numbered from zero, and only files starting at *iniframe* (a `FIXNUM`) and ending before *endframe* (also a `FIXNUM`) are printed. By default, only the values for *RMS1*, *RMS2*, and *ERR* are printed, but if

optional parameter *poles?* is non-NIL, then the LPC coefficients are also printed.

allpoles-from-lpc(*snd*, *lpc-frame*)
: A single LPC frame defines a filter. Use allpoles-from-lpc to apply this filter to *snd*, a SOUND. To obtain *lpc-frame*, a LIST containing an LPC frame, either send :next to an LPC iterator, or use nth-frame (see below). The result is a SOUND whose duration is the same as that of *snd*.

lpreson(*snd*, *lpc-iterator*, *skiptime*)
: Implements a time-varying all-pole filter controlled by a sequence of LPC frames from an iterator. The SOUND to be filtered is *snd*, and the source of LPC frames is *lpc-iterator*, typically an instance of lpanal-class or lpc-file-class. The frame period (in seconds) is given by *skiptime* (a FLONUM). This number does not have to agree with the *skiptime* used to analyze the frames. (Greater values will cause the filter evolution slow down, and smaller values will cause it to speed up.) The result is a SOUND. The duration of the result is the minimum of the duration of *snd* and that of the sequence of frames.

lpc-frame-rms1(*frame*)
: Get the energy of the input signal from a frame.

lpc-frame-rms2(*frame*)
: Get the energy of the residual from a frame.

lpc-frame-err(*frame*)
: Get the square root of *RMS1*/*RMS2* from a frame.

lpc-frame-filter-coefs(*frame*)
: Get the filter coefficients from a frame.

11.2. Low-level LPC Functions

The lowest-level Nyquist functions for LPC are

- snd-lpanal for analysis,
- snd-allpoles, an all-pole filter with fixed coefficients, and
- snd-lpreson, an all-pole filter that takes frames from an LPC iterator.

snd-lpanal(*samps*, *npoles*)
: Compute an LPC frame with *npoles* (a FIXNUM) poles from an ARRAY of samples (FLONUMS). Note that snd-fetch-array can be used to fetch a sequence of frames from a sound. Ordinarily, you should not use this function. Use make-lpanal-iterator instead.

snd-allpoles(*snd*, *lpc-coefs*, *gain*)
: A fixed all-pole filter. The input is *snd*, a SOUND. The filter coefficients are given by *lpc-coefs* (an ARRAY), and the filter gain is given by *gain*, a FLONUM. The result is a SOUND whose duration matches that of *snd*. Ordinarily, you should use allpoles-from-lpc instead (see above).

snd-lpreson(*snd*, *lpc-iterator*, *skiptime*)
: This function is identical to lpreson (see above).

12. Developing and Debugging in Nyquist

There are a number of tools, functions, and techniques that can help to debug Nyquist programs. Since these are described in many places throughout this manual, this chapter brings together many suggestions and techniques for developing code and debugging. You *really* should read this chapter before you spend too much time with Nyquist. Many problems that you will certainly run into are addressed here.

12.1. Debugging

Probably the most important debugging tool is the backtrace. When Nyquist encounters an error, it suspends execution and prints an error message. To find out where in the program the error occurred and how you got there, start by typing `(bt)`. This will print out the last several function calls and their arguments, which is usually sufficient to see what is going on.

In order for `(bt)` to work, you must have a couple of global variables set: `*tracenable*` is ordinarily set to `NIL`. If it is true, then a backtrace is automatically printed when an error occurs; `*breakenable*` must be set to `T`, as it enables the execution to be suspended when an error is encountered. If `*breakenable*` is `NIL` (false), then execution stops when an error occurs but the stack is not saved and you cannot get a backtrace. Finally, `bt` is just a macro to save typing. The actual backtrace function is `baktrace`, which takes an integer argument telling how many levels to print. All of these things are set up by default when you start Nyquist.

Since Nyquist sounds are executed with a lazy evaluation scheme, some errors are encountered when samples are being generated. In this case, it may not be clear which expression is in error. Sometimes, it is best to explore a function or set of functions by examining intermediate results. Any expression that yields a sound can be assigned to a variable and examined using one or more of: `s-plot`, `snd-print-tree`, and of course `play`. The `snd-print-tree` function prints a lot of detail about the inner representaion of the sound. Keep in mind that if you assign a sound to a global variable and then look at the samples (e.g. with `play` or `s-plot`), the samples will be retained in memory. At 4 bytes per sample, a big sound may use all of your memory and cause a crash.

Another technique is to use low sample rates so that it is easier to plot results or look at samples directly. The calls:

```
set-sound-srate(100)
set-control-srate(100)
```

set the default sample rates to 100, which is too slow for audio, but useful for examining programs and results. The function

`snd-samples(`*sound*`, `*limit*`)`

will convert up to *limit* samples from *sound* into a Lisp array. This is another way to look at results in detail.

The `trace` function is sometimes useful. It prints the name of a function and its arguments everytimg the function is called, and the result is printed when the function exits. To trace the osc function, type:

```
trace(osc)
```

and to stop tracing, type `untrace(osc)`.

If a variable needs a value or a function is undefined, and if `*breakenable*` was set, you will get a prompt where you can fix the error (by setting the variable or loading the function definition) and keep

going. At the debug (or break) prompt, your input must be in XLISP, not SAL syntax. Use `(co)`, short for `(continue)` to reevaluate the variable or function and continue execution.

When you finish debugging a particular call, you can ‘‘pop’’ up to the top level by typing `(top)`, a short name for `(top-level)`. There is a button named "Top" in the NyquistIDE that takes you back to the top level (ready to accept XLISP expressions), and another button named "SAL" that puts you back in SAL mode.

12.2. Useful Functions

`grindef(`*name*`)`
: Prints a formatted listing of a lisp function. This is often useful to quickly inspect a function without searching for it in source files. Do not forget to quote the name, e.g. `(grindef 'prod)`.

`args(`*name*`)`
: Similar to `grindef`, this function prints the arguments to a function. This may be faster than looking up a function in the documentation if you just need a reminder. For example, `(args 'lp)` prints ‘‘(LP S C),’’ which may help you to remember that the arguments are a sound (S) followed by the cutoff (C) frequency.

The following functions are useful short-cuts that might have been included in XLISP. They are so useful that they are defined as part of Nyquist.

`incf(`*symbol*`)`
: Increment *symbol* by one. This is a macro, and *symbol* can be anything that can be set by `setf`. Typically, *symbol* is a variable: ‘‘`(incf i)`,’’ but *symbol* can also be an array element: ‘‘`(incf (aref myarray i))`.’’

`decf(`*symbol*`)`
: Decrement *symbol* by one. (See `incf`, above.)

`push(`*val*`, `*lis*`)`
: Push *val* onto *lis* (a Lisp list). This is a macro that is equivalent to writing `(setf `*lis*` (cons `*val*` `*lis*`))`.

`pop(`*lis*`)`
: Remove (pop) the first item from *lis* (a Lisp list). This is a macro that is equivalent to writing `(setf `*lis*` (cdr `*lis*`))`. Note that the remaining list is returned, not the head of the list that has been popped. Retrieve the head of the list (i.e. the top of the stack) using `first` or, equivalently, `car`.

The following macros are useful control constructs.

`while(`*test*`, `*stmt1*`, `*stmt2*`, ...)`
: A conventional ‘‘while’’ loop. If *test* is true, perform the statements (*stmt1*, *stmt2*, etc.) and repeat. If *test* is false, return. This expression evaluates to NIL unless the expression `(return `*expr*`)` is evaluated, in which case the value of *expr* is returned.

`when(`*test*`, `*action*`)`
: A conventional ‘‘if-then’’ statement. If *test* is true, *action* is evaluated and returned. Otherwise, NIL is returned. (Use `if` or `cond` to implement ‘‘if-then-else’’ and more complex conditional forms.

It is often necessary to load a file *only if* it has not already been loaded. For example, the `pianosyn` library loads very slowly, so if some other file already loaded it, it would be good to avoid loading it again. How can you load a file once? Nyquist does not keep track of files that are loaded, but you must be loading a file to define some function, so the idea is to tell Nyquist "I require *function* from *file*"; if the

function does not yet exist, Nyquist satisfies the requirement by loading the file.

`require-from(`*fnsymbol*`,` *filename* `[,` *path*`])`
: Tests whether *fnsymbol*, an unquoted function name, is defined. If not, *filename*, a `STRING`, is loaded. Normally *fnsymbol* is a function that will be called from within the current file, and *filename* is the file that defines *fnsymbol*. The *path*, if a `STRING`, is prepended to *filename*. If *path* is `t` (true), then the directory of the current file is used as the path.

Sometimes it is important to load files relative to the current file. For example, the `lib/piano.lsp` library loads data files from the `lib/piano` directory, but how can we find out the full path of `lib`? The solution is:

`current-path()`
: Returns the full path name of the file that is currently being loaded (see `load`). Returns NIL if no file is being loaded.

Finally, there are some helpful math functions:

`real-random(`*from*`,` *to*`)`
: Returns a random `FLONUM` between *from* and *to*. (See also `rrandom`, which is equivalent to `(real-random 0 1)`).

`power(`*x*`,` *y*`)`
: Returns *x* raised to the *y* power.

13. Xmusic and Algorithmic Composition

Several Nyquist libraries offer support for algorithmic composition. Xmusic is a library for generating sequences and patterns of data. Included in Xmusic is the `score-gen` macro which helps to generate scores from patterns. Another important facility is the `distributions.lsp` library, containing many different random number generators.

13.1. Xmusic Basics

Xmusic is inspired by and based on Common Music by Rick Taube. Currently, Xmusic only implements patterns and some simple support for scores to be realized as sound by Nyquist. In contrast, Common Music supports MIDI and various other synthesis languages and includes a graphical interface, some visualization tools, and many other features. Common Music runs in Common Lisp and Scheme, but not XLISP, which is the base language for Nyquist.

Xmusic patterns are objects that generate data streams. For example, the `cycle-class` of objects generate cyclical patterns such as "1 2 3 1 2 3 1 2 3 ...", or "1 2 3 4 3 2 1 2 3 4 ...". Patterns can be used to specify pitch sequences, rhythm, loudness, and other parameters.

Xmusic functions are automatically loaded when you start Nyquist. To use a pattern object, you first create the pattern, e.g.

```
set pitch-source = make-cycle(list(c4, d4, e4, f4))
```

After creating the pattern, you can access it repeatedly with `next` to generate data, e.g.

```
play seqrep(i, 13, pluck(next(pitch-source), 0.2))
```

This will create a sequence of notes with the following pitches: c, d, e, f, c, d, e, f, c, d, e, f, c. If you evaluate this again, the pitch sequence will continue, starting on "d".

It is very important not to confuse the creation of a sequence with its access. Consider this example:

```
play seqrep(i, 13,
         pluck(next(make-cycle(list(c4, d4, e4, f4))), 0.2))
```

This looks very much like the previous example, but it only repeats notes on middle-C. The reason is that every time `pluck` is evaluated, `make-cycle` is called and creates a new pattern object. After the first item of the pattern is extracted with `next`, the cycle is not used again, and no other items are generated.

To summarize this important point, there are two steps to using a pattern. First, the pattern is created and stored in a variable using `setf`. Second, the pattern is accessed (multiple times) using `next`.

Patterns can be nested, that is, you can write patterns of patterns. In general, the `next` function does not return patterns. Instead, if the next item in a pattern is a (nested) pattern, `next` recursively gets the next item of the nested pattern.

While you might expect that each call to `next` would advance the top-level pattern to the next item, and descend recursively if necessary to the inner-most nesting level, this is not how `next` works. Instead, `next` remembers the last top-level item, and if it was a pattern, `next` continues to generate items from that same inner pattern until the end of the inner pattern's *period* is reached. The next paragraph explains the concept of the *period*.

The data returned by a pattern object is structured into logical groups called *periods*. You can get an entire period (as a list) by calling `next(`*pattern*`, t)`. For example:

```
set pitch-source = make-cycle(list(c4, d4, e4, f4))
print next(pitch-source, t)
```

This prints the list `(60 62 64 65)`, which is one period of the cycle.

You can also get explicit markers that delineate periods by calling `send(`*pattern*`, :next)`. In this case, the value returned is either the next item of the pattern, or the symbol `+eop+` if the end of a period has been reached. What determines a period? This is up to the specific pattern class, so see the documentation for specifics. You can override the ‘‘natural’’ period using the keyword `:for`, e.g.

```
set pitch-source = make-cycle(list(c4, d4, e4, f4), for: 3)
print next(pitch-source, t)
print next(pitch-source, t)
```

This prints the lists `(60 62 64) (65 60 62)`. Notice that these periods just restructure the stream of items into groups of 3.

Nested patterns are probably easier to understand by example than by specification. Here is a simple nested pattern of cycles:

```
set cycle-1 = make-cycle({a b c})
set cycle-2 = make-cycle({x y z})
set cycle-3 = make-cycle(list(cycle-1, cycle-2))
exec dotimes(i, 9, format(t, "~A ", next(cycle-3)))
```

This will print "A B C X Y Z A B C". Notice that the inner-most cycles `cycle-1` and `cycle-2` generate a period of items before the top-level `cycle-3` advances to the next pattern.

Before describing specific pattern classes, there are several optional parameters that apply in the creating of any pattern object. These are:

`:for`	The length of a period. This overrides the default by providing a numerical length. The value of this optional parameter may be a pattern that generates a sequence of integers that determine the length of each successive period. A period length may not be negative, but it may be zero.
`:name`	A pattern object may be given a name. This is useful if the `:trace` option is used.
`:trace`	If non-null, this optional parameter causes information about the pattern to be printed each time an item is generated from the pattern.

The built-in pattern classes are described in the following section.

13.2. Pattern Classes

13.2.1. cycle

The `cycle-class` iterates repeatedly through a list of items. For example, two periods of `make-cycle({a b c})` would be `(A B C) (A B C)`.

`make-cycle(`*items*`[ ,for:` *for*`] [, name:` *name*`,] [trace:` *trace*`])`

Make a cycle pattern that iterates over *items*. The default period length is the length of *items*. (See above for a description of the optional parameters.) If *items* is a pattern, a period of the pattern becomes the list from which items are generated. The list is replaced every period of the cycle.

13.2.2. line

The `line-class` is similar to the cycle class, but when it reaches the end of the list of items, it simply repeats the last item in the list. For example, two periods of `make-line({a b c})` would be `(A B C) (C C C)`.

`make-line(`*items*`[ ,for:` *for*`] [, name:` *name*`] [, trace:` *trace*`])`
Make a line pattern that iterates over *items*. The default period length is the length of *items*. As with `make-cycle`, *items* may be a pattern. (See above for a description of the optional parameters.)

13.2.3. random

The `random-class` generates items at random from a list. The default selection is uniform random with replacement, but items may be further specified with a weight, a minimum repetition count, and a maximum repetition count. Weights give the relative probability of the selection of the item (with a default weight of one). The minimum count specifies how many times an item, once selected at random, will be repeated. The maximum count specifies the maximum number of times an item can be selected in a row. If an item has been generated *n* times in succession, and the maximum is equal to *n*, then the item is disqualified in the next random selection. Weights (but not currently minima and maxima) can be patterns. The patterns (thus the weights) are recomputed every period.

