

# Nyquist Reference Manual

Version 3.23

Copyright 2013-2024 by Roger B. Dannenberg

January 7, 2024

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

# Contents

1	Introduction and Overview	4
1.1	Installation	4
1.1.1	Troubleshooting	5
1.2	Using NyquistIDE	5
1.3	Using SAL	6
1.4	Command Line	6
1.5	Helpful Hints	7
1.6	Using Lisp	8
1.7	Examples	8
1.7.1	Non-Sinusoidal Waveforms	8
1.7.2	Using Wavetables	9
1.7.3	Sequences	10
1.7.4	Envelopes	10
1.7.5	Piece-wise Linear Functions	11
1.8	Predefined Constants	12
1.9	More Examples	14
2	The NyquistIDE Program	16
2.1	NyquistIDE Overview	16
2.2	The Button Bar	17
2.3	Command Completion	18
2.4	Extension Manager	18
2.5	Browser	19
2.6	Envelope Editor	19
2.7	Equalizer Editor	20
2.8	UPIC Editor	20
3	Behavioral Abstraction	22
3.1	The Environment	22
3.2	Sequential Behavior	24
3.3	Simultaneous Behavior	24
3.4	Sounds vs. Behaviors	25
3.5	The At Transformation	26
3.6	The Stretch Transformation	26
3.7	Nested Transformations	27

3.8	Defining Behaviors . . . . .	27
3.9	Overriding Default Transformations . . . . .	27
3.10	Sampling Rates . . . . .	28
4	Continuous Transformations and Time Warps . . . . .	30
4.1	Simple Transformations . . . . .	30
4.2	Time Warps . . . . .	32
4.3	Abstract Time Warps . . . . .	32
4.4	Nested Transformations . . . . .	35
5	More Examples . . . . .	36
5.1	Stretching Sampled Sounds . . . . .	36
5.2	Saving Sound Files . . . . .	37
5.3	Memory Space and Normalization . . . . .	38
5.4	Frequency Modulation . . . . .	40
5.5	Building a Wavetable . . . . .	42
5.6	Filter Examples . . . . .	42
5.7	DSP in Lisp . . . . .	43
6	SAL . . . . .	46
6.1	SAL Syntax and Semantics . . . . .	46
6.1.1	Expressions . . . . .	47
6.1.1.1	Simple Expressions . . . . .	47
6.1.1.2	Operators . . . . .	48
6.1.1.3	Function Calls . . . . .	50
6.1.1.4	Array Notation . . . . .	50
6.1.1.5	Conditional Values . . . . .	50
6.1.2	SAL Statements . . . . .	50
6.1.2.1	begin and end . . . . .	50
6.1.2.2	chdir . . . . .	51
6.1.2.3	define variable . . . . .	51
6.1.2.4	define function . . . . .	51
6.1.2.5	exec . . . . .	52
6.1.2.6	if . . . . .	52
6.1.2.7	when . . . . .	53
6.1.2.8	unless . . . . .	53
6.1.2.9	load . . . . .	53
6.1.2.10	loop . . . . .	54
6.1.2.11	play . . . . .	56
6.1.2.12	plot . . . . .	56
6.1.2.13	print . . . . .	56
6.1.2.14	display . . . . .	56
6.1.2.15	return . . . . .	57
6.1.2.16	set . . . . .	57
6.1.2.17	with . . . . .	58
6.1.2.18	exit . . . . .	58

6.2	Interoperability of SAL and XLISP . . . . .	58
6.2.1	Function Calls . . . . .	59
6.2.2	Symbols and Functions . . . . .	59
6.2.3	Playing Tricks On the SAL Compiler . . . . .	59
7	Nyquist Functions . . . . .	61
7.1	Sounds . . . . .	61
7.1.1	What is a Sound? . . . . .	61
7.1.2	Multichannel Sounds . . . . .	62
7.1.3	Accessing and Creating Sound . . . . .	62
7.1.4	Miscellaneous Functions . . . . .	67
7.2	Behaviors . . . . .	71
7.2.1	Using Previously Created Sounds . . . . .	71
7.2.2	Sound Synthesis . . . . .	72
7.2.2.1	Oscillators . . . . .	76
7.2.2.2	Piece-wise Approximations . . . . .	79
7.2.2.3	Filter Behaviors . . . . .	83
7.2.2.4	Effects . . . . .	88
7.2.2.5	Physical Models . . . . .	88
7.2.2.6	Phase Vocoder . . . . .	92
7.2.2.7	More Behaviors . . . . .	93
7.3	Transformations . . . . .	97
7.4	Combination and Time Structure . . . . .	100
7.5	Sound File Input and Output . . . . .	103
7.6	Low-level Functions . . . . .	110
7.6.1	Creating Sounds . . . . .	110
7.6.2	Signal Operations . . . . .	112
7.6.3	Filters . . . . .	118
7.6.4	Table-Lookup Oscillator Functions . . . . .	122
7.6.5	Phase Vocoder Functions . . . . .	124
7.6.6	Physical Model Functions . . . . .	125
7.6.7	Sequence Support Functions . . . . .	127
8	Nyquist Globals . . . . .	128
9	Interactive Nyquist . . . . .	132
9.1	Interactive Control with the NyquistIDE . . . . .	133
9.1.1	Creating a Control Panel . . . . .	133
9.1.2	Creating Controls . . . . .	133
9.1.3	Accessing Control Values . . . . .	134
9.1.4	Starting and Stopping Sounds . . . . .	135
9.2	Using Open Sound Control . . . . .	136
9.2.1	Sending Open Sound Control Messages . . . . .	137
9.2.2	Python3 OSC Interface Demo . . . . .	137
9.2.3	Test Programs in C . . . . .	137
9.2.4	The ser-to-osc Program . . . . .	138

10	Time/Frequency Transformation	139
10.1	Spectral Processing	140
11	MIDI, Adagio, and Sequences	143
11.1	The SEQ Type	143
11.2	Adagio Score Language	145
11.3	Specifying Attributes	146
11.3.1	Time	146
11.3.2	Pitch	146
11.3.3	Duration	147
11.3.4	Next Time	148
11.3.5	Rest	148
11.3.6	Articulation	148
11.3.7	Loudness	149
11.3.8	Voice	149
11.3.9	Timbre (MIDI Program)	149
11.3.10	Tempo	150
11.3.11	Rate	150
11.4	Default Attributes	151
11.5	Examples	151
11.6	Advanced Features	154
11.6.1	Time Units and Resolution	154
11.6.2	Multiple Notes Per Line	154
11.6.3	Control Change Commands	155
11.6.4	Multiple Tempi	156
11.6.5	MIDI Synchronization	157
11.6.6	System Exclusive Messages	157
11.6.7	Control Ramps	158
11.6.8	The !End Command	159
11.6.9	Calling C Routines	159
11.6.10	Setting C Variables	159
12	Linear Prediction Analysis and Synthesis	161
12.1	LPC Classes and Functions	161
12.2	Low-level LPC Functions	163
13	Developing and Debugging in Nyquist	164
13.1	Debugging	164
13.2	Useful Functions	165
14	Xmusic and Algorithmic Composition	168
14.1	Xmusic Basics	168
14.2	Xmusic Patterns	168
14.2.1	Nested Patterns	169
14.2.2	Periods	169
14.2.3	General Parameters for Creating Pattern objects	170
14.2.4	cycle	171

14.2.5	line	171
14.2.6	random	171
14.2.7	palindrome	172
14.2.8	heap	172
14.2.9	accumulation	173
14.2.10	copier	173
14.2.11	accumulate	173
14.2.12	sum	174
14.2.13	product	174
14.2.14	eval	174
14.2.15	length	175
14.2.16	window	175
14.2.17	markov	175
14.3	Random Number Generators	177
14.4	Score Generation and Manipulation	185
14.4.1	Keyword Parameters	185
14.4.2	Using score-gen	186
14.4.3	Score Manipulation	187
14.4.4	Xmusic and Standard MIDI Files	192
14.4.5	Workspaces	194
14.4.6	Utility Functions	195
15	Nyquist Libraries	196
15.1	Statistics	196
15.2	Plots	196
15.3	Labeling Audio Events, Marking Audio Times, Displaying Marked Audio Times	196
15.4	Linear Regression	197
15.5	Vector Math, Linear Algebra	197
15.6	Piano Synthesizer	197
15.7	Dynamics Compression	197
15.8	Clipping Softener	199
15.9	Graphical Equalizer	199
15.10	Sound Reversal	200
15.11	Time Delay Functions	200
15.12	Multiple Band Effects	201
15.13	Granular Synthesis	201
15.14	Chowning FM Voices	202
15.15	Atonal Melody Composition	202
15.16	MIDI Utilities	202
15.17	Reverberation	203
15.18	DTMF Encoding	203
15.19	Dolby Surround(R), Stereo and Spatialization Effects	203
15.20	Drum Machine	205

A	Extending Nyquist	208
A.1	Translating Descriptions to C Code	208
A.2	Rebuilding Nyquist	208
A.3	Accessing the New Function	209
A.4	Why Translation?	209
A.5	Writing a .alg File	209
A.6	Attributes	210
A.7	Generated Names	214
A.8	Scalar Arguments	215
B	Intgen	216
B.1	Overview	216
B.1.1	Extending Xlisp	217
B.2	Header file format	217
B.3	Using #define'd macros	219
B.4	Lisp Include Files	220
B.5	Example	220
B.6	More Details	220
C	XLISP: An Object-oriented Lisp	221
C.1	Introduction	221
C.2	A Note From The Author	222
C.3	XLISP Command Loop	222
C.4	Special Characters	222
C.5	Break Command Loop	223
C.6	Data Types	223
C.7	The Evaluator	224
C.8	Lexical Conventions	224
C.9	Readtables	225
C.10	Lambda Lists	226
C.11	Objects	227
C.12	The "Object" Class	228
C.13	The "Class" Class	228
C.14	Profiling	229
C.15	Symbols	229
C.16	Evaluation Functions	230
C.17	Symbol Functions	231
C.18	Property List Functions	233
C.19	Array Functions	233
C.20	List Functions	233
C.21	Destructive List Functions	236
C.22	Predicate Functions	237
C.23	Control Constructs	240
C.24	Looping Constructs	242
C.25	The Program Feature	242

---

C.26 Debugging and Error Handling . . . . .	244
C.27 Arithmetic Functions . . . . .	245
C.28 Bitwise Logical Functions . . . . .	247
C.29 String Functions . . . . .	248
C.30 Character Functions . . . . .	250
C.31 Input/Output Functions . . . . .	252
C.32 The Format Function . . . . .	253
C.33 File I/O Functions . . . . .	254
C.34 String Stream Functions . . . . .	256
C.35 System Functions . . . . .	257
C.36 File I/O Functions . . . . .	259
C.36.1 Input from a File . . . . .	259
C.36.2 Output to a File . . . . .	260
C.36.3 A Slightly More Complicated File Example . . . . .	260

## 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](mailto:Dannenberg@CS.CMU.EDU) via email, or contact me by ordinary mail: 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 reimplement 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 Integrated Development Environment for Windows. Dave Mowatt contributed the original version of NyquistIDE, 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. John Chowning and Jorge Sastre contributed a SAL implementation of Chowning's voice synthesis technique. Chen Ziheng implemented the majority of the cmupv phase vocoder library used for Nyquist's phasevocoder and pv-pitch-time functions.

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 NyquistIDE 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. Dmitry Portnoy wrote the original "UPIC" editor. Azarakhsh Keipour wrote the original extension manager.

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 macOS, 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.

The core language of Nyquist is Lisp. Nyquist extends Lisp for sound synthesis and processing. Starting with Version 3, Nyquist supports a variant of SAL syntax. SAL is a language distinct from Lisp, but it can be translated to Lisp and SAL and Lisp programs can be combined. SAL was introduced in Rick Taube's Common Music system. Since there are some differences between Nyquist and Common Music, 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. An introduction to SAL is in Mary Simoni and Roger B. Dannenberg, *Algorithmic Composition: A Guide to Composing Music with Nyquist* [2]. To learn about Lisp, an excellent text by Touretzky is recommended [3]. Appendix C is the reference manual for XLISP, of which Nyquist is a superset.

## 1.1 Installation

Nyquist is a C++ program intended to run under various operating systems including Unix, macOS, and Windows. Nyquist is based on Lisp, but it includes its own Lisp interpreter (a modified version of XLISP), so you do not need to install some other Lisp to run Nyquist. Other Lisp systems are not compatible with Nyquist.

Most Nyquist users run Nyquist under the Nyquist IDE, which is written in Java and depends on the Java runtime system. Most systems already have Java, but if you do not, you will need to install it. Java from Oracle is recommended, e.g. Java SE Development Kit 17. When you install the Nyquist IDE, you will automatically get Nyquist and a set of runtime libraries.

There are generally two ways to install Nyquist:

- Get a pre-compiled version of the Nyquist IDE for Windows or macOS. The Windows version comes packaged in an installer that installs and configures the Nyquist IDE. The macOS version unpacks to a complete OS X application.
- Compile from sources. There is one set of sources for Mac, Windows, and Unix. Instructions for building applications from the sources are provided in the files `sys/win/README.txt`, `sys/mac/README.txt`, and `sys/unix/README.txt`.

You can download source code and precompiled versions from the Nyquist project on SourceForge (<http://sourceforge.net/projects/nyquist>). The latest source code can be obtained via Subversion (svn) using the following:

```
svn co svn://svn.code.sf.net/p/nyquist/code/trunk/nyquist nyquist
```

or by checking out nyquist using a graphical interface svn client such as TortoiseSVN for Windows.

### 1.1.1 Troubleshooting

All versions of Nyquist includes a Readme.txt file in the top-level `nyquist` directory with general information. Additional information and instructions on installation can be found in:

- `nyquist/doc/readme-win.txt`
- `nyquist/doc/readme-mac.txt`
- `nyquist/doc/readme-linux.txt`

## 1.2 Using NyquistIDE

The program named NyquistIDE is an “integrated development environment” for Nyquist. When you run NyquistIDE, it starts the Nyquist program and displays all Nyquist output in a window. NyquistIDE 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 NyquistIDE 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. NyquistIDE 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 NyquistIDE.

## 1.4 Command Line

When run from the command line, Nyquist (usually the command is `ny`) looks for some options and some files to load. There are several options.

The `-Rpaths` option gives Nyquist a list of paths (separated by semicolons) to directories where Nyquist is allowed to read files. Files in any subdirectories of the listed directories can also be read, along with the directories themselves. This option can be used to restrict access to files.

The `-Wpaths` option gives Nyquist a list of paths (separated by semicolons) to directories where Nyquist is allowed to write files (including sound files). If `-W` is used, Nyquist will not open audio output to play sounds and will not allow access to Open Sound Control. This option can be used to protect files from accidentally malicious code. (If you are truly worried about Nyquist code, you should run Nyquist in a virtual machine or other isolated machine. It is unlikely that Nyquist is secure from all attacks.)

The `-Lrun-time-limit` option gives Nyquist a run time limit. If the limit is exceeded, Nyquist prints an error message and exits. The time is based on Lisp expression evaluations rather than CPU seconds or wall time or even execution time involved in printing or signal processing. One unit of run time is on the order of 10ms of CPU time (for a medium speed computer in 2019), but depending on what you are computing, a unit of run time can vary from 60ms to 1.5ms. You can use the `GET-RUN-TIME` function to read the internal run-time counter to see how much computation you have done in the same units.

The `-Mmemory-limit` option gives Nyquist a memory limit. If the limit (in megabytes) is exceeded, Nyquist prints an error message and exits. The limit includes all XLISP data (cons cells, strings, etc.) plus sample blocks and tables allocated for signal processing.

The `-Ttranscript-file-name` option copies console output to a transcript file.

The `-V` option sets verbose mode. If set, any files specified on the command line to be loaded are loaded with the verbose option.

Additionally, the command line can name files to be loaded. Nyquist begins by loading `init.lsp`, which by default is in the `nyquist/runtime` directory. This file loads all the standard Nyquist functions. After that, Nyquist will load any other files named on the command line.

Finally, Nyquist expects to find search paths in the `XLISPPATH` environment variable (or for Windows, see installation instructions; the path is saved in the Windows Registry). Normally, you must set `XLISPPATH` for Nyquist to find `init.lsp` and the definitions of most Nyquist functions. Be sure you have `system.lsp` in `nyquist/runtime`. Part of the installation process copies the appropriate `system.lsp` from `nyquist/sys/*` to `nyquist/runtime` where it can be found and loaded by `init.lsp`.