`make-random(`*items*`[ ,for:` *for*`] [, name:` *name*`] [, trace:` *trace*`])`
Make a random pattern that selects from *items*. Any (or all) element(s) of *items* may be lists of the following form: `(`*value* `[:weight` *weight*`] [:min` *mincount*`] [:max` *maxcount*`]`, where *value* is the item (or pattern) to be generated, *weight* is the relative probability of selecting this item, *mincount* is the minimum number of repetitions when this item is selected, and *maxcount* is the maximum number of repetitions allowed before selecting some other item. The default period length is the length of *items*. If *items* is a pattern, a period from that pattern becomes the list from which random selections are made, and a new list is generated every period.

13.2.4. palindrome

The `palindrome-class` repeatedly traverses a list forwards and then backwards. For example, two periods of `make-palindrome({a b c})` would be `(A B C C B A) (A B C C B A)`. The `:elide` keyword parameter controls whether the first and/or last elements are repeated:

```
make-palindrome({a b c}, elide: nil)
     ;; generates A B C C B A A B C C B A ...

make-palindrome({a b c}, elide: t)
     ;; generates A B C B A B C B ...

make-palindrome({a b c}, elide: :first)
     ;; generates A B C C B A B C C B ...

make-palindrome({a b c}, elide: :last)
     ;; generates A B C B A A B C B A ...
```

`make-palindrome(`*items*`[ ,elide:` *elide*`,] [for:` *for*`] [, name:` *name*`] [, trace:` *trace*`])`
Generate items from list alternating in-order and reverse-order sequencing. The keyword parameter *elide* can have the values `:first`, `:last`, `t`, or `nil` to control repetition of the first and last elements. The *elide* parameter can also be a pattern, in which case it is evaluated every period. One period is one complete forward and backward traversal of the list. If *items* is a

pattern, a period from that pattern becomes the list from which random selections are made, and a new list is generated every period.

13.2.5. heap

The `heap-class` selects items in random order from a list without replacement, which means that all items are generated once before any item is repeated. For example, two periods of `make-heap({a b c})` might be `(C A B) (B A C)`. Normally, repetitions can occur even if all list elements are distinct. This happens when the last element of a period is chosen first in the next period. To avoid repetitions, the `:max` keyword argument can be set to 1. The `:max` keyword only controls repetitions from the end of one period to the beginning of the next. If the list contains more than one copy of the same value, it may be repeated within a period regardless of the value of `:max`.

`make-heap(`*items*`, [for:` *for*`] [, max:` *max*`] [, name:` *name*`] [, trace:` *trace*`])`
Generate items randomly from list without replacement. If *max* is 1, the first element of a new period will not be the same as the last element of the previous period, avoiding repetition. The default value of *max* is 2, meaning repetition is allowed. The period length is the length of *items*. If *items* is a pattern, a period from that pattern becomes the list from which random selections are made, and a new list is generated every period.

13.2.6. accumulation

The `accumulation-class` takes a list of values and returns the first, followed by the first two, followed by the first three, etc. In other words, for each list item, return all items from the first through the item. For example, if the list is (A B C), each generated period is (A A B A B C).

`make-accumulation(`*items*`[ ,name:` *name*`,] [trace:` *trace*`])`
For each item, generate items from the first to the item including the item. The period length is $(n^2 + n) / 2$ where n is the length of *items*. If *items* is a pattern, a period from that pattern becomes the list from which items are generated, and a new list is generated every period. Note that this is similar in name but different from `make-accumulate`.

13.2.7. copier

The `copier-class` makes copies of periods from a sub-pattern. For example, three periods of `make-copier(make-cycle({a b c}, for: 1), repeat: 2, merge: t)` would be `(A A) (B B) (C C)`. Note that entire periods (not individual items) are repeated, so in this example the `:for` keyword was used to force periods to be of length one so that each item is repeated by the `:repeat` count.

`make-copier(`*sub-pattern*`[ ,repeat:` *repeat*`] [, merge:` *merge*`,] [for:` *for*`] [, name:` *name*`] [, trace:` *trace*`])`
Generate a period from *sub-pattern* and repeat it *repeat* times. If *merge* is false (the default), each repetition of a period from *sub-pattern* results in a period by default. If *merge* is true (non-null), then all *repeat* repetitions of the period are merged into one result period by default. If the `:for` keyword is used, the same items are generated, but the items are grouped into periods determined by the `:for` parameter. If the `:for` parameter is a pattern, it is evaluated every result period. The *repeat* and *merge* values may be patterns that return a repeat count and a boolean value, respectively. If so, these patterns are evaluated initially and after each *repeat* copies are made (independent of the `:for` keyword parameter, if any). The *repeat* value returned by a pattern can also be negative. A negative number indicates how many periods of *sub-pattern* to skip. After skipping these patterns, new *repeat* and *merge* values are generated.

13.2.8. accumulate

The `accumulate-class` forms the sum of numbers returned by another pattern. For example, each period of `make-accumulate(make-cycle({1 2 -3}))` is `(1 3 0)`. The default output period length is the length of the input period.

`make-accumulate(`*sub-pattern*`[ ,for:` *for*`] [, max:` *maximum*`] [, min:` *minimum*`] [, name:` *name*`] [, trace:` *trace*`])`

Keep a running sum of numbers generated by *sub-pattern*. The default period lengths match the period lengths from *sub-pattern*. If *maximum* (a pattern or a number) is specified, and the running sum exceeds *maximum*, the running sum is reset to *maximum*. If *minimum* (a pattern or a number) is specified, and the running sum falls below *minimum*, the running sum is reset to *minimum*. If *minimum* is greater than *maximum*, the running sum will be set to one of the two values. Note that this is similar in name but not in function to `make-accumulation`.

13.2.9. sum

The `sum-class` forms the sum of numbers, one from each of two other patterns. For example, each period of `make-sum(make-cycle({1 2 3}), make-cycle({4 5 6}))` is `(5 7 9)`. The default output period length is the length of the input period of the first argument. Therefore, the first argument must be a pattern, but the second argument can be a pattern or a number.

`make-sum(`*x*`,` *y*`[ ,for:` *for*`] [, name:` *name*`] [, trace:` *trace*`])`

Form sums of items (which must be numbers) from pattern *x* and pattern or number *y*. The default period lengths match the period lengths from *x*.

13.2.10. product

The `product-class` forms the product of numbers, one from each of two other patterns. For example, each period of `make-product(make-cycle({1 2 3}), make-cycle({4 5 6}))` is `(4 10 18)`. The default output period length is the length of the input period of the first argument. Therefore, the first argument must be a pattern, but the second argument can be a pattern or a number.

`make-product(`*x*`,` *y*`[ ,for:` *for*`] [, name:` *name*`] [, trace:` *trace*`])`

Form products of items (which must be numbers) from pattern *x* and pattern or number *y*. The default period lengths match the period lengths from *x*.

13.2.11. eval

The `eval-class` evaluates an expression to produce each output item. The default output period length is 1.

`make-eval(`*expr*`[ ,for:` *for*`] [, name:` *name*`] [, trace:` *trace*`])`

Evaluate *expr* to generate each item. If *expr* is a pattern, each item is generated by getting the next item from *expr* and evaluating it.

13.2.12. length

The `length-class` generates periods of a specified length from another pattern. This is similar to using the `:for` keyword, but for many patterns, the `:for` parameter alters the points at which other patterns are generated. For example, if the palindrome pattern has an `:elide` pattern parameter, the value will be computed every period. If there is also a `:for` parameter with a value of 2, then `:elide` will be recomputed every 2 items. In contrast, if the palindrome (without a `:for` parameter) is embedded

in a *length* pattern with a lenght of 2, then the periods will all be of length 2, but the items will come from default periods of the palindrome, and therefore the `:elide` values will be recomputed at the beginnings of default palindrome periods.

`make-length(`*pattern*`, `*length-pattern*`, [name: `*name*`] [, trace: `*trace*`])`
Make a pattern of class `length-class` that regroups items generated by a *pattern* according to pattern lengths given by *length-pattern*. Note that *length-pattern* is not optional: There is no default pattern length and no `:for` keyword.

13.2.13. window

The `window-class` groups items from another pattern by using a sliding window. If the *skip* value is 1, each output period is formed by dropping the first item of the previous perioda and appending the next item from the pattern. The *skip* value and the output period length can change every period. For a simple example, if the period length is 3 and the skip value is 1, and the input pattern generates the sequence A, B, C, ..., then the output periods will be (A B C), (B C D), (C D E), (D E F), ...

`make-window(`*pattern*`, `*length-pattern*`, `*skip-pattern*`[ ,name: `*name*`] [, trace: `*trace*`])`
Make a pattern of class `window-class` that regroups items generated by a *pattern* according to pattern lengths given by *length-pattern* and where the period advances by the number of items given by *skip-pattern*. Note that *length-pattern* is not optional: There is no default pattern length and no `:for` keyword.

13.2.14. markov

The `markov-class` generates items from a Markov model. A Markov model generates a sequence of *states* according to rules which specify possible future states given the most recent states in the past. For example, states might be pitches, and each pitch might lead to a choice of pitches for the next state. In the `markov-class`, states can be either symbols or numbers, but not arbitrary values or patterns. This makes it easier to specify rules. However, symbols can be mapped to arbitrary values including pattern objects, and these become the actual generated items. By default, all future states are weighted equally, but weights may be associated with future states. A Markov model must be initialized with a sequence of past states using the `:past` keyword. The most common form of Markov model is a "first order Markov model" in which the future item depends only upon one past item. However, higher order models where the future items depend on two or more past items are possible. A "zero-order" Markov model, which depends on no past states, is essentially equivalent to the random pattern. As an example of a first-order Markov pattern, two periods of `make-markov({{a -> b c} {b -> c} {c -> a}}, past: {a})` might be `(C A C) (A B C)`.

`make-markov(`*rules*`[ ,past: `*past*`] [, produces: `*produces*`,] [for: `*for*`] [, name: `*name*`] [, trace: `*trace*`])`
Generate a sequence of items from a Markov process. The *rules* parameter has the form: `(`*prev1 prev2 ... prevn -> next1 next2 ... nextn*`)` where *prev1* through *prevn* represent a sequence of most recent (past) states. The symbol `*` is treated specially: it matches any previous state. If *prev1* through *prevn* (which may be just one state as in the example above) match the previously generated states, this rule applies. Note that every rule must specify the same number of previous states; this number is known as the order of the Markov model. The first rule in *rules* that applies is used to select the next state. If no rule applies, the next state is `NIL` (which is a valid state that can be used in rules). Assuming a rule applies, the list of possible next states is specified by *next1* through *nextn*. Notice that these are alternative choices for the next state, not a sequence of future states, and each rule can have any number of choices. Each choice may be the state itself (a symbol or a number), or the choice may be a list consisting of the state and a weight.

The weight may be given by a pattern, in which case the next item of the pattern is obtained every time the rule is applied. For example, this rules says that if the previous states were A and B, the next state can be A with a weight of 0.5 or C with an implied weight of 1: `(A B -> (A 0.5) C)`. The default length of the period is the length of *rules*. The *past* parameter must be provided. It is a list of states whose length matches the order of the Markov model. The keyword parameter *produces* may be used to map from state symbols or numbers to other values or patterns. The parameter is a list of alternating symbols and values. For example, to map A to 69 and B to 71, use `list(quote(a), 69, quote(b), 71)`. You can also map symbols to patterns, for example `list(quote(a), make-cycle({57 69}), quote(b), make-random({59 71}))`. The next item of the pattern is is generated each time the Markov model generates the corresponding state. Finally, the *produces* keyword can be `:eval`, which means to evaluate the Markov model state. This could be useful if states are Nyquist global variables such as `C4, CS4, D4, ]..., which evaluate to numerical values (60, 61, 62, ....`

`markov-create-rules(`*sequence*`, `*order*`[ ,`*generalize*`])`

Generate a set of rules suitable for the `make-markov` function. The *sequence* is a ‘‘typical’’ sequence of states, and *order* is the order of the Markov model. It is often the case that a sample sequence will not have a transition from the last state to any other state, so the generated Markov model can reach a ‘‘dead end’’ where no rule applies. This might lead to an infinite stream of NIL’s. To avoid this, the optional parameter *generalize* can be set to `t` (true), indicating that there should be a fallback rule that matches any previous states and whose future states are weighted according to their frequency in *sequence*. For example, if sequence contains 5 A’s, 5 B’s and 10 G’s, the default rule will be `(* -> (A 5) (B 5) (G 10))`. This rule will be appended to the end so it will only apply if no other rule does.

13.3. Random Number Generators

The `distributions.lsp` library implements random number generators that return random values with various probability distributions. Without this library, you can generate random numbers with *uniform* distributions. In a uniform distribution, all values are equally likely. To generate a random integer in some range, use `random`. To generate a real number (FLONUM) in some range, use `real-random` (or `rrandom` if the range is 0-1). But there are other interesting distributions. For example, the Gaussian distribution is often used to model real-world errors and fluctuations where values are clustered around some central value and large deviations are more unlikely than small ones. See Dennis Lorrain, "A Panoply of Stochastic ’Canons’," *Computer Music Journal* vol. 4, no. 1, 1980, pp. 53-81.

In most of the random number generators described below, there are optional parameters to indicate a maximum and/or minimum value. These can be used to truncate the distribution. For example, if you basically want a Gaussian distribution, but you never want a value greater than 5, you can specify 5 as the maximum value. The upper and lower bounds are implemented simply by drawing a random number from the full distribution repeatedly until a number falling into the desired range is obtained. Therefore, if you select an acceptable range that is unlikely, it may take Nyquist a long time to find each acceptable random number. The intended use of the upper and lower bounds is to weed out values that are already fairly unlikely.

`linear-dist(`*g*`)`

Return a `FLONUM` value from a linear distribution, where the probability of a value decreases linearly from zero to *g* which must be greater than zero. (See Figure 7.) The linear distribution is useful for generating for generating time and pitch intervals.

`exponential-dist(`*delta*`[ ,`*high*`])`

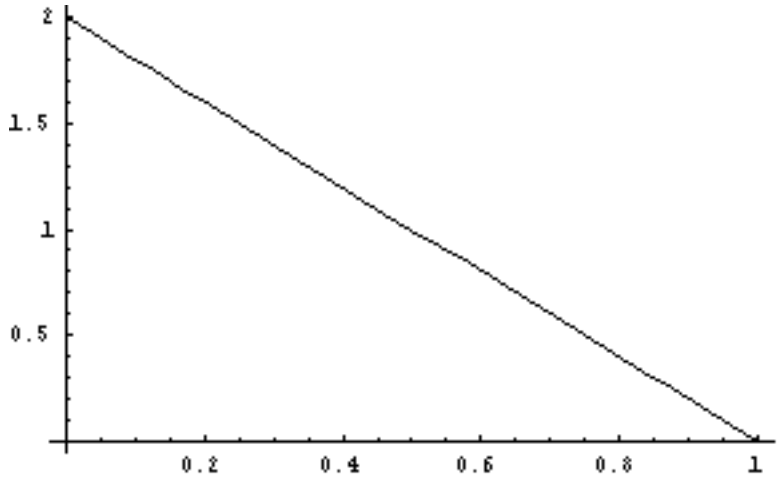

Figure 7: The Linear Distribution, *g* = 1.

Return a `FLONUM` value from an exponential distribution. The initial downward slope is steeper with larger values of *delta*, which must be greater than zero. (See Figure 8. The optional *high* parameter puts an artificial upper bound on the return value. The exponential distribution generates values greater than 0, and can be used to generate time intervals. Natural random intervals such as the time intervals between the release of atomic particles or the passing of yellow volkswagons in traffic have exponential distributions. The exponential distribution is memory-less: knowing that a random number from this distribution is greater than some value (e.g. a note duration is at least 1 second) tells you nothing new about how soon the note will end. This is a continuous distribution, but `geometric-dist` (described below) implements the discrete form.

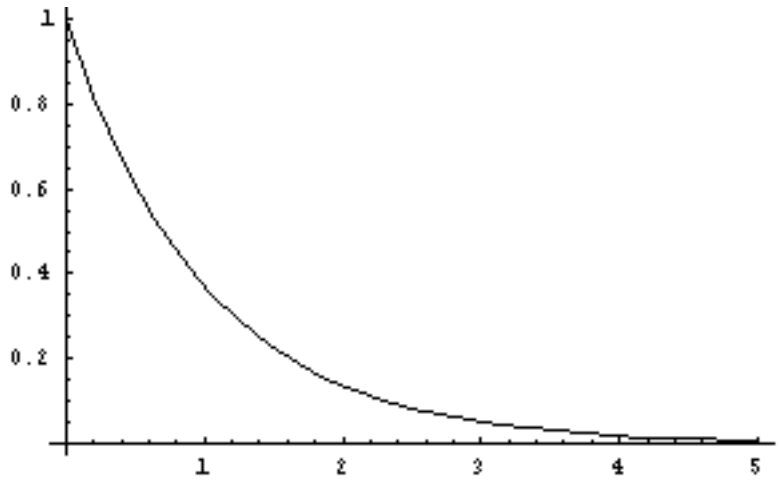

Figure 8: The Exponential Distribution, *delta* = 1.

`gamma-dist(`*nu*`[ ,`*high*`])`

Return a `FLONUM` value from a Gamma distribution. The value is greater than zero, has a mean of *nu* (a `FIXNUM` greater than zero), and a mode (peak) of around *nu* - 1. The optional *high* parameter puts an artificial upper bound on the return value.

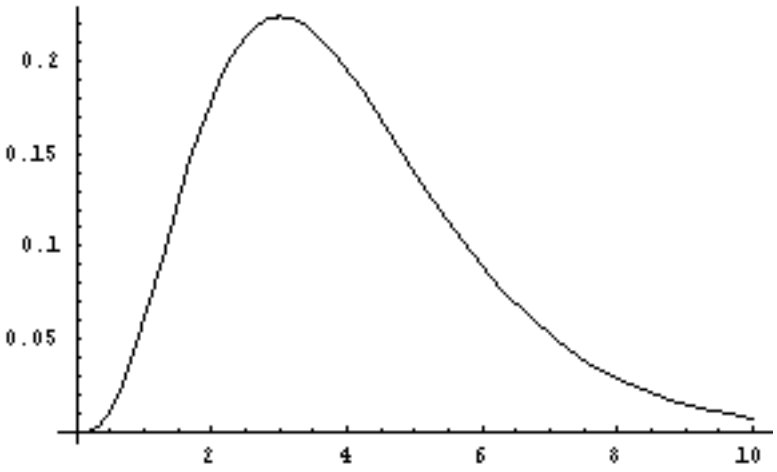

Figure 9: The Gamma Distribution, *nu* = 4.

`bilateral-exponential-dist(`*xmu*`, `*tau*`[ ,`*low*`] [, `*high*`])`

Returns a `FLONUM` value from a bilateral exponential distribution, where *xmu* is the center of the double exponential and *tau* controls the spread of the distribution. A larger *tau* gives a wider distribution (greater variance), and *tau* must be greater than zero. The *low* and *high* parameters give optional artificial bounds on the minimum and maximum output values, respectively. This distribution is similar to the exponential, except it is centered at 0 and can output negative values as well. Like the exponential, it can be used to generate time intervals; however, it might be necessary to add a lower bound so as not to compute a negative time interval.

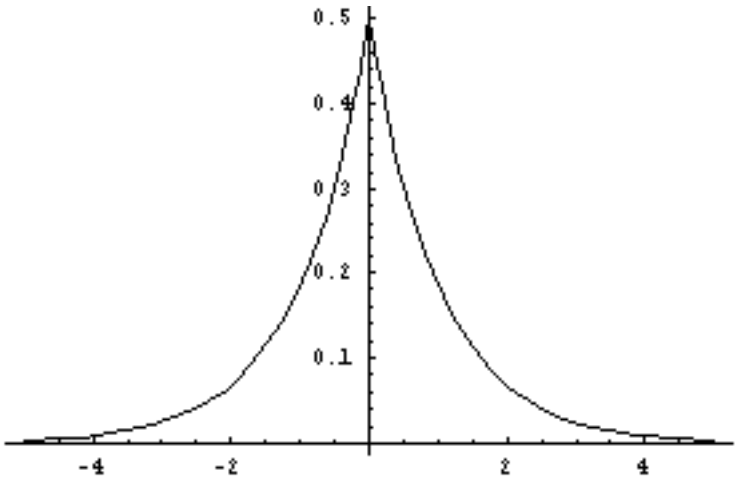

Figure 10: The Bilateral Exponential Distribution.

`cauchy-dist(`*tau*`[ ,`*low*`] [, `*high*`])`

Returns a `FLONUM` from the Cauchy distribution, a symetric distribution with a high peak at zero and a width (variance) that increases with parameter *tau*, which must be greater than zero. The *low* and *high* parameters give optional artificial bounds on the minimum and maximum output values, respectively.

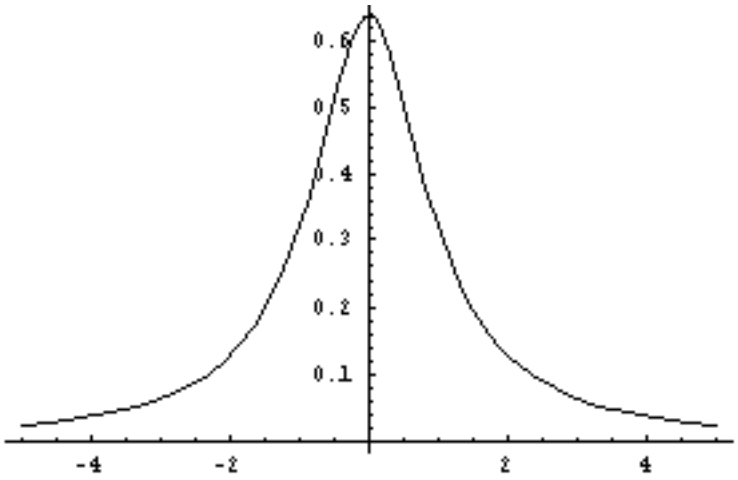

Figure 11: The Cauchy Distribution, *tau* = 1.

`hyperbolic-cosine-dist( [`*low*`] [, `*high*`])`

Returns a `FLONUM` value from the hyperbolic cosine distribution, a symetric distribution with its peak at zero. The *low* and *high* parameters give optional artificial bounds on the minimum and maximum output values, respectively.

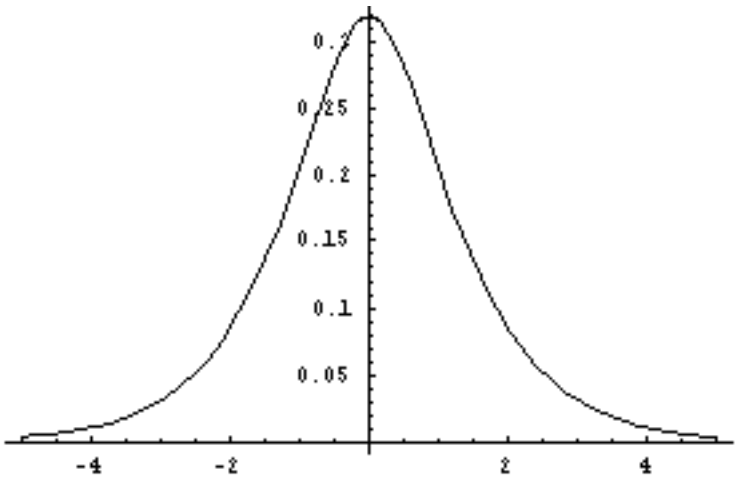

Figure 12: The Hyperbolic Cosine Distribution.

`logistic-dist(`*alpha*`, `*beta*`[ ,`*low*`] [, `*high*`])`

Returns a `FLONUM` value from the logistic distribution, which is symetric about the mean. The *alpha* parameter primarily affects dispersion (variance), with larger values resulting in values closer to the mean (less variance), and the *beta* parameter primarily influences the mean. The *low* and *high* parameters give optional artificial bounds on the minimum and maximum output values, respectively.

`arc-sine-dist()`

Returns a `FLONUM` value from the arc sine distribution, which outputs values between 0 and 1. It is symetric about the mean of 1/2, but is more likely to generate values closer to 0 and 1.

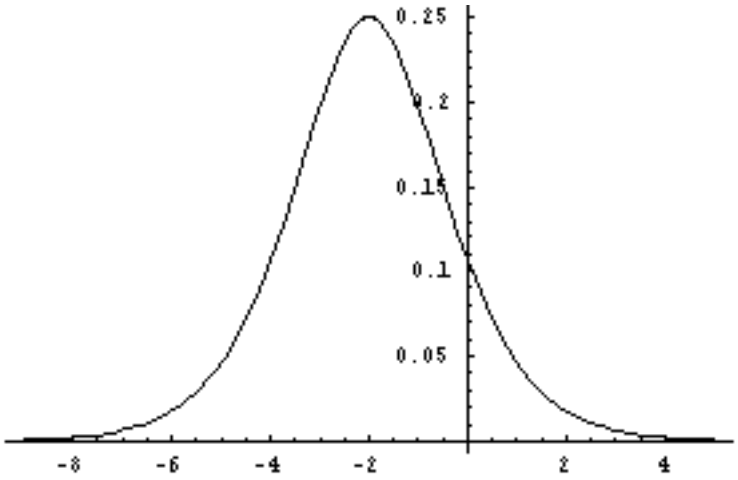

Figure 13: The Logistic Distribution, alpha = 1, beta = 2.

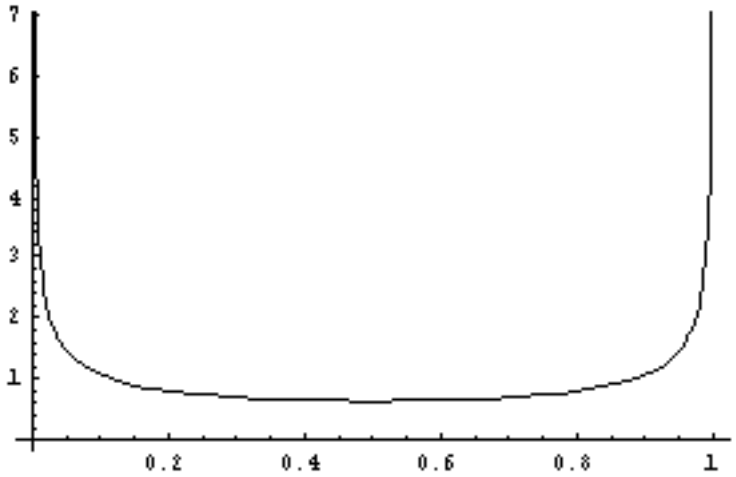

Figure 14: The Arc Sine Distribution.

`gaussian-dist(`*xmu*`, `*sigma*`[ ,`*low*`] [, `*high*`])`
Returns a `FLONUM` value from the Gaussian or Gauss-Laplace distribution, a linear function of the normal distribution. It is symetric about the mean of *xmu*, with a standard deviation of *sigma*, which must be greater than zero. The *low* and *high* parameters give optional artificial bounds on the minimum and maximum output values, respectively.

`beta-dist(`*a*`, `*b*`)`
Returns a `FLONUM` value from the Beta distribution. This distribution outputs values between 0 and 1, with outputs more likely to be close to 0 or 1. The parameter *a* controls the height (probability) of the right side of the distribution (at 1) and *b* controls the height of the left side (at 0). The distribution is symetric about 1/2 when *a* = *b*.

`bernoulli-dist(`*px1*`[ ,`*x1*`] [, `*x2*`])`
Returns either *x1* (default value is 1) with probability *px1* or *x2* (default value is 0) with probability 1 - *px1*. The value of *px1* should be between 0 and 1. By convention, a result of *x1* is

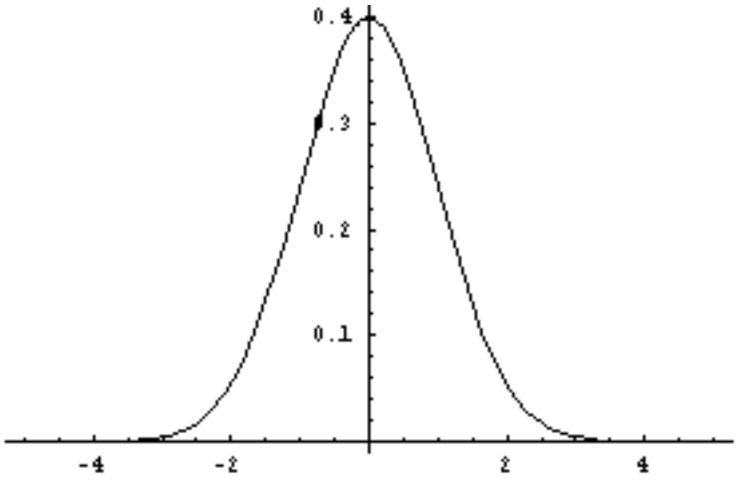

Figure 15: The Gauss-Laplace (Gaussian) Distribution, *xmu* = 0, *sigma* = 1.

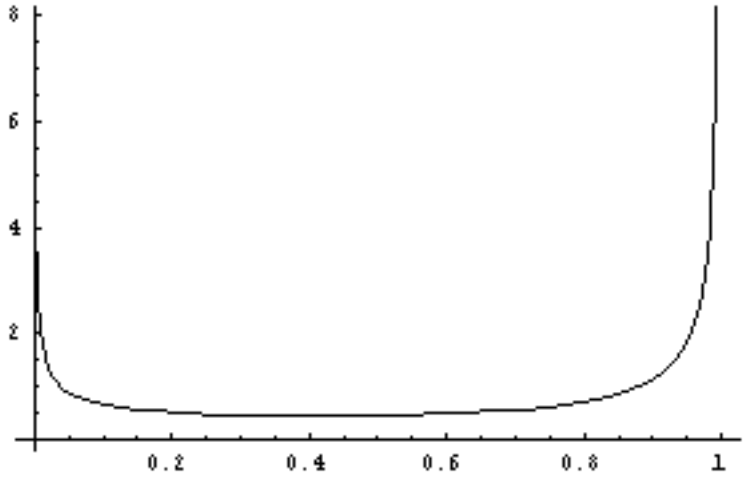

Figure 16: The Beta Distribution, *alpha* = .5, *beta* = .25.

viewed as a success while *x2* is viewed as a failure.

`(binomial-dist` *n p*

Returns a `FIXNUM` value from the binomial distribution, where *n* is the number of Bernoulli trials run (a `FIXNUM`) and *p* is the probability of success in the Bernoulli trial (a `FLONUM` from 0 to 1). The mean is the product of *n* and *p*.

`geometric-dist(`*p*`)`

Returns a `FIXNUM` value from the geometric distribution, which is defined as the number of failures before a success is achieved in a Bernoulli trial with probability of success *p* (a `FLONUM` from 0 to 1).

`poisson-dist(`*delta*`)`

Returns a `FIXNUM` value from the Poisson distribution with a mean of *delta* (a `FIXNUM`). The Poisson distribution is often used to generate a sequence of time intervals, resulting in random but often pleasing rhythms.