## 1.5 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 NyquistIDE, 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 helpful 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](http://sourceforge.net).

The NyquistIDE 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 and using Lisp syntax, try the `:print t` option of the `load` function. 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.

## 1.6 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 NyquistIDE.

## 1.7 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/lib/examples.sal`. Corresponding Lisp syntax examples are in the file `nyquist/lib/examples.lisp`.

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.

### 1.7.1 Non-Sinusoidal 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 special function, `build-harmonic` is used to compute the harmonics, but it is just a convenient wrapper for the primitive function `snd-sine`.

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 to local variable `wave`.

A complete Nyquist wavetable specification 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
    with wave = 0.5 * build-harmonic(1, 2048) +
               0.25 * build-harmonic(2, 2048) +
               0.125 * build-harmonic(3, 2048) +
               0.0625 * build-harmonic(4, 2048)
    set *mytable* = list(wave, hz-to-step(1.0), #t)
  end
```

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

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

## 1.7.2 Using 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 wavetable 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.

### 1.7.3 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, *mytable*)

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 `*mytable*` as a wave table. In earlier examples, we saw `osc` with only two parameters. In that case, `*table*` is used, but here we specify our new table `*mytable*`.

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 1.8 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.

### 1.7.4 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 1.1.

```
; 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*.

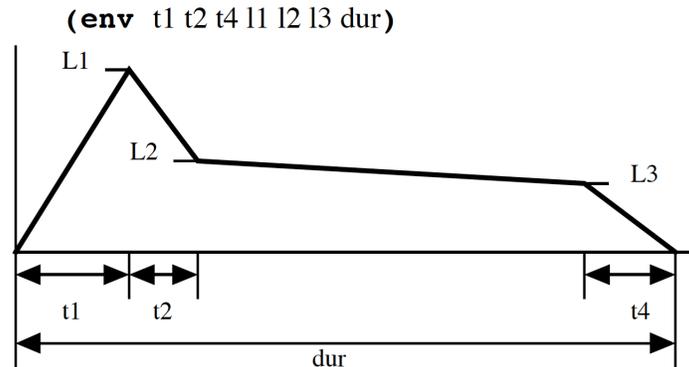


Figure 1.1: An envelope generated by the env function.

In the next example, The *stretch* operator ( $\sim$ ) 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 [1].

### 1.7.5 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 pw1. Since pw1 is used in some examples later on, we will take a look at pw1 now. The pw1 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:

```
; symmetric rise to 10 (at time 1) and fall back to 0 (at time 2):
;
pw1(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.
;

```

## 1.8 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

## 1.9 More Examples

More examples can be found in the directory `demos`, part of the standard Nyquist release. In the Apple OS X version of Nyquist, the `demos` and `doc` directories are inside the NyquistIDE application bundle. To make it easier to access these hidden files, the NyquistIDE installs links to these directories the first time you run it. The links will be in the same directory as the NyquistIDE itself.

The file `demos/index.htm` is an index to many examples and demonstrations. Many examples are now Nyquist Extensions which can be downloaded by the NyquistIDE using the Window : Extension Manager menu item. Here are some highlights of what is available:

- Code to create atonal melodies (`nyquist/lib/atonal/atonal-melodies.sal`).
- How to make arpeggios (`nyquist/lib/arpeggiator/arpeggiator.htm` and `nyquist/lib/arpeggiator/arp.sal`)
- Gong sounds by additive synthesis (`nyquist/lib/pmoraes/b1.lsp` and `nyquist/lib/pmoralies/mateos/gong.lsp`)
- Risset’s spectral analysis of a chord (`nyquist/lib/pmoraes/b2.lsp`)
- Bell sounds (`nyquist/demos/pmoraes/b3.lsp`, `nyquist/lib/pmoraes/e2.lsp`, `nyquist/lib/pmoraes/partial.lsp`, and `nyquist/lib/mateos/bell.lsp`)
- Drum sounds by Risset (`nyquist/lib/pmoraes/b8.lsp`)
- Shepard tones (`nyquist/lib/shepard/shepard.lsp` and `nyquist/lib/pmoraes/b9.lsp`)
- Random signals (`nyquist/lib/pmoraes/c1.lsp`)
- Buzz with formant filters (`nyquist/lib/pmoraes/buzz.lsp`)
- Computing samples directly in Lisp (using Karplus-Strong and physical modelling as examples) (`nyquist/lib/pmoraes/d1.lsp`)
- FM Synthesis examples, including FM voices (`nyquist/lib/fm-voices-chowning/fm-voices-chowning.sal`) designed by John Chowning, tuba sound (`nyquist/lib/mateos/tuba.lsp`), and see `nyquist/lib/pmoraes/e2.lsp` for examples of FM bell, wood drum, brass sounds, and clarinet sounds.

- Rhythmic patterns ([nyquist/lib/rhythm/rhythm\\_tutorial.html](nyquist/lib/rhythm/rhythm_tutorial.html))
- Drum Samples and Drum Machine (<nyquist/lib/plight/drum.lsp>). (See Section [15.20](#)).

## 2 The NyquistIDE Program

The NyquistIDE program combines many helpful functions and interfaces to help you get the most out of Nyquist. NyquistIDE is implemented in Java, and you will need the Java runtime system or development system installed on your computer to use NyquistIDE. The best way to learn about NyquistIDE 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 NyquistIDE Overview

The NyquistIDE runs the command-line version of Nyquist as a subtask, so everything that works in Nyquist should work when using the NyquistIDE and vice-versa. Input to Nyquist is usually entered in the top left window of the NyquistIDE. 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 NyquistIDE is to create or open one or more files. You edit these files and then click the Load button. To load a file, NyquistIDE 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 NyquistIDE 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, NyquistIDE 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 NyquistIDE 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.

Many of the key commands for editing files are standard and built into Java. A special command, "Copy to Lisp," or Control-U (Command-U on macOS) copies the selection and enters it into the input window as if it had been typed there.

## 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 2.5.)
- **EQ** – Equivalent to the Window:EQ menu item. (See Section 2.7.)
- **EnvEdit** – Equivalent to the Window:Envelope Edit menu item. (See Section 2.6.)
- **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, NyquistIDE maintains a command-completion window. As you type the first letters of function names, NyquistIDE 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 macOS, hold down the Alt/Option key and click) on an entry, NyquistIDE 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 NyquistIDE window or in an external browser (determined by the “Use window in NyquistIDE 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 Extension Manager

The NyquistIDE Extension Manager is a facility to download extensions that are written in Lisp or SAL. (If you want to extend the nyquist program itself or the NyquistIDE program, you will have to write C or Java respectively and recompile. See Appendix A for information on writing new DSP functions, also known as unit generators.)

Selecting the Extension Manager item in the Window menu will open a window in the NyquistIDE with a list of extensions. Each extension is stored as a subdirectory of the `lib` directory, which should be on the search path (`XLISPPATH`) for `nyquist`. A list of extensions is hosted in a fixed location (<https://www.cs.cmu.edu/~music/nyquist/extensions/extlist.txt>) along with secure checksums to guard against malicious code.

When you install an extension, a single file is first downloaded using a URL that is obtained from the list of extensions, and the file is scanned for a header that can specify additional files to download.

Some special files may be included in an extension. `autoload.lisp` is a Lisp file that is loaded automatically when `nyquist` is started. Nyquist scans the subdirectories of the `lib` directory to find all the `autoload.lisp` files. Typically, `autoload.lisp` files are used to create stubs for functions in the extensions so that extension code is mostly loaded dynamically on demand, saving time and space when `nyquist` starts.

Extensions may also include `nyquistwords.txt`, which is a description of functions in the extensions that is used to generate the completion list. When an extension function appears in the completion list, it is followed by "ext:" and the extension name, e.g. the completion item "speed-dial(list) ext: dtmf" means that the `speed-dial` function is defined in the `dtmf` directory. By convention, any function in the completion list should be directly callable, with an entry in `autoload.lisp` so that the function will be dynamically loaded.

To create a new extension, see the extension `ext-template1` for an example with documentation in HTML, or `ext-template2` for a simple single-file extension where the documentation consists of comments in the

SAL source file.