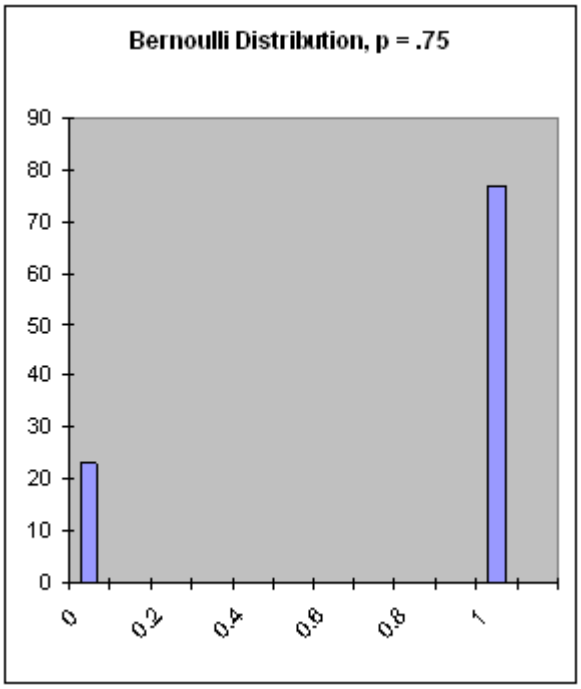

Figure 17: The Bernoulli Distribution, *px1* = .75.

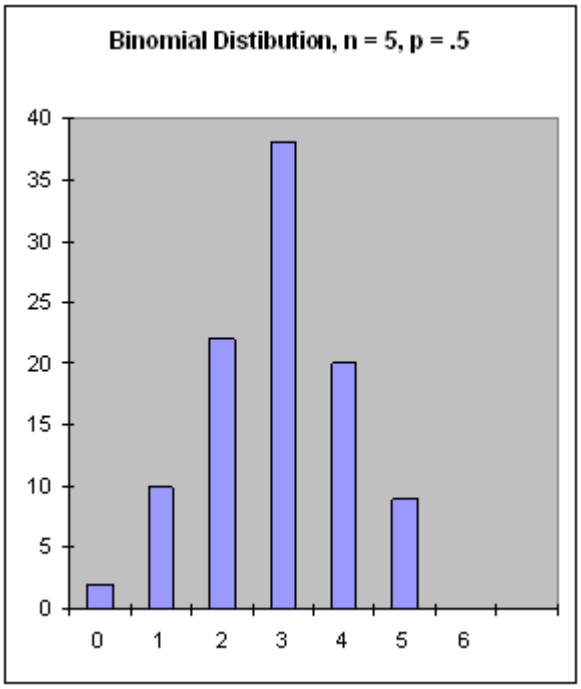

Figure 18: The Binomial Distribution, $n = 5$, $p = .5$.

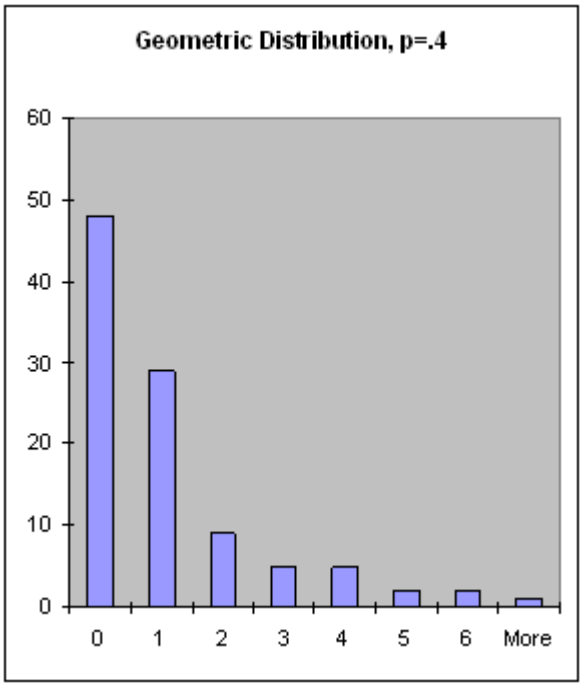

Figure 19: The Geometric Distribution, *p* = .4.

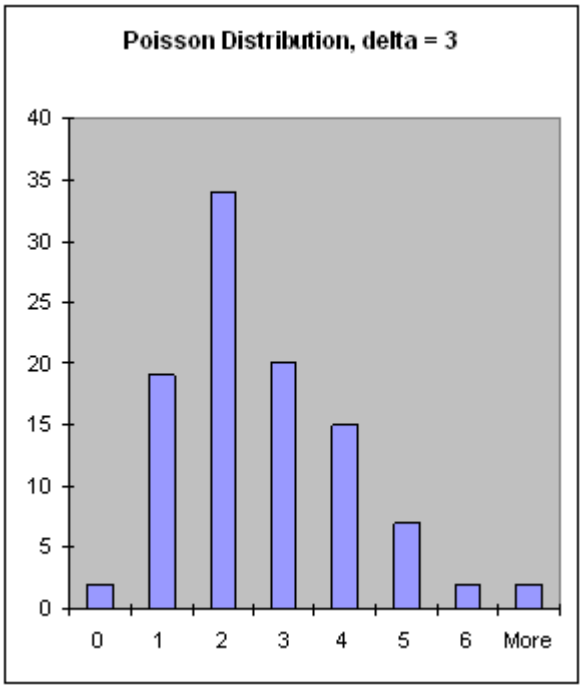

Figure 20: The Poisson Distribution, *delta* = 3.

13.4. Score Generation and Manipulation

A common application of pattern generators is to specify parameters for notes. (It should be understood that ‘‘notes’’ in this context means any Nyquist behavior, whether it represents a conventional note, an abstract sound object, or even some micro-sound event that is just a low-level component of a hierarchical sound organization. Similarly, ‘‘score’’ should be taken to mean a specification for a sequence of these ‘‘notes.’’) The `score-gen` macro (defined by loading `xm.lsp`) establishes a convention for representing scores and for generating them using patterns.

The `timed-seq` macro, described in Section 7.4, already provides a way to represent a ‘‘score’’ as a list of expressions. The Xmusic representation goes a bit further by specifying that *all notes are specified by an alternation of keywords and values, where some keywords have specific meanings and interpretations.*

The basic idea of `score-gen` is you provide a template for notes in a score as a set of keywords and values. For example,

```
set pitch-pattern = make-cycle(list(c4, d4, e4, f4))
score-gen(dur: 0.4, name: quote(my-sound),
          pitch: next(pitch-pattern), score-len: 9)
```

generates a score of 9 notes as follows:

```
((0 0 (SCORE-BEGIN-END 0 3.6))
 (0 0.4 (MY-SOUND :PITCH 60))
 (0.4 0.4 (MY-SOUND :PITCH 62))
 (0.8 0.4 (MY-SOUND :PITCH 64))
 (1.2 0.4 (MY-SOUND :PITCH 65))
 (1.6 0.4 (MY-SOUND :PITCH 60))
 (2 0.4 (MY-SOUND :PITCH 62))
 (2.4 0.4 (MY-SOUND :PITCH 64))
 (2.8 0.4 (MY-SOUND :PITCH 65))
 (3.2 0.4 (MY-SOUND :PITCH 60)))
```

The use of keywords like `:PITCH` helps to make scores readable and easy to process without specific knowledge of about the functions called in the score. For example, one could write a transpose operation to transform all the `:pitch` parameters in a score without having to know that pitch is the first parameter of `pluck` and the second parameter of `piano-note`. Keyword parameters are also used to give flexibility to note specification with `score-gen`. Since this approach requires the use of keywords, the next section is a brief explanation of how to define functions that use keyword parameters.

13.4.1. Keyword Parameters

Keyword parameters are parameters whose presence is indicated by a special symbol, called a keyword, followed by the actual parameter. Keyword parameters in SAL have default values that are used if no actual parameter is provided by the caller of the function. (See Appendix IV to learn about keywords in XLISP.)

To specify that a parameter is a keyword parameter, use a keyword symbol (one that ends in a colon) followed by a default value. For example, here is a function that accepts keyword parameters and invokes the `pluck` function:

```
define function k-pluck(pitch: 60, dur: 1)
  return pluck(pitch, dur)
```

Now, we can call k-pluck with keyword parameters. The keywords are simply the formal parameter

names with a prepended colon character (`:pitch` and `:dur` in this example), so a function call would look like:

```
pluck(pitch: c3, dur: 3)
```

Usually, it is best to give keyword parameters useful default values. That way, if a parameter such as `dur:` is missing, a reasonable default value (1) can be used automatically. It is never an error to omit a keyword parameter, but the called function can check to see if a keyword parameter was supplied or not. Because of default values, we can call `k-pluck(pitch: c3)` with no duration, `k-pluck(dur: 3)` with only a duration, or even `k-pluck()` with no parameters.

In XLISP, there is additional syntax to specify an alternate symbol to be used as the keyword and to allow the called function to determine whether or not a keyword parameter was supplied, but these features are little-used. See the XLISP manual for details.

13.4.2. Using score-gen

The `score-gen` macro computes a score based on keyword parameters. Some keywords have a special meaning, while others are not interpreted but merely placed in the score. The resulting score can be synthesized using `timed-seq` (see Section 7.4).

The form of a call to `score-gen` is simply `score-gen(`*k1: e1, k2: e2, ...)*, where the *k*'s are keywords and the *e*'s are expressions. A score is generated by evaluating the expressions once for each note and constructing a list of keyword-value pairs. A number of keywords have special interpretations. The rules for interpreting these parameters will be explained through a set of "How do I ..." questions:

How many notes will be generated? The keyword parameter `:score-len` specifies an upper bound on the number of notes. The keyword `:score-dur` specifies an upper bound on the starting time of the last note in the score. (To be more precise, the `:score-dur` bound is reached when the default starting time of the next note is greater than or equal to the `:score-dur` value. This definition is necessary because note times are not strictly increasing.) When either bound is reached, score generation ends. At least one of these two parameters must be specified or an error is raised. These keyword parameters are evaluated just once and are not copied into the parameter lists of generated notes.

What is the duration of generated notes? The keyword `:dur` defaults to 1 and specifies the nominal duration in seconds. Since the generated note list is compatible with `timed-seq`, the starting time and duration (to be precise, the *stretch factor*) are not passed as parameters to the notes. Instead, they control the Nyquist environment in which the note will be evaluated.

What is the start time of a note? The default start time of the first note is zero. Given a note, the default start time of the next note is the start time plus the inter-onset time, which is given by the `:ioi` parameter. If no `:ioi` parameter is specified, the inter-onset time defaults to the duration, given by `:dur`. In all cases, the default start time of a note can be overridden by the keyword parameter `:time`.

When does the score begin and end? The behavior `SCORE-BEGIN-END` contains the beginning and ending of the score (these are used for score manipulations, e.g. when scores are merged, their begin times can be aligned.) When `timed-seq` is used to synthesize a score, the `SCORE-BEGIN-END` marker is not evaluated. The `score-gen` macro inserts a ‘‘note’’ of the form `(0 0 (SCORE-BEGIN-END` *begin-time end-time*`))` at the time given by the `:begin` keyword, with *begin-time* and *end-time*

determined by the `:begin` and `:end` keyword parameters, respectively. If the *:begin* keyword is not provided, the score begins at zero. If the `:end` keyword is not provided, the score ends at the default start time of what would be the next note after the last note in the score (as described in the previous paragraph). Note: if `:time` is used to compute note starting times, and these times are not increasing, it is strongly advised to use `:end` to specify an end time for the score, because the default end time may be anywhere in the middle of the generated sequence.

What function is called to synthesize the note? The `:name` parameter names the function. Like other parameters, the value can be any expression, including something like `next(fn-name-pattern)`, allowing function names to be recomputed for each note. The default value is `note`.

Can I make parameters depend upon the starting time or the duration of the note? Parameter expressions can use the variable `sg:time` to access the start time of the note, `sg:ioi` to access the inter-onset time, and `sg:dur` to access the duration (stretch factor) of the note. Also, `sg:count` counts how many notes have been computed so far, starting at 0. The order of computation is: `sg:time` first, then `sg:ioi` and `sg:dur`, so for example, an expression to compute `sg:dur` can depend on `sg:ioi`.

Can parameters depend on each other? The keyword `:pre` introduces an expression that is evaluated before each note, and `:post` provides an expression to be evaluated after each note. The `:pre` expression can assign one or more global variables which are then used in one or more expressions for parameters.

How do I debug `score-gen` *expressions?* You can set the `:trace` parameter to true (`t`) to enable a print statement for each generated note.

How can I save scores generated by `score-gen` *that I like?* If the keyword parameter `:save` is set to a symbol, the global variable named by the symbol is set to the value of the generated sequence. Of course, the value returned by `score-gen` is just an ordinary list that can be saved like any other value.

In summary, the following keywords have special interpretations in `score-gen`: `:begin`, `:end`, `:time`, `:dur`, `:name`, `:ioi`, `:trace`, `:save`, `:score-len`, `:score-dur`, `:pre`, `:post`. All other keyword parameters are expressions that are evaluated once for each note and become the parameters of the notes.

13.4.3. Score Manipulation

Nyquist encourages the representation of music as executable programs, or *behaviors*, and there are various ways to modify behaviors, including time stretching, transposition, etc. An alternative to composing executable programs is to manipulate scores as editable data. Each approach has its strengths and weaknesses. This section describes functions intended to manipulate Xmusic scores as generated by, or at least in the form generated by, `score-gen`. Recall that this means scores are lists of events (e.g. notes), where events are three-element lists of the form (*time duration expression*, and where *expression* is a standard lisp function call where all parameters are keyword parameters. In addition, the first ‘‘note’’ may be the special `SCORE-BEGIN-END` expression. If this is missing, the score begins at zero and ends at the end of the last note.

For convenience, a set of functions is offered to access properties of events (or notes) in scores. Although lisp functions such as `car`, `cadr`, and `caddr` can be used, code is more readable when more mnemonic functions are used to access events.

`event-time(`*event*`)`
Retrieve the time field from an event.

`event-set-time(`*event*`, `*time*`)`
Construct a new event where the time of *event* is replaced by *time*.

`event-dur(`*event*`)`
Retrieve the duration (i.e. the stretch factor) field from an event.

`event-set-dur(`*event*`, `*dur*`)`
Construct a new event where the duration (or stretch factor) of *event* is replaced by *dur*.

`event-expression(`*event*`)`
Retrieve the expression field from an event.

`event-set-expression(`*event*`, `*dur*`)`
Construct a new event where the expression of *event* is replaced by *expression*.

`event-end(`*event*`)`
Retrieve the end time of *event*, its time plus its duration.

`expr-has-attr(`*expression*`, `*attribute*`)`
Test whether a score event *expression* has the given *attribute*.

`expr-get-attr(`*expression*`, `*attribute*`, [`*default*`])`
Get the value of the given *attribute* from a score event *expression*. If *attribute* is not present, return *default* if specified, and otherwise `nil`.

`expr-set-attr(`*expr*`, `*attribute*`, `*value*`)`
Construct a new expression identical to *expr* except that the *attribute* has *value*.

`event-has-attr(`*event*`, `*attribute*`)`
Test whether a given score *event*'s expression has the given *attribute*.

`event-get-attr(`*event*`, `*attribute*`, [`*default*`])`
Get the value of the given *attribute* from a score *event*'s expression. If *attribute* is not present, return *default* if specified, and otherwise `nil`.