To submit an extension for possible publication, it is best to put the extension file(s) in a local directory and use a customized local list of extensions. The Nyquist Preferences includes a button where you can designate a local file for the list of extensions. It should have the same format as `extlist.txt` at the URL given above (you can simply open the URL in your browser to see it. The extension under development can be referenced in the extension list using a URL with the `file://` protocol. For the checksum, use any value. The Extension Manager will compute the correct checksum and display it, so you can then copy that into your `extlist.txt`, click the Update button to fetch it, and try again to install the extension.

## 2.5 Browser

If you click on the Browse button or use the Window:Browse menu command, NyquistIDE 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.

## 2.6 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 14.4.5). 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 NyquistIDE. 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 `pw1` or `pwe`, or you can specify the initial values using `pw1v` or `pwev`. The envelope editor uses `pw1` or `pwe` if no point is explicitly entered as the initial or final point. To create a `pw1v` 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.

## 2.7 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 used to record changes.

## 2.8 UPIC Editor

The UPIC Editor is inspired by the UPIC system by Iannis Xenakis at the Centre d'Études de Mathématique et Automatique Musicales (CEMaMu). The UPIC Editor is accessed by the “Upic Edit” menu item in the “Window” menu of the NyquistIDE. Once opened, you can draw pitch contours in the main panel by pressing the left mouse button and dragging with the mouse down. Contours represent tones in a frequency vs. time coordinate system. Any contour can be deleted by right-clicking (or shift-clicking on an Apple computer) to select the contour (indicated by the color red), and typing the Delete key.

A collection of contours can be saved to a file and later retrieved using the items in the File menu (use the File menu in the UPIC Editor window, not in the main NyquistIDE window.) The file is a SAL program in a special format that can be parsed by the UPIC Editor. The file can also be loaded into Nyquist using the File:Load menu item, or by executing a load command in Nyquist.

The panel at the top of the editor offers control over various parameters. The Name box is a Nyquist variable name. This name takes effect when you save a file from the UPIC Editor. The variable name is stored in the file so that when a UPIC Editor-generated file is loaded into Nyquist, the data is assigned to this variable name. The data is a list of contours, where each contour specifies a waveform, an envelope, and a list of time-frequency coordinates.

The next item in the panel is the Waveform box. The Waveform box names a waveform for a contour. Default waveforms are sinusoid, triangle, and sawtooth, but you can type in your own names. The currently selected waveform is stored with the contour when it is created (entered by drawing). You cannot change or edit the waveform name associated with a contour once the contour is created, but you can always delete the contour and replace it. The Envelope box names an envelope for a contour. The envelope names a Nyquist function. The default, `upic-env` is a trapezoid shape with an onset time and offset time of 10ms. As with waveforms, the envelope is stored with each contour when the contour is created and cannot be edited.

The Stop Time box gives the duration of the drawing area in seconds. The Min Freq box gives the minimum frequency (at the bottom of the drawing area), and the Max Freq box gives the maximum frequency (at the top of the drawing area). The vertical frequency axis can use a linear scale corresponding to frequency in Hertz or a logarithmic scale corresponding to semitones. The “linear” checkbox selects the linear scale. When *any* of these parameters (described in this paragraph and delimited by the border labeled “Range” on the control panel) is changed, you must press the Update Range button for the change to take effect.

The Background menu lets you display a grid that indicates pitch locations. The “C’s” item draws a line at C in every visible octave. E.g. middle C is about 260 Hz, so a reference line will be drawn near 260 Hz. Lines will be drawn around 130 Hz (an octave below middle C), and around 520 Hz (an octave above middle C), etc. The “GrandStaff” menu item draws reference lines for each line of the grand staff commonly used for piano music. The pitches are G2, B2, D3, F3, A3, E4, G4, B4, D5, and F5. Finally, you can load a picture using the Background:Load Picture... menu item. Then, the Background:Show Picture menu item toggles whether the picture is displayed or not. This feature allows you to trace an image. (For example, see the Sonic Self-Portrait at <http://www.cs.cmu.edu/~rbd>.) You may wish to use an image editor to lighten the image so that superimposed contours will be more visible.

Each change to the Range data, background choice, and each entry of a contour is an action that you can undo or redo with the Undo and Redo buttons.

To convert UPIC data into sound, first load `upic.sal` and load a file generated by the UPIC Editor. Now, suppose the variable name used is `upicdata`. You can play the data by writing

```
play upic(upicdata)
```

If you created your own names for waveforms or envelopes, you must be sure that these exist before calling the `upic` function. Each waveform must be the name of a variable which is set to a Nyquist wave table. (See Section 1.7.1 for information on how to create a wave table.) Also, each envelope must name a function with no parameters that will return an amplitude envelope. The following is the built-in definition for `upic-env`:

```
define function upic-env()
  return env(0.01, 0.01, 0.01, 1, 1, 1)
```

To make a custom envelope function named `upic-smooth` with a 0.2 second attack and a 0.3 second decay, you could write:

```
define function upic-smooth()
  return env(0.2, 0.01, 0.3, 1, 1, 1)
```

## 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`. See also `get-duration()` and `get-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 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!
play seq(a-snd, a-snd @ 3) ; ALSO 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 both of these cases, the second `a-snd` *should* start at time 0.0, just like the first. 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. The second case emphasizes this: Nyquist will return a sound duration of only 2 seconds because the `@` transform does not apply to a sound. 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)) ; CORRECT!
play seq(cue(a-snd), cue(a-snd) @ 3) ; CORRECT! Duration = 2 seconds
```

These examples are correct because the second `cue` expressions will shift the sound stored in `a-snd` to start at the desired times.

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. Imagine a different design where `sim` takes all of its parameters (a set of sounds) and lines them up to start at the same time. That would make it almost impossible to arrange sounds with different start times. You might try using `@` and write `cue(a-snd) @ 0.7`, but this expression just creates a particular sound, and we are assuming that `sim` aligns *all* sounds to start together, so this design of `sim` would “undo” the shift (`@`)! This design is no good, so in Nyquist, `sim` respects the intrinsic starting times of sounds. This creates a new problem when you want to reuse an existing sound, except you want it to start at a different time. The solution is the `cue` operation, which makes a copy of the sound with the starting time determined by the environment. (This is a virtual copy that avoids memory allocation or actually copying samples.)

### 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 determine 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 abstract 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 Sampling 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 NyquistIDE. 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 `chromatic-scale` behavior 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`.

If you try this, you will discover that the `chromatic-scale` no longer plays a chromatic scale. You will hear the first 4 notes going up in intervals of 5 semitones (perfect fourths) followed by repeated pitches. What is happening is that the `sustain` operation applies to `pitch-rise` in addition to `tone-seq`, so now the 4s ramp from 0 to 16 becomes a 0.8s ramp. To fix this problem, we need to shield `pitch-rise` from the effect of `sustain` using the `sustain-abs` transformation. Here is a corrected version of `chromatic-scale`:

```
define function chromatic-scale()
  return transpose(sustain-abs(1, pitch-rise()), tone-seq())
```

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`, `ramp`, 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  $1/\text{tempo}$ . 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  $.4/0.25 = 1.6$ . It may be easier to think in reciprocal terms: the initial tempo is  $0.25/0.4 = .625$ . 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 4.1 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 4.2 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 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.

)

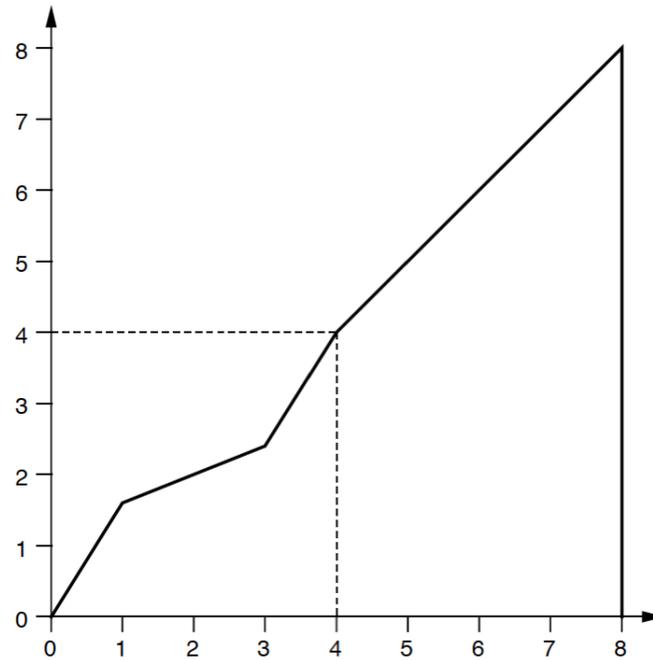


Figure 4.1: The result of `(warp4)`, intended to map 4 seconds of score time into 4 seconds of real time. The function extends beyond 4 seconds (the dashed lines) to make sure the function is well-defined at location (4, 4). Nyquist sounds are ordinarily open on the right.

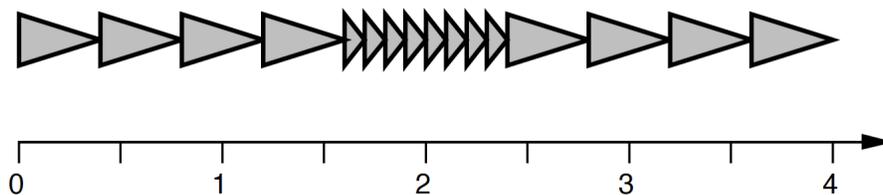


Figure 4.2: When `(warp4)` is applied to `(tone-seq-2)`, the note onsets and durations are warped.

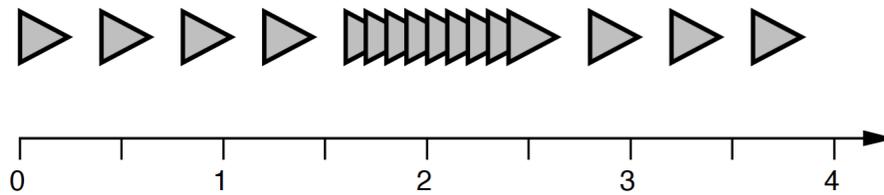


Figure 4.3: When `(warp4)` is applied to `(tone-seq-3)`, the note onsets are warped, but not the duration, which remains a constant 0.25 seconds. In the fast middle section, this causes notes to overlap. Nyquist will sum (mix) them.

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 environment. Figure 4.3 illustrates the result graphically.

## 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.

The last line of this example 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, pw1(0.1))
```

```
; make a longer sine tone -- note that the duration of
; the modulator determines the duration of the tone
;
play fmosc(c4, pw1(0.5))
```

In the example above, `pw1` (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 `pw1` is in order. This operation constructs a piece-wise linear function sampled at the `*control-rate*`. 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 `pw1` 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, pw1(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, pw1(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-rate\**, 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<sub>4</sub>) 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 [nyquist/lib/warble/warble\\_tutorial.htm](http://nyquist/lib/warble/warble_tutorial.htm).

## 5.5 Building a Wavetable

In Section 1.7.1, 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 `nyquist/lib/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(), pw1(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 [nyquist/lib/wind/wind\\_tutorial.htm](http://nyquist/lib/wind/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 A. 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 A. 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 C) 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 NyquistIDE 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.

The set SAL identifiers is difficult to describe due to a number of interacting rules designed to prevent surprising or confusing code. The exact details can be found in the symbol-token? function in sal-parse.lsp. To determine if an identifier is allowed, you can also follow the following example, which tests whether \$a@ is a valid identifier:

```
SAL> print type-of(quote($a@))
```

If this statement prints `SYMBOL`, then the quoted expression is a valid identifier. If SAL reports a "parse error" or some other type, then the quoted expression is not a valid identifier.

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 NyquistIDE 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 (which represent XLISP's `T` or `NIL`, SAL identifiers (representing XLISP symbols), strings, SAL operators (representing XLISP symbols), and nested 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 (`nil`) are the same value. Since SAL is based on Lisp, `#f` and `{}` (the empty list) and `nil` are all 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 <code>equal</code> for non-lists, and comparing lists element-by-element recursively) <sup>1</sup>
!=	not equal
>	greater than
<	less than
>=	greater than or equal

---

<sup>1</sup>In the Common Music implementation of SAL, `=` compiles to the Lisp `=` function, so this is an incompatible difference.

<=	less than or equal
~=	approximately equal. Numbers are approximately equal if they are within *~=tolerance* of each other. *~=tolerance* is initially 0.000001. Non-numbers are compared with the XLISP equal function, and lists are compared element-by-element (recursively) using ~=. <sup>2</sup>
&	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:

```

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

```

<sup>2</sup>In the Common Music implementation of SAL, ~= compiles to the Lisp equal function, so this is an incompatible difference.

### 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}<sup>+</sup> indicate one or more occurrences of a syntax element. Braces with a star {like this}<sup>\*</sup> indicate zero or more occurrences of a syntax element: {*non-terminal*}<sup>\*</sup> is equivalent to [ {*non-terminal*}<sup>+</sup> ].

### 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 NyquistIDE. 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 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
```

### 6.1.2.6 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 the *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
  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.

#### 6.1.2.7 when

when *test statement*

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

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

#### 6.1.2.8 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
```

#### 6.1.2.9 load

load *path-expression*

The load command loads a file named by *path-expression*, which must be either a literal string or a variable name. When evaluated, *path-expression* must result in a string path name for the file. The extension `.sal` is appended to the path if needed. 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 name ends in the extension `.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`.

### 6.1.2.10 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. **Warning:** All for clauses evaluate *before* stopping conditions are evaluated, which means if the loop iterates N times, the for clauses run N+1 times. To avoid evaluation after the last iteration, you can often use `set var = expr2`.

```
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 initialized 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, or above clause, no iteration 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 iterating 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 below 22 by 2
    print i
  end

; collect even numbers in a list
loop
  with lis
    for i from 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 set result = reverse(lis)
  end
; since "to" is used, the loop stops when i > 10
; now, the variable result has a list of evens: {0 2 4 6 8 10}

```

```

; 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)

```

### 6.1.2.11 play

play *expr*

The play statement plays the sound denoted by *expr*, an expression.

### 6.1.2.12 plot

plot *expr* {, *dur*, *n*}

The plot statement plots the sound denoted by *expr*, an expression. If you plot a long sound, the plot statement will by default truncate the sound to 2.0 seconds and resample the signal to 1000 points. The optional *dur* is an expression that specifies the (maximum) duration to be plotted, and the optional *n* specifies the number of points to be plotted. Executing a plot statement is equivalent to calling the `s-plot` function (see Section 7.5).

### 6.1.2.13 print

print *expr* {, *expr*}\*

The print statement prints the values separated by spaces and followed by a newline. [Note that in the original SAL, the newline is printed *before* the values, not after.]

### 6.1.2.14 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.”

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

### 6.1.2.15 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.

### 6.1.2.16 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
```

### 6.1.2.17 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 initialized only when the loop is entered, not on each iteration.

### 6.1.2.18 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 evaluates 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 1.8).

### 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<sup>1</sup>.

**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)` [SAL]

`(sref sound time)` [LISP]

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*):

<sup>1</sup>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.

$$t = (n/srate) + t0$$

Thus, the Lisp code to access the  $n^{\text{th}}$  sample of a sound would look like:

```
(sref sound (global-to-local (+ (/ n (snd-srate sound)) (snd-t0 sound))))
```

Or in SAL, it 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 (shown first in LISP and then in SAL syntax):

```
> (sref (ramp 1) 0.5) ; evaluate a ramp at time 0.5
0.5
SAL> print 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
SAL> sref(ramp(1), 0.5) @ 2.0 ; 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 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)` [SAL]

`(sref-inverse sound value)` [LISP]

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)` [SAL]

`(snd-from-array t0 sr array)` [LISP]

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

`snd-fromarraystream(t0, sr, object)` [SAL]

`(snd-fromarraystream t0 sr object)` [LISP]

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 C.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 FLONUMs. The values in these arrays are concatenated to form the samples of the resulting sound. In the current implementation, all arrays must have the same length. 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)` [SAL]

`(snd-fromobject t0 sr object)` [LISP]

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 C.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)` [SAL]

`(snd-extent sound maxsamples)` [LISP]

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)` [SAL]

`(snd-fetch sound)` [LISP]

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)` [SAL]

`(snd-fetch-array sound len step)` [LISP]

Reads sequential arrays of samples from *sound*, returning either an array of FLONUMs 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. When an array is returned, it will have *len* samples. If necessary, `snd-fetch-array` will read zeros beyond the end of the sound to fill the array. When this happens, `*result*` is set to a FIXNUM number of samples in the array that were read from the sound before the physical stop time of the sound. If all samples in the array are “valid” samples from the sound (coming from the sound before the sound terminates), `*result*` is set to NIL. The `*result*` variable is global and used to return extra results from other functions, so programs should not assume `*result*` is valid after subsequent function calls. **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)` [SAL]

`(snd-flatten sound maxlen)` [LISP]

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)` [SAL]

`(snd-length sound maxlen)` [LISP]

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)` [SAL]

`(snd-maxsamp sound)` [LISP]

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 115).

`snd-play(expression)` [SAL]

`(snd-play expression)` [LISP]

Evaluates *expression* to obtain a sound, computes all of the samples (without retaining them in memory), and returns the number of samples computed. 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 write samples to a file, 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. Note that it does not accept multichannel sounds. To time mult-channel sound computation, you might try applying `to-mono` (see Section 7.1.4) to get a SOUND.

`snd-print-tree(sound)` [SAL]

`(snd-print-tree sound)` [LISP]

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)` [SAL]

`(snd-samples sound limit)` [LISP]

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)` [SAL]  
(`snd-srate sound`) [LISP]  
Returns the sample rate of the sound. Safe for ordinary use.
- `snd-time(sound)` [SAL]  
(`snd-time sound`) [LISP]  
Returns the start time of the sound. This will probably go away in a future version, so use `snd-t0` instead.
- `snd-t0(sound)` [SAL]  
(`snd-t0 sound`) [LISP]  
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)` [SAL]  
(`snd-print expression maxlen`) [LISP]  
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)` [SAL]  
(`snd-set-logical-stop sound time`) [LISP]  
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)` [SAL]  
(`snd-sref sound time`) [LISP]  
Evaluates *sound* at the global time given by *time*. Safe for ordinary use, but normally, you should call `sref` instead.
- `snd-stop-time(sound)` [SAL]  
(`snd-stop-time sound`) [LISP]  
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 the programmer has not specified a stop time for the sound. Safe for ordinary use.
- `soundp(sound)` [SAL]  
(`soundp sound`) [LISP]  
Returns true iff *sound* is a SOUND. Safe for ordinary use.
- `stats()` [SAL]  
(`stats`) [LISP]  
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.

`snd-set-max-audio-mem(bytes)` [SAL]

`(snd-set-max-audio-mem bytes)` [LISP]

Sets a limit (a FIXNUM in bytes) on the amount of main memory that Nyquist will allocate for SOUNDS (this does not count table memory). The default is about 1GB. The return value is the previous limit, in bytes. This is not a limit on how big sounds can be. Since Nyquist computes sound incrementally, it can generally play sounds or write sounds to files without storing them in memory. However, if you store large sounds in variables, memory usage can exceed your available RAM, causing extremely slow computation as main memory is swapped to and from disk or Nyquist might run out of memory and crash. The goal of placing a limit on audio memory is to terminate computations before memory is totally exhausted, allowing Nyquist to print an error message and allowing the user to view or save work. Generally, if you see the message “The maximum number of sample blocks has been reached, ...,” you should fix your code to avoid accumulating samples in memory, e.g. do not assign sounds to global variables. Alternatively, you can use this function to increase the limit, but of course you are still limited by the actual size of memory on your computer, and exceeding that could cause severe performance problems in Nyquist and any other applications that are running. See also Section 1.4 for command line options to limit yquist memory, run time, and file access (these options are not available through the NyquistIDE.)

### 7.1.4 Miscellaneous Functions

These are all safe and recommended for ordinary use.

`to-mono(sound)` [SAL]

`(to-mono sound)` [LISP]

Returns the sum of all channels of *sound* if it is a multichannel sound. If *sound* is a SOUND, it is simply returned unchanged. See `sim` (Section 7.4) for more details on how channels are summed.

`db-to-linear(x)` [SAL]

`(db-to-linear x)` [LISP]

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*.

`db-to-vel(x [, float])` [SAL]

`(db-to-vel x [float])` [LISP]

Returns the conversion of *x* from decibels to MIDI velocity using a rule that maps -60 dB to 1 and 0 dB to 127. The MIDI velocity varies linearly with the square root of amplitude. The default value of *float* is `nil` and the result is a FIXNUM clipped to fall in the legal range of 1-127, but if a non-`nil` value is provided, the result is a FLONUM that is not rounded or clipped. The input parameter must be a FIXNUM or FLONUM. Sounds are not allowed.

`follow(sound, floor, risetime, falltime, lookahead)` [SAL]

`(follow sound floor risetime falltime lookahead)` [LISP]

An envelope follower intended as a component 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 through 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, lookahead, risetime, falltime, floor, threshold)` [SAL]

`(gate sound lookahead risetime falltime floor threshold)` [LISP]

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 output 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*. Similarly, the fall is a constant-rate exponential such that a fall from unity to *floor* takes *falltime*.

`noise-gate(sound [, lookahead, risetime, falltime, floor, threshold] [, rms: use-rms, link: link-option])` [SAL]

`(noise-gate sound [lookahead risetime falltime floor threshold] [:rms use-rms :link link-option])` [LISP]

A simple noise gate implementation based on `gate`. All parameters except *snd* are optional and default values are *lookahead*: 0.5, *risetime*: 0.02, *falltime*: 0.5, *floor*: 0.01, *threshold*: 0.01. The keyword parameters `:rms` and `:link` are also optional with default values of *use-rms*: NIL (false) and *link-option*: T (true). The result is the input *snd*, where below-threshold segments of sound are attenuated to a maximum of *floor* (see the `gate` function above). If *use-rms* is non-NIL, the *threshold* applies

to the RMS of the *sound* computed with 10 ms non-overlapping rectangular windows. Otherwise, *threshold* applies to the absolute value of each sample in *sound*. If *link-option* is non-NIL, and if the input *sound* is multichannel, then a single gate is computed and applied to all channels. The gate threshold is considered to be exceeded when *any* channel would exceed the threshold and open the gate. (In other words, whether *use-rms* or not, a maximum value is computed from all the channels and used to control the gate.)

`hz-to-step(freq)` [SAL]

`(hz-to-step freq)` [LISP]

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)` [SAL]

`(linear-to-db x)` [LISP]

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*.

`linear-to-vel(x [, float])` [SAL]

`(linear-to-vel x [float])` [LISP]

Returns the conversion of *x* from linear amplitude to MIDI velocity using a rule that maps -60 dB to 1 and 0 dB to 127. The MIDI velocity varies linearly with the square root of amplitude. The default value of *float* is nil and the result is a FIXNUM clipped to fall in the legal range of 1-127, but if a non-nil value is provided, the result is a FLONUM that is not rounded or clipped. The input parameter must be a FIXNUM or FLONUM. Sounds are not allowed.

`log(x)` [SAL]

`(log x)` [LISP]

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

`set-control-srate(rate)` [SAL]

`(set-control-srate rate)` [LISP]

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)` [SAL]

`(set-sound-srate rate)` [LISP]

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()` [SAL]

`(set-pitch-names)` [LISP]

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)` [SAL]

`(step-to-hz pitch)` [LISP]

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-ioi(dur)` [SAL]

`(get-ioi dur)` [LISP]

Gets the duration of of something starting at a local time of 0 and ending at a local time of *dur*. For convenience, `*rslt*` is set to the global time corresponding to local time zero. Note that `sustain` is ignored in this calculation. The intent is to calculate the nominal start time of the next event in a sequence (or the logical stop time of the current event) rather than the actual sound duration of the current event.

`get-duration(dur)` [SAL]

`(get-duration dur)` [LISP]

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. See also `get-ioi` (above).

`get-loud()` [SAL]

`(get-loud)` [LISP]

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()` [SAL]

`(get-sustain)` [LISP]

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()` [SAL]

`(get-transpose)` [LISP]

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()` [SAL]

`(get-warp)` [LISP]

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)` [SAL]

`(local-to-global local-time)` [LISP]

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

`round(x)` [SAL]

`(round x)` [LISP]

Round  $x$  to the nearest integer. If  $x$  is  $n + 0.5$ , where  $n$  is an integer, then return  $n + 1$ , even if  $n$  is negative.

`snd-set-latency(latency)` [SAL]

`(snd-set-latency latency)` [LISP]

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.