`event-set-attr(`*event*`, `*attribute*`, `*value*`)`
Construct a new event identical to *event* except that the *attribute* has *value*.

Functions are provided to shift the starting times of notes, stretch times and durations, stretch only durations, add an offset to a keyword parameter, scale a keyword parameter, and other manipulations. Functions are also provided to extract ranges of notes, notes that match criteria, and to combine scores. Most of these functions (listed below in detail) share a set of keyword parameters that optionally limit the range over which the transformation operates. The `:from-index` and `:to-index` parameters specify the index of the first note and the index of the last note to be changed. If these numbers are negative, they are offsets from the end of the score, e.g. -1 denotes the last note of the score. The `:from-time` and `:to-time` indicate a range of starting times of notes that will be affected by the manipulation. Only notes whose time is greater than or equal to the *from-time* and *strictly less than* the *to-time* are modified. If both index and time ranges are specified, only notes that satisfy *both* constraints are selected.

`score-sorted(`*score*`)`
Test if *score* is sorted.

`score-sort(`*score*`[ ,`*copy-flag*`])`
Sort the notes in a score into start-time order. If copy-flag is nil, this is a destructive operation which should only be performed if the top-level score list is a fresh copy that is not shared by any other variables. (The *copy-flag* is intended for internal system use only.) For the following operations, it is assumed that scores are sorted, and all operations return a sorted score.

`score-shift(`*score*`, `*offset*`[ ,from-index: `*i*`,] [to-index: `*j*`] [, from-time: `*x*`]`

[, to-time: *y*])
Add a constant *offset* to the starting time of a set of notes in *score*. By default, all notes are modified, but the range of notes can be limited with the keyword parameters. The begin time of the score is not changed, but the end time is increased by *offset*. The original score is not modified, and a new score is returned.

score-stretch(*score*, *factor*, [dur: *dur-flag*] [, time: *time-flag*] [, from-index: *i*,] [to-index: *j*] [, from-time: *x*] [, to-time: *y*])
Stretch note times and durations by *factor*. The default *dur-flag* is non-null, but if *dur-flag* is null, the original durations are retained and only times are stretched. Similarly, the default *time-flag* is non-null, but if *time-flag* is null, the original times are retained and only durations are stretched. If both *dur-flag* and *time-flag* are null, the score is not changed. If a range of notes is specified, times are scaled within that range, and notes after the range are shifted so that the stretched region does not create a "hole" or overlap with notes that follow. If the range begins or ends with a time (via :from-time and :to-time), time stretching takes place over the indicated time interval independent of whether any notes are present or where they start. In other words, the ‘‘rests’’ are stretched along with the notes. The original score is not modified, and a new score is returned.

score-transpose(*score*, *keyword*, *amount*[,from-index: *i*,] [to-index: *j*] [, from-time: *x*] [, to-time: *y*])
For each note in the score and in any indicated range, if there is a keyword parameter matching *keyword* and the parameter value is a number, increment the parameter value by *amount*. For example, to tranpose up by a whole step, write (score-transpose 2 :pitch *score*). The original score is not modified, and a new score is returned.

score-scale(*score*, *keyword*, *amount*, [from-index: *i*,] [to-index: *j*] [, from-time: *x*] [, to-time: *y*])
For each note in the score and in any indicated range, if there is a keyword parameter matching *keyword* and the parameter value is a number, multiply the parameter value by *amount*. The original score is not modified, and a new score is returned.

score-sustain(*score*, *factor*, [from-index: *i*,] [to-index: *j*] [, from-time: *x*] [, to-time: *y*])
For each note in the score and in any indicated range, multiply the duration (stretch factor) by *amount*. This can be used to make notes sound more legato or staccato, and does not change their starting times. The original score is not modified, and a new score is returned.

score-voice(*score*, *replacement-list*, [from-index: *i*,] [to-index: *j*] [, from-time: *x*] [, to-time: *y*])
For each note in the score and in any indicated range, replace the behavior (function) name using *replacement-list*, which has the format: ((*old1 new1*) (*old2 new2*) ...), where *oldi* indicates a current behavior name and *newi* is the replacement. If *oldi* is *, it matches anything. For example, to replace my-note-1 by trombone and my-note-2 by horn, use score-voice(*score*, {{my-note-1 trombone} {my-note-2 horn}}). To replace all instruments with piano, use score-voice(*score*, {{* piano}}). The original score is not modified, and a new score is returned.

score-merge(*score1*, *score2*, ...)
Create a new score containing all the notes of the parameters, which are all scores. The resulting notes retain their original times and durations. The merged score begin time is the minimum of the begin times of the parameters and the merged score end time is the maximum of the end times of the parameters. The original scores are not modified, and a new score is returned.

score-append(*score1*, *score2*, ...)
Create a new score containing all the notes of the parameters, which are all scores. The begin time of the first score is unaltered. The begin time of each other score is aligned to the end time of the previous score; thus, scores are ‘‘spliced’’ in sequence. The original scores are not modified, and a new score is returned.

`score-select(`*score*`, `*predicate*`, [from-index: `*i*`] [, to-index: `*j*`,] [from-time: `*x*`] [, to-time: `*y*`] [, reject: `*flag*`])`
Select (or reject) notes to form a new score. Notes are selected if they fall into the given ranges of index and time *and* they satisfy *predicate*, a function of three parameters that is applied to the start time, duration, and the expression of the note. Alternatively, *predicate* may be `t`, indicating that all notes in range are to be selected. The selected notes along with the existing score begin and end markers, are combined to form a new score. Alternatively, if the `:reject` parameter is non-null, the notes *not* selected form the new score (in other words the selected notes are rejected or removed to form the new score). The original score is not modified, and a new score is returned.

`score-set-begin(`*score*`, `*time*`)`
The begin time from the *score*'s `SCORE-BEGIN-END` marker is set to *time*. The original score is not modified, and a new score is returned.

`score-get-begin(`*score*`)`
Return the begin time of the *score*.

`score-set-end(`*score*`, `*time*`)`
The end time from the *score*'s `SCORE-BEGIN-END` marker is set to *time*. The original score is not modified, and a new score is returned.

`score-get-end(`*score*`)`
Return the end time of the *score*.

`score-must-have-begin-end(`*score*`)`
If *score* does not have a begin and end time, construct a score with a `SCORE-BEGIN-END` expression and return it. If score already has a begin and end time, just return the score. The orignal score is not modified.

`score-filter-length(`*score*`, `*cutoff*`)`
Remove notes that extend beyond the *cutoff* time. This is similar to `score-select`, but the here, events are removed when their nominal ending time (start time plus duration) exceeds the *cutoff*, whereas the `:to-time` parameter is compared to the note's start time. The original score is not modified, and a new score is returned.

`score-repeat(`*score*`, `*n*`)`
Make a sequence of *n* copies of *score*. Each copy is shifted to that it's begin time aligns with the end time of the previous copy, as in `score-append`. The original score is not modified, and a new score is returned.

`score-stretch-to-length(`*score*`, `*length*`)`
Stretch the score so that the end time of the score is the score's begin time plus *length*. The original score is not modified, and a new score is returned.

`score-filter-overlap(`*score*`)`
Remove overlapping notes (based on the note start time and duration), giving priority to the positional order within the note list (which is also time order). The original score is not modified, and a new score is returned.

`score-print(`*score*`)`
Print a score with one note per line. Returns `nil`.

`score-play(`*score*`)`
Play *score* using `timed-seq` to convert the score to a sound, and `play` to play the sound.

`score-adjacent-events(`*score*`, `*function*`, [from-index: `*i*`] [, to-index: `*j*`,] [from-time: `*x*`] [, to-time: `*y*`])`
Call (*function A B C*), where *A*, *B*, and *C* are consecutive notes in the score. The result replaces *B*. If the result is `nil`, *B* is deleted, and the next call will be (*function A C D*), etc. The first call is to (*function* `nil` *A B*) and the last is to (*function Y Z* `nil`). If there is just one

note in the score, (*function* `nil` *A* `nil`) is called. Function calls are not made if the note is outside of the indicated range. This function allows notes and their parameters to be adjusted according to their immediate context. The original score is not modified, and a new score is returned.

`score-apply(`*score*`,` *function*`, [from-index:` *i*`] [, to-index:` *j*`,] [from-time:` *x*`] [, to-time:` *y*`])`
Replace each note in the score with the result of (*function time dur expression*), where *time*, *dur*, and *expression* are the time, duration, and expression of the note. If a range is indicated, only notes in the range are replaced. The original score is not modified, and a new score is returned.

`score-indexof(`*score*`,` *function*`, [from-index:` *i*`] [, to-index:` *j*`,] [from-time:` *x*`] [, to-time:` *y*`])`
Return the index (position) of the first score event (in range) for which applying *function* using (*function time dur expression*) returns true.

`score-last-indexof(`*score*`,` *function*`, [from-index:` *i*`] [, to-index:` *j*`,] [from-time:` *x*`] [, to-time:` *y*`])`
Return the index (position) of the last score event (in range) for which applying *function* using (*function time dur expression*) returns true.

`score-randomize-start(`*score*`,` *amt*`[ ,from-index:` *i*`] [, to-index:` *j*`,] [from-time:` *x*`] [, to-time:` *y*`])`
Alter the start times of notes by a random amount up to plus or minus *amt*. The original score is not modified, and a new score is returned.

13.4.4. Xmusic and Standard MIDI Files

Nyquist has a general facility to read and write MIDI files. You can even translate to and from a text representation, as described in Chapter 10. It is also useful sometimes to read notes from Standard MIDI Files into Xmusic scores and vice versa. At present, Xmusic only translates notes, ignoring the various controls, program changes, pitch bends, and other messages.

MIDI notes are translated to Xmusic score events as follows:

(*time dur* (`NOTE :chan` *channel* `:pitch` *keynum* `:vel` *velocity*)),

where *channel*, *keynum*, and *velocity* come directly from the MIDI message (channels are numbered starting from zero). Note also that note-off messages are implied by the stretch factor *dur* which is duration in seconds.

`score-read-smf(`*filename*`)`
Read a standard MIDI file from *filename*. Return an Xmusic score, or `nil` if the file could not be opened. The start time is zero, and the end time is the maximum end time of all notes. A very limited interface is offered to extract MIDI program numbers from the file: The global variable `*rslt*` is set to a list of MIDI program numbers for each channel. E.g. if `*rslt*` is `(0 20 77)`, then program for channel 0 is 0, for channel 1 is 20, and for channel 2 is 77. Program changes were not found on other channels. The default program number is 0, so in this example, it is not known whether the program 0 on channel 0 is the result of a real MIDI program change command or just a default value. If more than one program change exists on a channel, the *last* program number is recorded and returned, so this information will only be completely correct when the MIDI file sends single program change per channel before any notes are played. This, however, is a fairly common practice. Note that the list returned as `*rslt*` can be passed to `score-write-smf`, described below.

`score-write-smf(`*score*`,` *filename*`, [`*programs*`])`
Write a standard MIDI file to *filename* with notes in *score*. In this function, *every* event in the score with a `:pitch` attribute, regardless of the ‘‘instrument’’ (or function name), generates a

MIDI note, using the `:chan` attribute for the channel (default 0) and the `:vel` attribute for velocity (default 100). There is no facility (in the current implementation) to issue control changes, but to allow different instruments, MIDI programs may be set in two ways. The simplest is to associate programs with channels using the optional *programs* parameter, which is simply a list of up to 16 MIDI program numbers. Corresponding program change commands are added to the beginning of the MIDI file. If *programs* has less than 16 elements, program change commands are only sent on the first *n* channels. The second way to issue MIDI program changes is to add a `:program` keyword parameter to a note in the score. Typically, the note will have a `:pitch` of `nil` so that no actual MIDI note-on message is generated. If program changes and notes have the same starting times, their relative playback order is undefined, and the note may be cut off by an immediately following program change. Therefore, program changes should occur slightly, e.g. 1 ms, before any notes. *Program numbers and channels are numbered starting at zero, matching the internal MIDI representation. This may be one less than displayed on MIDI hardware, sequencers, etc.*

13.4.5. Workspaces

When working with scores, you may find it necessary to save them in files between work sessions. This is not an issue with functions because they are normally edited in files and loaded from them. In contrast, scores are created as Lisp data, and unless you take care to save them, they will be destroyed when you exit the Nyquist program.

A simple mechanism called a workspace has been created to manage scores (and any other Lisp data, for that matter). A workspace is just a set of lisp global variables. These variables are stored in the file `workspace.lsp`. For simplicity, there is only one workspace, and no backups or versions are maintained, but the user is free to make backups and copies of `workspace.lsp`. To help remember what each variable is for, you can also associate and retrieve a text string with each variable. The following functions manage workspaces.

In addition, when a workspace is loaded, you can request that functions be called. For example, the workspace might store descriptions of a graphical interface. When the workspace is loaded, a function might run to convert saved data into a graphical interface. (This is how sliders are saved by the IDE.)

`add-to-workspace(`*symbol*`)`
: Adds a global variable to the workspace. The *symbol* should be a (quoted) symbol.

`save-workspace()`
: All global variables in the workspace are saved to `workspace.lsp` (in the current directory), overwriting the previous file.

`describe(`*symbol*`, [`*description*`])`
: If *description*, a text string, is present, associate *description* with the variable named by the *symbol*. If *symbol* is not already in the workspace, it is added. If *description* is omitted, the function returns the current description (from a previous call) for *symbol*.

`add-action-to-workspace(`*symbol*`)`
: Requests that the function named by *symbol* be called when the workspace is loaded (if the function is defined).

To restore a workspace, use the command `load "workspace"`. This restores the values of the workspace variables to the values they had when `save-workspace` was last called. It also restores the documentation strings, if set, by `describe`. If you load two or more `workspace.lsp` files, the variables will be merged into a single workspace. The current set of workspace variables are saved in the list `*workspace*`. To clear the workspace, set `*workspace*` to `nil`. This does not delete any

variables, but means that no variables will be saved by `save-workspace` until variables are added again.

Functions to be called are saved in the list `*workspace-actions*`. to clear the functions, set `*workspace-actions*` to `nil`. Restore functions to the list with `add-action-to-workspace`.

13.4.6. Utility Functions

This chapter concludes with details of various utility functions for score manipulation.

`patternp(`*expression*`)`
: Test if *expression* is an Xmusic pattern.

`params-transpose(`*params*`, `*keyword*`, `*amount*`)`
: Add a transposition amount to a score event parameter. The *params* parameter is a list of keyword/value pairs (not preceded by a function name). The *keyword* is the keyword of the value to be altered, and *amount* is a number to be added to the value. If no matching keyword is present in *params*, then *params* is returned. Otherwise, a new parameter list is constructed and returned. The original *params* is not changed.

`params-scale(`*params*`, `*keyword*`, `*amount*`)`
: Scale a score event parameter by some factor. This is like `params-transpose`, only using multiplication. The *params* list is a list of keyword/value pairs, *keyword* is the parameter keyword, and *amount* is the scale factor.

`interpolate(`*x*`, `*x1*`, `*y1*`, `*x2*`, `*y2*`)`
: Linearly interpolate (or extrapolate) between points (*x1*, *y1*) and (*x2*, *y2*) to compute the y value corresponding to *x*.

`intersection(`*a*`, `*b*`)`
: Compute the set intersection of lists *a* and *b*.

`union(`*a*`, `*b*`)`
: Compute the set union of lists *a* and *b*.

`set-difference(`*a*`, `*b*`)`
: Compute the set of all elements that are in *a* but not in *b*.

`subsetp(`*a*`, `*b*`)`
: Returns true iff *a* is a subset of *b*, that is, each element of *a* is a member of *b*.

14. Nyquist Libraries

Nyquist is always growing with new functions. Functions that are most fundamental are added to the core language. These functions are automatically loaded when you start Nyquist, and they are documented in the preceding chapters. Other functions seem less central and are implemented as lisp files that you can load. These are called library functions, and they are described here.

To use a library function, you must first load the library, e.g. `(load "pianosyn")` loads the piano synthesis library. The libraries are all located in the `lib` directory, and you should therefore include this directory on your `XLISPPATH` variable. (See Section 1.) Each library is documented in one of the following sections. When you load the library described by the section, all functions documented in that section become available.

14.1. Piano Synthesizer

The piano synthesizer (library name is `pianosyn.lsp`) generates realistic piano tones using a multiple wavetable implementation by Zheng (Geoffrey) Hua and Jim Beauchamp, University of Illinois. Please see the notice about acknowledgements that prints when you load the file. Further informations and example code can be found in `demos/piano.htm`. There are several useful functions in this library:

`piano-note(`*duration*`,` *step*`,` *dynamic*`)`
: Synthesizes a piano tone. *Duration* is the duration to the point of key release, after which there is a rapid decay. *Step* is the pitch in half steps, and *dynamic* is approximately equivalent to a MIDI key velocity parameter. Use a value near 100 for a loud sound and near 10 for a soft sound.

`piano-note-2(`*step*`,` *dynamic*`)`
: Similar to `piano-note` except the duration is nominally 1.0.

`piano-midi(`*midi-file-name*`)`
: Use the piano synthesizer to play a MIDI file. The file name (a string) is given by *midi-file-name*.

`piano-midi2file(`*midi-file-name*`,` *sound-file-name*`)`
: Use the piano synthesizer to play a MIDI file. The MIDI file is given by *midi-file-name* and the (monophonic) result is written to the file named *sound-file-name*.

14.2. Dymanics Compression

To use these functions, load the file `compress.lsp`. This library implements a compressor originally intended for noisy speech audio, but usable in a variety of situations. There are actually two compressors that can be used in series. The first, `compress`, is a fairly standard one: it detects signal level with an RMS detector and uses table-lookup to determine how much gain to place on the original signal at that point. One bit of cleverness here is that the RMS envelope is ‘‘followed’’ or enveloped using `snd-follow`, which does look-ahead to anticipate peaks before they happen.

The other interesting feature is `compress-map`, which builds a map in terms of compression and expansion. For speech, the recommended procedure is to figure out the noise floor on the signal you are compressing (for example, look at the signal where the speaker is not talking). Use a compression map that leaves the noise alone and boosts signals that are well above the noise floor. Alas, the `compress-map` function is not written in these terms, so some head-scratching is involved, but the results are quite good.

The second compressor is called `agc`, and it implements automatic gain control that keeps peaks at or

below 1.0. By combining `compress` and `agc`, you can process poorly recorded speech for playback on low-quality speakers in noisy environments. The `compress` function modulates the short-term gain to to minimize the total dynamic range, keeping the speech at a generally loud level, and the `agc` function rides the long-term gain to set the overall level without clipping.

`compress-map(`*compress-ratio*`,` *compress-threshold*`,` *expand-ratio*`,` *expand-ratio*`[ ,limit:` *limit*`] [, transition:` *transition*`])`
: Construct a map for the compress function. The map consists of two parts: a compression part and an expansion part. The intended use is to compress everything above compress-threshold by compress-ratio, and to downward expand everything below expand-ratio by expand-ratio. Thresholds are in dB and ratios are dB-per-dB. 0dB corresponds to a peak amplitude of 1.0 or rms amplitude of 0.7 If the input goes above 0dB, the output can optionally be limited by setting `:limit` (a keyword parameter) to `T`. This effectively changes the compression ratio to infinity at 0dB. If `:limit` is `nil` (the default), then the compression-ratio continues to apply above 0dB.

Another keyword parameter, `:transition`, sets the amount below the thresholds (in dB) that a smooth transition starts. The default is 0, meaning that there is no smooth transition. The smooth transition is a 2nd-order polynomial that matches the slopes of the straight-line compression curve and interpolates between them.

It is assumed that expand-threshold <= compress-threshold <= 0 The gain is unity at 0dB so if compression-ratio > 1, then gain will be greater than unity below 0dB.

The result returned by this function is a sound for use in the `shape` function. The sound maps input dB to gain. Time 1.0 corresponds to 0dB, time 0.0 corresponds to -100 dB, and time 2.0 corresponds to +100dB, so this is a 100hz ‘‘sample rate’’ sound. The sound gives gain in dB.

`db-average(`*input*`)`
: Compute the average amplitude of *input* in dB.

`compress(`*input*`,` *map*`,` *rise-time*`,` *fall-time*`, [`*lookahead*`])`
: Compress *input* using *map*, a compression curve probably generated by `compress-map` (see above). Adjustments in gain have the given *rise-time* and *fall-time*. Lookahead tells how far ahead to look at the signal, and is *rise-time* by default.

`agc(`*input*`,` *range*`,` *rise-time*`,` *fall-time*`[ ,`*lookahead*`])`
: An automatic gain control applied to *input*. The maximum gain in dB is *range*. Peaks are attenuated to 1.0, and gain is controlled with the given *rise-time* and *fall-time*. The look-ahead time default is *rise-time*.

14.3. Clipping Softener

This library, in `soften.lsp`, was written to improve the quality of poorly recorded speech. In recordings of speech, extreme clipping generates harsh high frequency noise. This can sound particulary bad on small speakers that will emphasize high frequencies. This problem can be ameliorated by low-pass filtering regions where clipping occurs. The effect is to dull the harsh clipping. Intelligibility is not affected by much, and the result can be much more pleasant on the ears. Clipping is detected simply by looking for large signal values. Assuming 8-bit recording, this level is set to 126/127.

The function works by cross-fading between the normal signal and a filtered signal as opposed to changing filter coefficients.

`soften-clipping(`*snd*`,` *cutoff*`)`
: Filter the loud regions of a signal where clipping is likely to have generated additional high frequencies. The input signal is *snd* and *cutoff* is the filter cutoff frequency (4 kHz is recommended for speech).

14.4. Graphical Equalizer

There's nothing really ''graphical'' about this library (`grapheq.lsp`), but this is a common term for multi-band equalizers. This implementation uses Nyquist's `eq-band` function to split the incoming signal into different frequency bands. Bands are spaced geometrically, e.g. each band could be one octave, meaning that each successive band has twice the bandwidth. An interesting possibility is using computed control functions to make the equalization change over time.

`nband-range(`*input*`, `*gains*`, `*lowf*`, `*highf*`)`
: A graphical equalizer applied to *input* (a `SOUND`). The gain controls and number of bands is given by *gains*, an ARRAY of `SOUND`s (in other words, a Nyquist multichannel `SOUND`). Any sound in the array may be replaced by a `FLONUM`. The bands are geometrically equally spaced from the lowest frequency *lowf* to the highest frequency *highf* (both are `FLONUM`s).

`nband(`*input*`, `*gains*`)`
: A graphical equalizer, identical to `nband-range` with a range of 20 to 20,000 Hz.

14.5. Sound Reversal

The `reverse.lsp` library implements functions to play sounds in reverse.

`s-reverse(`*snd*`)`
: Reverses *snd* (a `SOUND`). Sound must be shorter than `*max-reverse-samples*`, which is currently initialized to 25 million samples. Reversal allocates about 4 bytes per sample. This function uses XLISP in the inner sample loop, so do not be surprised if it calls the garbage collector a lot and runs slowly. The result starts at the starting time given by the current environment (not necessarily the starting time of *snd*). If *snd* has multiple channels, a multiple channel, reversed sound is returned.

`s-read-reverse(`*filename*`[ ,time-offset: `*offset*`] [, srate: `*sr*`] [, dur: `*dur*`] [, nchans: `*chans*`] [, format: `*format*`] [, mode: `*mode*`] [, bits: `*n*`] [, swap: `*flag*`])`
: This function is identical to `s-read` (see 7.5), except it reads the indicated samples in reverse. Like `s-reverse` (see above), it uses XLISP in the inner loop, so it is slow. Unlike `s-reverse`, `s-read-reverse` uses a fixed amount of memory that is independent of how many samples are computed. Multiple channels are handled.

14.6. Time Delay Functions

The `time-delay-fns.lsp` library implements chorus, phaser, and flange effects.

`phaser(`*snd*`)`
: A phaser effect applied to *snd* (a `SOUND`). There are no parameters, but feel free to modify the source code of this one-liner.

`flange(`*snd*`)`
: A flange effect applied to *snd*. To vary the rate and other parameters, see the source code.

`stereo-chorus(`*snd*`)`
: A chorus effect applied to *snd*, a `SOUND` (monophonic). The output is a stereo sound. All parameters are built-in, but see the simple source code to make modifications.

`chorus(`*snd*`, `*maxdepth*`, `*depth*`, `*rate*`, `*saturation*`)`
: A chorus effect applied to *snd*. All parameters may be arrays as usual. The *maxdepth* is a `FLONUM` giving twice the maximum value of *depth*, which may be a `FLONUM` or a `SOUND`. The chorus is implemented as a variable delay modulated by a sinusoid running at *rate* Hz (a `FLONUM`). The sinusoid is scaled by *depth* and offset by *maxdepth*/2. The delayed signal is mixed

with the original, and *saturation* gives the fraction of the delayed signal (from 0 to 1) in the mix. A reasonable choice of parameter values is *maxdepth* = 0.05, *depth* = 0.025, *rate* = 0.5, and *saturation* = 0.5.

14.7. Multiple Band Effects

The `bandfx.lsp` library implements several effects based on multiple frequency bands. The idea is to separate a signal into different frequency bands, apply a slightly different effect to each band, and sum the effected bands back together to form the result. This file includes its own set of examples. After loading the file, try `f2()`, `f3()`, `f4()`, and `f5()` to hear them.

There is much room for expansion and experimentation with this library. Other effects might include distortion in certain bands (for example, there are commercial effects that add distortion to low frequencies to enhance the sound of the bass), separating bands into different channels for stereo or multi-channel effects, adding frequency-dependent reverb, and performing dynamic compression, limiting, or noise gate functions on each band. There are also opportunities for cross-synthesis: using the content of bands extracted from one signal to modify the bands of another. The simplest of these would be to apply amplitude envelopes of one sound to another. Please contact us (dannenberg@cs.cmu.edu) if you are interested in working on this library.

`apply-banded-delay(` *s*, *lowp*, *highp*, *num-bands*, *lowd*, *highd*, *fb*, *wet*`)`
Separates input `SOUND` *s* into `FIXNUM` *num-bands* bands from a low frequency of *lowp* to a high frequency of *highp* (these are `FLONUMS` that specify steps, not Hz), and applies a delay to each band. The delay for the lowest band is given by the `FLONUM` *lowd* (in seconds) and the delay for the highest band is given by the `FLONUM` *highd*. The delays for other bands are linearly interpolated between these values. Each delay has feedback gain controlled by `FLONUM` *fb*. The delayed bands are scaled by `FLONUM` *wet*, and the original sound is scaled by 1 - *wet*. All are summed to form the result, a `SOUND`.

`apply-banded-bass-boost(`*s*, *lowp*, *highp*, *num-bands*, *num-boost*, *gain*`)`
Applies a boost to low frequencies. Separates input `SOUND` *s* into `FIXNUM` *num-bands* bands from a low frequency of *lowp* to a high frequency of *highp* (these are `FLONUMS` that specify steps, not Hz), and scales the lowest *num-boost* (a `FIXNUM`) bands by *gain*, a `FLONUM`. The bands are summed to form the result, a `SOUND`.

`apply-banded-treble-boost(`*s*, *lowp*, *highp*, *num-bands*, *num-boost*, *gain*`)`
Applies a boost to high frequencies. Separates input `SOUND` *s* into `FIXNUM` *num-bands* bands from a low frequency of *lowp* to a high frequency of *highp* (these are `FLONUMS` that specify steps, not Hz), and scales the highest *num-boost* (a `FIXNUM`) bands by *gain*, a `FLONUM`. The bands are summed to form the result, a `SOUND`.

14.8. Granular Synthesis

Some granular synthesis functions are implemented in the `gran.lsp` library file. There are many variations and control schemes one could adopt for granular synthesis, so it is impossible to create a single universal granular synthesis function. One of the advantages of Nyquist is the integration of control and synthesis functions, and users are encouraged to build their own granular synthesis functions incorporating their own control schemes. The `gran.lsp` file includes many comments and is intended to be a useful starting point. Another possibility is to construct a score with an event for each grain. Estimate a few hundred bytes per score event (obviously, size depends on the number of parameters) and avoid using all of your computer's memory.

`sf-granulate(`*filename*, *grain-dur*, *grain-dev*, *ioi*, *ioi-dev*, *pitch-dev*, [*file-start*] [,

file-end])
Granular synthesis using a sound file named *filename* as the source for grains. Grains are extracted from a sound file named by *filename* by stepping through the file in equal increments. Each grain duration is the sum of *grain-dur* and a random number from 0 to *grain-dev*. Grains are then multiplied by a raised cosine smoothing window and resampled at a ratio between 1.0 and *pitch-dev*. If *pitch-dev* is greater than one, grains are stretched and the pitch (if any) goes down. If *pitch-dev* is less than one, grains are shortened and the pitch goes up. Grains are then output with an inter-onset interval between successive grains (which may overlap) determined by the sum of *ioi* and a random number from 0 to *ioi-dev*. The duration of the resulting sound is determined by the stretch factor (not by the sound file). The number of grains is the total sound duration (determined by the stretch factor) divided by the mean inter-onset interval, which is *ioi + ioi-dev* * 0.5. The grains are taken from equally-spaced starting points in *filename*, and depending on grain size and number, the grains may or may not overlap. The output duration will simply be the sum of the inter-onset intervals and the duration of the last grain. If *ioi-dev* is non-zero, the output duration will vary, but the expected value of the duration is the stretch factor. To achieve a rich granular synthesis effect, it is often a good idea to sum four or more copies of `sf-granulate` together. (See the `gran-test` function in `gran.lsp`.)

14.9. MIDI Utilities

The `midishow.lsp` library has functions that can print the contents fo MIDI files. This intended as a debugging aid.

`midi-show-file(`*file-name*`)`
Print the contents of a MIDI file to the console.

`midi-show(`*the-seq*`[ ,`*out-file*`])`
Print the contents of the sequence *the-seq* to the file *out-file* (whose default value is the console.)

14.10. Reverberation

The `reverb.lsp` library implements artificial reverberation.

`reverb(`*snd*`, `*time*`)`
Artificial reverberation applied to *snd* with a decay time of *time*.

14.11. DTMF Encoding

The `dtmf.lsp` library implements DTMF encoding. DTMF is the ‘‘touch tone’’ code used by telephones.

`dtmf-tone(`*key*`, `*len*`, `*space*`)`
Generate a single DTMF tone. The *key* parameter is either a digit (a `FIXNUM` from 0 through 9) or the atom `STAR` or `POUND`. The duration of the done is given by *len* (a `FLONUM`) and the tone is followed by silence of duration *space* (a `FLONUM`).

`speed-dial(`*thelist*`)`
Generates a sequence of DTMF tones using the keys in *thelist* (a `LIST` of keys as described above under `dtmf-tone`). The duration of each tone is 0.2 seconds, and the space between tones is 0.1 second. Use `stretch` to change the ‘‘dialing’’ speed.