`vel-to-db(x)` [SAL]

`(vel-to-db x)` [LISP]

Returns the conversion of  $x$  from MIDI velocity to decibels using a rule that maps MIDI velocity 1 to -60 dB and 127 to 0 dB. The amplitude is proportional to the square of MIDI velocity. The input  $x$  can be a FIXNUM or FLONUM but not a sound. The result is a FLONUM.

`vel-to-linear(x)` [SAL]

`(vel-to-linear x)` [LISP]

Returns the conversion of  $x$  from MIDI velocity to linear amplitude ratio using a rule that maps MIDI velocity 1 to -60 dB (0.001) and 127 to unity gain. The amplitude is proportional to the square of MIDI velocity. The input  $x$  can be a FIXNUM or FLONUM but not a sound. The result is a FLONUM.

## 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)` [SAL]

`(cue sound)` [LISP]

Applies *\*loud\**, the starting time from *\*warp\**, *\*start\**, and *\*stop\** to *sound*.

`cue-file(filename)` [SAL]

`(cue-file filename)` [LISP]

Same as `cue`, except the sound comes from the named file, samples from which are coerced to the current default `*sound-rate*` sample rate.

`sound(sound)` [SAL]

`(sound sound)` [LISP]

Applies `*loud*`, the starting time and (possibly continuously varying) stretching from `*warp*`, `*start*`, and `*stop*` to `sound`.

`control(sound)` [SAL]

`(control sound)` [LISP]

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])` [SAL]

`(const value [duration])` [LISP]

Creates a constant function at the `*control-rate*`. 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(t1, t2, t4, l1, l2, l3, [dur])` [SAL]

`(env t1 t2 t4 l1 l2 l3 dur)` [LISP]

Creates a 4-phase envelope.  $t_i$  is the duration of phase  $i$ , and  $l_i$  is the final level of phase  $i$ .  $t_3$  is implied by the duration `dur`, and  $l_4$  is 0.0. If `dur` is not supplied, then 1.0 is assumed. The envelope duration is the product of `dur`, `*stretch*`, and `*sustain*`. If  $t_1 + t_2 + 2ms + t_4$  is greater than the envelope duration, then a two-phase envelope is substituted that has an attack/release time ratio of  $t_1/t_4$ . The sample rate of the returned sound is `*control-rate*`. (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)` [SAL]

`(exp-dec hold halfdec length)` [LISP]

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)` [SAL]

`(force-srate srate sound)` [LISP]

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])` [SAL]

`(lfo freq duration table phase)` [LISP]

Just like `osc` (below) except this computes at the `*control-srate*` and frequency is specified in Hz. Initial *phase* is specified in degrees, defaulting to zero. 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])` [SAL]

`(fmlfo freq [table phase])` [LISP]

A low-frequency oscillator that computes at the `*control-srate*` using a sound to specify a time-varying frequency in Hz. Initial *phase* is a FLONUM in degrees. The duration of the result is determined by *freq*.

`maketable(sound)` [SAL]

`(maketable sound)` [LISP]

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 100,000,000 samples. This limit is the compile-time constant `max_table_len` set in `sound.h`. A wavetable 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. Wavetables are used by `osc`, `osc`, `hzosc`, `amosc`, `fmosc`, `lfo`, and `fmlfo`.

`build-harmonic(n, table-size)` [SAL]

`(build-harmonic n table-size)` [LISP]

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 9 for an example. A scaled sum of these harmonics can be passed to `maketable` to construct a wavetable suitable for `osc` and other oscillators.

`control-warp(warp-fn, signal, [wrate])` [SAL]

`(control-warp warp-fn signal wrate)` [LISP]

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(beh1, beh2, ...)` [SAL]

`(mult beh1 beh2 ...)` [LISP]

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(beh1, beh2, ...)` [SAL]

`(prod beh1 beh2 ...)` [LISP]

Same as `mult`.

`pan(sound, where)` [SAL]

`(pan sound where)` [LISP]

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(beh1, beh2, ...)` [SAL]

`(prod beh1 beh2 ...)` [LISP]

Same as `mult`.

`resample(sound, srate)` [SAL]

`(resample sound srate)` [LISP]

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)` [SAL]

`(scale scale sound)` [LISP]

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)` [SAL]

`(scale-db db sound)` [LISP]

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)` [SAL]

`(scale-srate sound scale)` [LISP]

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).

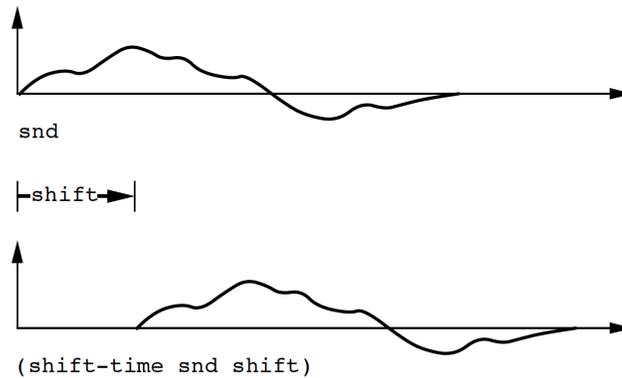


Figure 7.1: The `shift-time` function shifts a sound in time according to its `shift` argument.

```
shift-time(sound, shift) [SAL]
(shift-time sound shift) [LISP]
  Shift sound by shift seconds. If the sound is
```

See Figure 7.1. This is a special case of `snd-xform` (see Section 7.6.2).

```
sound-warp(warp-fn, signal [, wrate]) [SAL]
(sound-warp warp-fn signal [wrate]) [LISP]
```

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)` [SAL]

`(integrate signal)` [LISP]

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

`slope(signal)` [SAL]

`(slope signal)` [LISP]

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])` [SAL]

`(osc pitch [duration table phase])` [LISP]

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:** A *table* is 3-element list. See `maketable` for a detailed description.

**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)` [SAL]

`(partial pitch env)` [LISP]

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-rate\**. The `partial` function is faster than `osc`.

`sine(pitch [, duration])` [SAL]

`(sine pitch [duration])` [LISP]

Returns a sinusoid at the indicated pitch. The sample rate is *\*sound-rate\**. This function is like `osc` with respect to transformations. The `sine` function is faster than `osc`.

`hzosc(hz [, table, phase])` [SAL]

`(hzosc hz [table phase])` [LISP]

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 degrees. 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-rate\**.

`osc-saw(hz)` [SAL]

`(osc-saw hz)` [LISP]

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

`osc-tri(hz)` [SAL]

`(osc-tri hz)` [LISP]

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

`osc-pulse(hz, bias [, compare-shape])` [SAL]

`(osc-pulse hz bias [compare-shape])` [LISP]

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])` [SAL]

`(amosc pitch modulation [table phase])` [LISP]

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-rate\**.

`fmosc(pitch, modulation [, table, phase])` [SAL]

`(fmosc pitch modulation [table phase])` [LISP]

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-rate\**.

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

`(fmfb pitch index [dur])` [LISP]

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-rate\**. A sinusoid table is used. If *index* is below 1.1, this generates a sawtooth-like waveform.

`buzz(n, pitch, modulation)` [SAL]

`(buzz n pitch modulation)` [LISP]

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 *pitch* specifies the fundamental frequency (a number, in steps) assuming no modulation. 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-rate\**.

`pluck(pitch [, duration, final-amplitude])` [SAL]

`(pluck pitch [duration final-amplitude])` [LISP]

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-rate\**.

`siosc(pitch, modulation, tables)` [SAL]

`(siosc pitch modulation tables)` [LISP]

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])` [SAL]

`(sampler pitch modulation [sample npoints])` [LISP]

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 2048-point sinusoid) 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, `pw1` 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 because exponential decays never reach zero. This inability to reach zero can be addressed by a pseudo-exponential function, `pwz`, whose design was borrowed from the `ZynAddSubFx` software synthesizer, where we add a small bias (0.01) to breakpoints, compute the exponential envelope between these new breakpoints, then subtract 0.01 so that the envelope can actually decay all the way to zero. These curves are very close to exponential (within 0.01) until they approach zero, where they become essentially linear. Like `pw1` (linear), the `pwz` envelopes assume an initial point at (0, 0) and a final value of 0. For amplitude envelopes, the `linear-attack` option for the `pwz` variants is suggested for a more natural-sounding envelope.