14.12. Dolby Surround(R), Stereo and Spatialization Effects

The `spatial.lsp` library implements various functions for stereo manipulation and spatialization. It also includes some functions for Dolby Pro-Logic panning, which encodes left, right, center, and surround channels into stereo. The stereo signal can then be played through a Dolby decoder to drive a surround speaker array. This library has a somewhat simplified encoder, so you should certainly test the output. Consider using a high-end encoder for critical work. There are a number of functions in `spatial.lsp` for testing. See the source code for comments about these.

`stereoize(`*snd*`)`
: Convert a mono sound, *snd*, to stereo. Four bands of equalization and some delay are used to create a stereo effect.

`widen(`*snd*`, `*amt*`)`
: Artificially widen the stereo field in *snd*, a two-channel sound. The amount of widening is *amt*, which varies from 0 (*snd* is unchanged) to 1 (maximum widening). The *amt* can be a `SOUND` or a number.

`span(`*snd*`, `*amt*`)`
: Pan the virtual center channel of a stereo sound, *snd*, by *amt*, where 0 pans all the way to the left, while 1 pans all the way to the right. The *amt* can be a `SOUND` or a number.

`swapchannels(`*snd*`)`
: Swap left and right channels in *snd*, a stereo sound.

`prologic(`*l*`, `*c*`, `*r*`, `*s*`)`
: Encode four monaural `SOUND`s representing the front-left, front-center, front-right, and rear channels, respectively. The return value is a stereo sound, which is a Dolby-encoded mix of the four input sounds.

`pl-left(`*snd*`)`
: Produce a Dolby-encoded (stereo) signal with *snd*, a `SOUND`, encoded as the front left channel.

`pl-center(`*snd*`)`
: Produce a Dolby-encoded (stereo) signal with *snd*, a `SOUND`, encoded as the front center channel.

`pl-right(`*snd*`)`
: Produce a Dolby-encoded (stereo) signal with *snd*, a `SOUND`, encoded as the front right channel.

`pl-rear(`*snd*`)`
: Produce a Dolby-encoded (stereo) signal with *snd*, a `SOUND`, encoded as the rear, or surround, channel.

`pl-pan2d(`*snd*`, `*x*`, `*y*`)`
: Comparable to Nyquist's existing pan function, `pl-pan2d` provides not only left-to-right panning, but front-to-back panning as well. The function accepts three parameters: *snd* is the (monophonic) input `SOUND`, *x* is a left-to-right position, and *y* is a front-to-back position. Both position parameters may be numbers or `SOUND`s. An *x* value of 0 means left, and 1 means right. Intermediate values map linearly between these extremes. Similarly, a *y* value of 0 causes the sound to play entirely through the front speakers(s), while 1 causes it to play entirely through the rear. Intermediate values map linearly. Note that, although there are usually two rear speakers in Pro-Logic systems, they are both driven by the same signal. Therefore any sound that is panned totally to the rear will be played over both rear speakers. For example, it is not possible to play a sound exclusively through the rear left speaker.

`pl-position(`*snd*`, `*x*`, `*y*`, `*config*`)`
: The position function builds upon speaker panning to allow more abstract placement of sounds. Like `pl-pan2d`, it accepts a (monaural) input sound as well as left-to-right (*x*) and front-to-back (*y*) coordinates, which may be `FLONUM`s or `SOUND`s. A fourth parameter *config* specifies the distance from listeners to the speakers (in meters). Current settings assume this to be constant for

all speakers, but this assumption can be changed easily (see comments in the code for more detail). There are several important differences between `pl-position` and `pl-pan2d`. First, `pl-position` uses a Cartesian coordinate system that allows x and y coordinates outside of the range (0, 1). This model assumes a listener position of (0,0). Each speaker has a predefined position as well. The input sound's position, relative to the listener, is given by the vector (*x*,*y*).

`pl-doppler(`*snd*`, `*r*`)`

Pitch-shift moving sounds according to the equation: *fr = f0*((*c*+*vr*)/*c*), where *fr* is the output frequency, *f0* is the emitted (source) frequency, *c* is the speed of sound (assumed to be 344.31 m/s), and *vr* is the speed at which the emitter approaches the receiver. (*vr* is the first derivative of parameter *r*, the distance from the listener in meters.

14.13. Drum Machine

The drum machine software in `demos/plight` deserves further explanation. to use the software, load the code by evaluating:

```
load "../demos/plight/drum.lsp"
exec load-props-file(strcat(*plight-drum-path*,
                            "beats.props"))
exec create-drum-patches()
exec create-patterns()
```

Drum sounds and patterns are specified in the `beats.props` file (or whatever name you give to `load-props-file`). This file contains two types of specifications. First, there are sound file specifications. Sound files are located by a line of the form:

```
set sound-directory = "kit/"
```

This gives the name of the sound file directory, relative to the `beats.props` file. Then, for each sound file, there should be a line of the form:

```
track.2.5 = big-tom-5.wav
```

This says that on track 2, a velocity value of 5 means to play the sound file `big-tom-5.wav`. (Tracks and velocity values are described below.) The `beats.props` file contains specifications for all the sound files in `demos/plight/kit` using 8 tracks. If you make your own specifications file, tracks should be numbered consecutively from 1, and velocities should be in the range of 1 to 9.

The second set of specifications is of beat patterns. A beat pattern is given by a line in the following form:

```
beats.5 = 2--32--43-4-5---
```

The number after `beats` is just a pattern number. Each pattern is given a unique number. After the equal sign, the digits and dashes are velocity values where a dash means ‘‘no sound.’’ Beat patterns should be numbered consecutively from 1.

Once data is loaded, there are several functions to access drum patterns and create drum sounds (described below). The `demos/plight/drums.lsp` file contains an example function `plight-drum-example` to play some drums. There is also the file `demos/plight/beats.props` to serve as an example of how to specify sound files and beat patterns.

`drum(`*tracknum*`, `*patternnum*`, `*bpm*`)`

Create a sound by playing drums sounds associated with track *tracknum* (a FIXNUM) using pattern *patternnum*. The tempo is given by *bpm* in beats per minute. Normally patterns are a

sequence of sixteenth notes, so the tempo is in *sixteenth notes per minute*. For example, if *patternnum* is 10, then use the pattern specified for `beats.10`. If the third character of this pattern is 3 and *tracknum* is 5, then on the third beat, play the soundfile assigned to `track.5.3`. This function returns a `SOUND`.

`drum-loop(`*snd, duration, numtimes*`)`
Repeat the sound given by *snd numtimes* times. The repetitions occur at a time offset of *duration*, regardless of the actual duration of *snd*. A `SOUND` is returned.

`length-of-beat(`*bpm*`)`
Given a tempo of *bpm*, return the duration of the beat in seconds. Note that this software has no real notion of beat. A ‘‘beat’’ is just the duration of each character in the beat pattern strings. This function returns a `FLONUM`.

14.14. Minimoog-inspired Synthesis

The `moog.lsp` library gives the Nyquist user easy access to ‘‘classic’’ synthesizer sounds through an emulation of the Minimoog Synthesizer. Unlike modular Moogs that were very large, the Minimoog was the first successful and commonly used portable synthesizer. The trademark filter attack was unique and easily recognizable. The goal of this Nyquist instrument is not only to provide the user with default sounds, but also to give control over many of the ‘‘knobs’’ found on the Minimoog. In this implementation, these parameters are controlled using keywords. The input to the `moog` instrument is a user-defined sequence of notes, durations, and articulations that simulate notes played on a keyboard. These are translated into control voltages that drive multiple oscillators, similar to the Voltage Controlled Oscillator or VCO found in the original analog Moog.

The basic functionality of the Minimoog has been implemented, including the often-used "glide". The glide feature essentially low-pass filters the control voltage sequence in order to create sweeps between notes. Figure 21 is a simplified schematic of the data flow in the Moog. The control lines have been omitted.

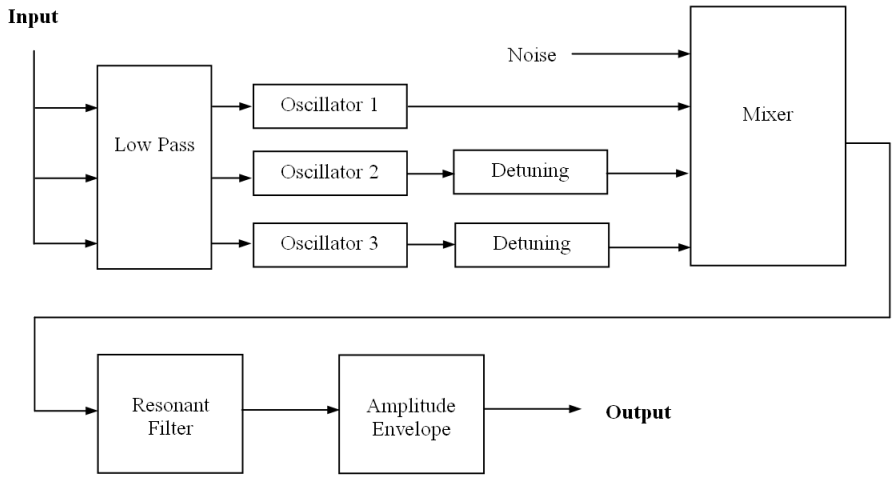

Figure 21: System diagram for Minimoog emulator.

The most recognizable feature of the Minimoog is its resonant filter, a Four-Pole Ladder Filter invented by Robert Moog. It is simply implemented in a circuit with four transistors and provides an outstanding

24 dB/octave rolloff. It is modeled here using the built-in Nyquist resonant filter. One of the Moog filter features is a constant Q, or center frequency to bandwidth ratio. This is implemented and the user can control the Q.

The user can control many parameters using keywords. Their default values, acceptable ranges, and descriptions are shown below. The defaults were obtained by experimenting with the official Minimoog software synthesizer by Arturia.

14.14.1. Oscillator Parameters

`range-osc1` (2)
`range-osc2` (1)
`range-osc3` (3)
These parameters control the octave of each oscillator. A value of 1 corresponds to the octave indicated by the input note. A value of 3 is two octaves above the fundamental. The allowable range is 1 to 7.

`detun2` (-.035861)
`detun3` (.0768)
Detuning of two oscillators adds depth to the sound. A value of 1 corresponds to an increase of a single semitone and a -1 corresponds to a decrease in a semitone. The range is -1 to 1.

`shape-osc1` (`*saw-table*`)
`shape-osc2` (`*saw-table*`)
`shape-osc3` (`*saw-table*`)
Oscilators can use any wave shape. The default sawtooth waveform is a built-in Nyquist variable. Other waveforms can be defined by the user.

`volume-osc1` (1)
`volume-osc2` (1)
`volume-osc3` (1)
These parameters control the relative volume of each oscillator. The range is any `FLONUM` greater than or equal to zero.

14.14.2. Noise Parameters

`noiselevel` (.05)
This parameter controls the relative volume of the noise source. The range is any `FLONUM` greater than or equal to zero.

14.14.3. Filter Parameters

`filter-cutoff` (768)
The cutoff frequency of the filter in given in Hz. The range is zero to 20,000 Hz.

`Q` (2)
Q is the ratio of center frequency to bandwidth. It is held constant by making the bandwidth a function of frequency. The range is any `FLONUM` greater than zero.

`contour` (.65)

Contour controls the range of the transient frequency sweep from a high to low cutoff frequency when a note is played. The high frequency is proportional to contour. A contour of 0 removes this sweep. The range is 0 to 1.

`filter-attack` (.0001)
Filter attack controls the attack time of the filter, i.e. the time to reach the high cutoff frequency. The range is any `FLONUM` greater than zero (seconds).

`filter-decay` (.5)
Filter decay controls the decay time of the filter, i.e. the time of the sweep from the high to low cutoff frequency. The range is any `FLONUM` greater than zero (seconds).

`filter-sustain` (.8)
Filter sustain controls the percentage of the filter cutoff frequency that the filter settles on following the sweep. The range is 0 to 1.

14.14.4. Amplitude Parameters

`amp-attack` (.01)
This parameter controls the amplitude envelope attack time, i.e. the time to reach maximum amplitude. The range is any `FLONUM` greater than zero (seconds).

`amp-decay` (1)
This parameter controls the amplitude envelope decay time, i.e. the time between the maximum and sustain volumes. The range is any `FLONUM` greater than zero (seconds).

`amp-sustain` (1)
This parameter controls the amplitude envelope sustain volume, a fraction of the maximum. The range is 0 to 1.

`amp-release` (0)
This parameter controls the amplitude envelope release time, i.e. the time it takes between the sustain volume and 0 once the note ends. The duration controls the overall length of the sound. The range of `amp-release` is any `FLONUM` greater than zero (seconds).

14.14.5. Other Parameters

`glide` (0)
Glide controls the low-pass filter on the control voltages. This models the glide knob on a Minimoog. A higher value corresponds to a lower cutoff frequency and hence a longer "glide" between notes. A value of 0 corresponds to no glide. The range is zero to 10.

14.14.6. Input Format

A single note or a series of notes can be input to the Moog instrument by defining a list with the following format:

(list (list *frequency duration articulation*) ...)

where *frequency* is a `FLONUM` in steps, *duration* is the duration of each note in seconds (regardless of the release time of the amplifier), and *articulation* is a percentage of the duration that a sound will be played,

representing the amount of time that a key is pressed. The filter and amplitude envelopes are only triggered if a note is played when the articulation of the previous note is less than 1, or a key is not down at the same time. This Moog instrument is a monophonic instrument, so only one note can sound at a time. The release section of the amplifier is triggered when the articulation is less than 1 at the time (*duration* * *articulation*).

14.14.7. Sample Code/Sounds

Sound 1 (default parameters):

```
set s = {{24 .5 .99} {26 .5 .99} {28 .5 .99}
         {29 .5 .99} {31 2 1}}
play moog(s)
```

Sound 2 (articulation, with amplitude release):

```
set s = {{24 .5 .5} {26 .5 1} {28 .5 .25} {29 .5 1} {31 1 .8}}
play moog(s, amp-release: .2)
```

Sound 3 (glide):

```
set s = {{24 .5 .5} {38 .5 1} {40 .5 .25}
         {53 .5 1} {55 2 1} {31 2 .8} {36 2 .8}}
play moog(s, amp-release: .2, glide: .5)
```

Sound 4 (keyword parameters): Filter attack and decay are purposely longer than notes being played with articulation equal to 1.

```
set s = {{20 .5 1} {27 .5 1} {26 .5 1} {21 .5 1}
         {20 .5 1} {27 .5 1} {26 .5 1} {21 .5 1}}
play moog(s, shape-osc1: *tri-table*, shape-osc2: *tri-table*,
             filter-attack: 2, filter-decay: 2,
             filter-cutoff: 300, contour: .8, glide: .2, Q: 8)
```

Sound 5: This example illustrates the ability to completely define a new synthesizer with different parameters creating a drastically different sound. Sine waves are used for wavetables. There is a high value for glide.

```
define function my-moog(freq)
  return moog(freq,
    range-osc1: 3, range-osc2: 2, range-osc3: 4,
    detun2: -.043155, detun3: .015016,
    noiselevel: 0,
    filter-cutoff: 400, Q: .1, contour: .0000001,
    filter-attack: 0, filter-decay: .01, filter-sustain: 1,
    shape-osc1: *sine-table*, shape-osc2: *sine-table*,
    shape-osc3: *sine-table*, volume-osc1: 1, volume-osc2: 1,
    volume-osc3: .1, amp-attack: .1, amp-decay: 0,
    amp-sustain: 1, amp-release: .3, glide: 2)

set s = {{80 .4 .75} {28 .2 1} {70 .5 1} {38 1 .5}}
play my-moog(s)
```

Sound 6: This example has another variation on the default parameters.

```
set s = {{24 .5 .99} {26 .5 .99} {28 .5 .99}
         {29 .5 .99} {31 2 1}}
play moog(s, shape-osc1: *tri-table*, shape-osc2: *tri-table*,
             filter-attack: .5, contour: .5)
```

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.)

I.1. Translating Descriptions to C Code

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.

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!

Note: On the Macintosh, just run Nyquist out of the `runtime` directory and then use the `Load` menu command to load `init.lsp` from the `trnsrc` directory. This will load the translation code and change Nyquist's current directory to `trnsrc` so that commands like `(translate "prod")` will work.

I.2. Rebuilding Nyquist

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)`.

I.3. Accessing the New Function

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.4. 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.5. 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.6. 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.7. 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.8. 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
Open Sound Control and Nyquist

Open Sound Control (OSC) is a simple protocol for communicating music control parameters between software applications and across networks. For more information, see `http://www.cnmat.berkeley.edu/OpenSoundControl/`. The Nyquist implementation of Open Sound Control is simple: an array of floats can be set by OSC messages and read by Nyquist functions. That is about all there is to it.

Note: Open Sound Control must be enabled by calling `osc-enable(t)`. If this fails under Windows, see the installation instructions regarding `SystemRoot`.

To control something in (near) real-time, you need to access a slider value as if it a signal, or more properly, a Nyquist `SOUND` type. The function `snd-slider`, described in Section 7.6.1, takes a slider number and returns a `SOUND` type representing the current value of the slider. To fully understand this function, you need to know something about how Nyquist is actually computing sounds.

Sounds are normally computed on demand. So the result returned by `snd-slider` does not immediately compute any samples. Samples are only computed when something tries to use this signal. At that time, the slider value is read. Normally, if the slider is used to control a sound, you will hear changes in the sound pretty soon after the slider value changes. However, one thing that can interfere with this is that `SOUND` samples are computed in blocks of about 1000 samples. When the slider value is read, the same value is used to fill a block of 1000 samples, so even if the sample rate is 44,100 Hz, the effective slider sample rate is 44,100/1000, or 44.1 Hz. If you give the slider a very low sample rate, say 1000, then slider value changes will only be noticed by Nyquist approximately once per second. For this reason, you should normally use the audio sample rate (typically 44,100 Hz) for the rate of the `snd-slider` output `SOUND`. (Yes, this is terribly wasteful to represent each slider value with 1000 samples, but Nyquist was not designed for low-latency computation, and this is an expedient work-around.)

In addition to reading sliders as continually changing `SOUND`s, you can get the slider value as a Lisp `FLONUM` (a floating point number) using `get-slider-value`, described in Section 7.6.1. This might be useful if you are computing a sequence of many notes (or other sound events) and want to apply the current slider value to the whole note or sound event.

Note that if you store the value returned by `snd-slider` in a variable, you will capture the history of the slider changes. This will take a lot of memory, so be careful.

Suppose you write a simple expression such as `(hzosc (mult 1000 (snd-slider 0 ...)))` to control an oscillator frequency with a slider. How long does this sound last? The duration of `hzosc` is the duration of the frequency control, so what is the duration of a slider? To avoid infinitely long signals, you must specify a duration as one of the parameters of `snd-slider`.

You might be thinking, what if I just want to tell the slider when to stop? At present, you cannot do that, but in the future there should be a function that stops when its input goes to zero. Then, moving a slider to zero could end the signal (and if you multiplied a complex sound by one of these ending functions, everything in the sound would end and be garbage collected).

Another thing you might want to do with interactive control is start some sound. The `trigger`

function computes an instance of a behavior each time an input `SOUND` goes from zero to greater-than-zero. This could be used, for example, to create a sequence of notes.

The `snd-slider` function has some parameters that may be unfamiliar. The second parameter, *t0*, is the starting time of the sound. This should normally be `local-to-global(0)`, an expression that computes the instantiation time of the current expression. This will often be zero, but if you call `snd-slider` from inside a `seq` or `seq-rep`, the starting time may not be zero.

The *srate* parameter is the sample rate to return. This should normally be the audio sample rate you are working with, which is typically `*default-sound-srate*`.

II.1. Sending Open Sound Control Messages

A variety of programs support OSC. The only OSC message interpreted by Nyquist has an address of `/slider`, and two parameters: an integer slider number and a float value, nominally from 0.0 to 1.0.

Two small programs are included in the Nyquist distribution for sending OSC messages. (Both can be found in the same directory as the nyquist executable.) The first one, `osc-test-client` sends a sequence of messages that just cause slider 0 to ramp slowly up and down. If you run this on a command line, you can use "?" or "h" to get help information. There is an interactive mode that lets you send each OSC message by typing RETURN.

II.2. The ser-to-osc Program

The second program is `ser-to-osc`, a program that reads serial input (for example from a PIC-based microcontroller) and sends OSC messages. Run this command-line program from a shell (a terminal window under OS X or Linux; use the CMD program under Windows). You must name the serial input device on the command line, e.g. under OS X, you might run:

```
./ser-to-osc /dev/tty.usbserial-0000103D
```

(Note that the program name is preceded by ‘‘`./`". This tells the shell exactly where to find the executable program in case the current directory is not on the search path for executable programs.) Under Windows, you might run:

```
ser-to-osc com4
```

(Note that you do not type ‘‘`./`’’ in front of a windows program.)

To use `ser-to-osc`, you will have to find the serial device. On the Macintosh and Linux, try the following:

```
ls /dev/*usb*
```

This will list all serial devices with ‘‘usb’’ in their names. Probably, one will be a name similar to `/dev/tty.usbserial-0000103D`. The `ser-to-osc` program will echo data that it receives, so you should know if things are working correctly.

Under Windows, open Control Panel from the Start menu, and open the System control panel. Select the Hardware tab and click the Device Manager button. Look in the device list under Ports (COM & LPT). When you plug in your serial or USB device, you should see a new entry appear, e.g. `COM4`. This is the device name you need.

The format for the serial input is: any non-whitespace character(s), a slider number, a slider value, and

a newline (control-j or ASCII 0x0A). These fields need to be separated by tabs or spaces. An optional carriage return (control-m or ASCII 0x0D) preceding the ASCII 0x0A is ignored. The slider number should be in decimal, and theh slider value is a decimal number from 0 to 255. This is scaled to the range 0.0 to 1.0 (so an input of 255 translates to 1.0).

There is a simple test program in `demos/osc-test.lsp` you can run to try out control with Open Sound Control. There are two examples in that file. One uses `snd-slider` to control the frequency of an oscillator. The other uses `get-slider-value` to control the pitch of grains in a granular synthesis process.

Appendix III
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.

Alternatively, the command line may specify a command file from which to read file names. The command file name should be preceded by "@", for example:

```
intgen @sndfns.cl
```

reads sndfns.cl to get the command-line input. Only one level of indirection is allowed.

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`.

III.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.

III.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`.

III.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 *`.

III.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.*

III.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) */
```

III.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 IV
XLISP: An Object-oriented Lisp

Version 2.0

February 6, 1988

by
David Michael Betz
127 Taylor Road
Peterborough, NH 03458

IV.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.

Implementations of XLISP run on virtually every operating system. XLISP 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.

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.

IV.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.

IV.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.

IV.4. Special Characters

When XLISP is running from a console, some control characters invoke operations:

- Backspace and Delete characters erase the previous character on the input line (if any).
- Control-U erases the entire input line.
- Control-C executes the TOP-LEVEL function.
- Control-G executes the CLEAN-UP function.
- Control-P executes the CONTINUE function.
- Control-B stops execution and enters the break command loop. Execution can be continued by typing Control-P or (CONTINUE).
- Control-E turns on character echoing (Linux and Mac OS X only).
- Control-F turns off character echoing (Linux and Mac OS X only).
- Control-T evaluates the INFO function.

IV.5. 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.

IV.6. 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)

IV.7. 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.

IV.8. 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

IV.9. 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*>)

IV.10. 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

IV.11. 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.

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.

Within the body of a method, a message can be sent to the current object by calling the `(send self ...)`. The method lookup starts with the object's class regardless of the class containing the current method.

Sometimes it is desirable to invoke a general method in a superclass even when it is overridden by a more specific method in a subclass. This can be accomplished by calling `send-super`, which begins the method lookup in the superclass of the class defining the current method rather than in the class of the current object.

The `send-super` function takes a selector as its first argument (which must be a symbol) and the message arguments as its remaining arguments. Notice that `send-super` can only be sent from within a method, and the target of the message is always the current object (`self`). `(send-super ...)` is similar to `(send self ...)` except that method lookup begins in the superclass of the class containing the current method rather than the class of the current object.

IV.12. 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

`:isa(:isa)` *class* — test if object inherits from class
returns — `t` if object is an instance of *class* or a subclass of *class*, otherwise `nil`

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

IV.13. 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.

IV.14. 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.

IV.15. 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.

IV.16. 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

IV.17. 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)

IV.18. 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`

IV.19. 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

IV.20. 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

IV.21. 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

IV.22. 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

(filep *expr*)[8] — is this a file?
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

[8]This is not part of standard XLISP nor is it built-in. Nyquist defines it though.

(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

IV.23. 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

IV.24. 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)

IV.25. 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`)

IV.26. 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 *flag*)[9] — turn profiling on or off.
flag — `nil` turns profiling off, otherwise on
returns — the previous state of profiling.

IV.27. 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

---

[9]This is not a standard XLISP 2.0 function.

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
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 0 and n-1 inclusive
n — the upper bound (integer)
returns — a random number

(rrandom) — compute a random real number between 0 and 1 inclusive
returns — a random floating point 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

(atan *expr* [*expr2*])[10] — compute the arctangent
expr — the value of *x*
expr2 — the value of *y* (default value is 1.0)
returns — the arctangent of *x*/*y*

(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.

---

[10]This is not a standard XLISP 2.0 function.

IV.28. 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

IV.29. String Functions

(string *expr*) — make a string from a value
expr — an integer (which is first converted into its ASCII character value), string, character, or symbol
returns — the string representation of the argument

(string-search *pat str* &key :start :end)[11] — 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
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

[11]This is not a standard XLISP 2.0 function.

(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.

IV.30. 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.

IV.31. 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

IV.32. 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
~<newline> — ignore this one newline and white space on the next line up to the first non-white-space character or newline. This allows strings to continue across multiple lines

IV.33. File I/O Functions

Note that files are ordinarily opened as text. Binary files (such as standard midi files) must be opened with `open-binary` on non-unix systems.

(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

(open-binary *fname* &key :direction) — open a binary 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`

(setdir *path*)[12] — set current directory

path — the path of the new directory
returns — the resulting full path, e.g. (setdir ".") gets the current working directory, or `nil` if an error occurs

(listdir *path*)[13] — get a directory listing

path — the path of the directory to be listed
returns — list of filenames in the directory

---

[12]This is not a standard XLISP 2.0 function.

[13]This is not a standard XLISP 2.0 function.

(get-temp-path)[14] — get a path where a temporary file can be created. Under Windows, this is based on environment variables. If XLISP is running as a sub-process to Java, the environment may not exist, in which case the default result is the unfortunate choice `c:\windows\`.
returns — the resulting full path as a string

(get-user)[15] — get the user ID. In Unix systems (including OS X and Linux), this is the value of the USER environment variable. In Windows, this is currently just ‘‘nyquist’’, which is also returned if the environment variable cannot be accessed. This function is used to avoid the case of two users creating files of the same name in the same temp directory.
returns — the string naming the user

(find-in-xlisp-path *filename*)[16] — search the XLISP search path (e.g. `XLISPPATH` from the environment) for *filename*. If *filename* is not found as is, and there is no file extension, append "`.lsp`" to *filename* and search again. The current directory is not searched.
filename — the name of the file to search for
returns — a full path name to the first occurrence found

(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-int [*stream* [*length*]]) — read a binary integer from a stream
stream — the input stream (default is standard input)
length — the length of the integer in bytes (default is 4)
returns — the integer
Note: Integers are assumed to be big-endian (high-order byte first) and signed, regardless of the platform. To read little-endian format, use a negative number for the length, e.g. -4 indicates a 4-bytes, low-order byte first. The file should be opened in binary mode.

(write-int *ch* [*stream* [*length*]]) — write a binary integer to a stream

---

[14]This is not a standard XLISP 2.0 function.

[15]This is not a standard XLISP 2.0 function.

[16]This is not a standard XLISP 2.0 function.

ch — the character to write
stream — the output stream (default is standard output)
length — the length of the integer in bytes (default is 4)
returns — the integer
Note: Integers are assumed to be big-endian (high-order byte first) and signed, regardless of the platform. To write in little-endian format, use a negative number for the length, e.g. -4 indicates a 4-bytes, low-order byte first. The file should be opened in binary mode.

(read-float [*stream* [*length*]]) — read a binary floating-point number from a stream
stream — the input stream (default is standard input)
length — the length of the float in bytes (default is 4, legal values are -4, -8, 4, and 8)
returns — the integer
Note: Floats are assumed to be big-endian (high-order byte first) and signed, regardless of the platform. To read little-endian format, use a negative number for the length, e.g. -4 indicates a 4-bytes, low-order byte first. The file should be opened in binary mode.

(write-float *ch* [*stream* [*length*]]) — write a binary floating-point number to a stream
ch — the character to write
stream — the output stream (default is standard output)
length — the length of the float in bytes (default is 4, legal values are -4, -8, 4, and 8)
returns — the integer
Note: Floats are assumed to be big-endian (high-order byte first) and signed, regardless of the platform. To write in little-endian format, use a negative number for the length, e.g. -4 indicates a 4-bytes, low-order byte first. The file should be opened in binary mode.

(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)

IV.34. 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 `get-output-stream-list` return a 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

IV.35. System Functions

Note: the `load` function first tries to load a file from the current directory. A `.lsp` extension is added if there is not already an alphanumeric extension following a period. If that fails, XLISP searches the path, which is obtained from the XLISPPATH environment variable in Unix and HKEY_LOCAL_MACHINE\SOFTWARE\CMU\Nyquist\XLISPPATH under Win32. (The Macintosh version has no search path.)
(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

(info) — show information about memory usage.
    returns — `nil`

(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

(bigendianp) — is this a big-endian machine?
    returns — T if this a big-endian architecture, storing the high-order byte of an integer at the

lowest byte address of the integer; otherwise, NIL.[17]

(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

(setup-console) — set default console attributes
returns — NIL
Note: Under Windows, Nyquist normally starts up in a medium-sized console window with black text and a white background, with a window title of ‘‘Nyquist.’’ This is normally accomplished by calling `setup-console` in `system.lsp`. In Nyquist, you can avoid this behavior by setting `*setup-console*` to NIL in your `init.lsp` file. If `setup-console` is not called, Nyquist uses standard input and output as is. This is what you want if you are running Nyquist inside of emacs, for example.

(echoenabled *flag*) — turn console input echoing on or off
flag — T to enable echo, NIL to disable
returns — NIL
Note: This function is only implemented under Linux and Mac OS X. If Nyquist I/O is redirected through pipes, the Windows version does not echo the input, but the Linux and Mac versions do. You can turn off echoing with this function. Under windows it is defined to do nothing.

IV.36. File I/O Functions

IV.36.1. 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

[17]This is not a standard XLISP 2.0 function.

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.

IV.36.2. 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.

IV.36.3. 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