Each of these three forms has variants. You can specify the initial and final values (instead of taking the default). You can specify time in intervals rather than cumulative time. Finally, you can pass a list rather than an argument list. This leads to 16 versions, as shown in Figure 7.2.

All of these functions are implemented in terms of `pw1` (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.

## 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

## Piece-wise Pseudo-Exponential Functions with Bias:

## Cummulative Time:

Default initial point at (0, 0), final value at 0:

pwz

pwz-list

Explicit initial value:

pwzv

pwzv-list

## Relative Time:

Default initial point at (0, 0), final value at 0:

pwzr

pwzr-list

Explicit initial value:

pwzvr

pwzvr-list

Figure 7.2: Variants of piece-wise functions.

`pwl( $t_1, l_1, t_2, l_2, \dots t_n$ )` [SAL]

`(pwl  $t_1 l_1 t_2 l_2 \dots t_n$ )` [LISP]

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-rate*`. 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( $breakpoints$ )` [SAL]

`(pwl-list  $breakpoints$ )` [LISP]

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( $l_1, t_2, l_2, t_3, l_3, \dots t_n, l_n$ )` [SAL]

`(pwlv  $l_1 t_2 l_2 t_3 l_3 \dots t_n l_n$ )` [LISP]

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( $breakpoints$ )` [SAL]

`(pwlv-list  $breakpoints$ )` [LISP]

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

`pwlr( $i_1, l_1, i_2, l_2, \dots i_n$ )` [SAL]

`(pwlr  $i_1 l_1 i_2 l_2 \dots i_n$ )` [LISP]

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 cumulative time. Otherwise, the behavior is like that of `pwl`.

`pwlr-list( $breakpoints$ )` [SAL]

`(pwlr-list  $breakpoints$ )` [LISP]

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

`pwlvr( $l_1, i_2, l_2, i_3, l_3, \dots i_n, l_n$ )` [SAL]

`(pwlvr  $l_1 i_2 l_2 i_3 l_3 \dots i_n l_n$ )` [LISP]

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 cumulative time. Otherwise, the behavior is like that of `pwlv`.

`pwlvr-list(breakpoints)` [SAL]

`(pwlvr-list breakpoints)` [LISP]

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

`pwe( $t_1, l_1, t_2, l_2, \dots, t_n$ )` [SAL]

`(pwe  $t_1 l_1 t_2 l_2 \dots t_n$ )` [LISP]

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 the `pwz` functions which allow decay to zero, and `pwev` which has explicit initial and final values.*

`pwe-list(breakpoints)` [SAL]

`(pwe-list breakpoints)` [LISP]

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

`pwev( $l_1, t_2, l_2, t_3, l_3, \dots, t_n, l_n$ )` [SAL]

`(pwev  $l_1 t_2 l_2 t_3 l_3 \dots t_n l_n$ )` [LISP]

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)` [SAL]

`(pwev-list breakpoints)` [LISP]

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

`pwer( $i_1, l_1, i_2, l_2, \dots, i_n$ )` [SAL]

`(pwer  $i_1 l_1 i_2 l_2 \dots i_n$ )` [LISP]

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 cumulative time. Otherwise, the behavior is like that of `pwe`. Consider using `pwerv` instead of this one.

`pwer-list(breakpoints)` [SAL]

`(pwer-list breakpoints)` [LISP]

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

`pwerv( $l_1, i_2, l_2, i_3, l_3, \dots, i_n, l_n$ )` [SAL]

`(pwerv  $l_1 i_2 l_2 i_3 l_3 \dots i_n l_n$ )` [LISP]

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 cumulative 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)` [SAL]

`(pwevr-list breakpoints)` [LISP]

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

`pwz( $t_1, l_1, t_2, l_2, \dots, t_n$ , [bias:  $bias$ ], [linear-attack:  $linatk$ ])` [SAL]

`(pwz  $t_1 l_1 t_2 l_2 \dots t_n$  [:bias  $bias$ ] [:linear-attack  $linatk$ ])` [LISP]

Creates a piece-wise pseudo-exponential envelope with breakpoints at  $(0, 0)$ ,  $(t_1, l_1)$ ,  $(t_2, l_2)$ , ...  $(t_n, 0)$ . Each breakpoint value is first incremented by  $bias$ , then interpolated with exponentials as in `pwe` (see above), then  $bias$  is subtracted from all samples so that the envelope passes through the original breakpoints as specified, including decay to zero. Breakpoint values ( $l_j$ ) must be non-negative. 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). The `linear-attack` keyword parameter, if true (default is false), causes a linear interpolation from  $(0, 0)$  to  $(t_1, l_1)$  rather than exponential. The `bias` keyword parameter is optional, and defaults to 0.01 (about -40dB). As  $bias$  grows large, the "half-life" of the decay becomes large, so the decay is more linear. Even large values such as 1 are possible. As  $bias$  grows small, the "half-life" of the decay becomes small, so the decay is more rapid. Example: If  $bias$  is 0.01 (default), a decay from 1 to 0 in 1 second will decay to about -20dB in 0.5 seconds. If  $bias$  is 0.0001 (-80dB), the envelope will decay to about -20dB in only 0.25 seconds. You can think of  $bias$  as approximately how much attenuation you want to occur exponentially before the final decay to zero where the envelope becomes more linear. With  $bias$  values at or below 0.01, the "linear part" occurs when the signal is barely audible anyway.

`pwz-list(breakpoints, [bias:  $bias$ ], [linear-attack:  $linatk$ ])` [SAL]

`(pwz-list breakpoints [:bias  $bias$ ] [:linear-attack  $linatk$ ])` [LISP]

A version of `pwz` that takes a single list of breakpoints as its argument. See `pwl-list` above for the rationale for this "-list" form of `pwz` and for an explanation of the `linear-attack` keyword parameter.

`pwzv( $l_1, t_2, l_2, t_3, t_3, \dots, t_n, l_n$ , [bias:  $bias$ ], [linear-attack:  $linatk$ ])` [SAL]

`(pwzv  $l_1 t_2 l_2 t_3 t_3 \dots t_n l_n$  [:bias  $bias$ ] [:linear-attack  $linatk$ ])` [LISP]

Creates a piece-wise pseudo-exponential envelope with breakpoints at  $(0, l_1)$ ,  $(t_2, l_2)$ , etc., ending with  $(t_n, l_n)$ . If the `linear-attack` parameter is true (default is false), linear interpolation is used from  $(0, l_1)$  to  $(t_2, l_2)$ . Otherwise, the behavior is like that of `pwz`.

`pwzv-list(breakpoints, [bias:  $bias$ ], [linear-attack:  $linatk$ ])` [SAL]

`(pwzv-list breakpoints [:bias  $bias$ ] [:linear-attack  $linatk$ ])` [LISP]

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

`pwzr(i1, l1, i2, l2, ... in, [bias: bias], [linear-attack: linatk])` [SAL]

`(pwzr i1 l1 i2 l2 ... in [:bias bias] [:linear-attack linatk])` [LISP]

Creates a piece-wise pseudo-exponential envelope with breakpoints at (0, 0), (*t*<sub>1</sub>, *l*<sub>1</sub>), (*t*<sub>2</sub>, *l*<sub>2</sub>), ... (*t*<sub>*n*</sub>, 0), where *t*<sub>*j*</sub> is the sum of *i*<sub>1</sub> through *i*<sub>*j*</sub>. In other words, the breakpoint times are specified in terms of intervals rather than cumulative time. Otherwise, the behavior is like that of `pwz`.

`pwzr-list(breakpoints, [bias: bias], [linear-attack: linatk])` [SAL]

`(pwzr-list breakpoints [:bias bias] [:linear-attack linatk])` [LISP]

A version of `pwzr` that takes a single list of breakpoints as its argument. See `pw1-list` above for the rationale for this “-list” form of `pwzr`.

`pwzvr(l1, i2, l2, i3, i3, ... in, ln, [bias: bias], [linear-attack: linatk])` [SAL]

`(pwzvr l1 i2 l2 i3 i3 ... in ln [:bias bias] [:linear-attack linatk])` [LISP]

Creates a piece-wise exponential envelope with breakpoints at (0, *l*<sub>1</sub>), (*t*<sub>2</sub>, *l*<sub>2</sub>), etc., ending with (*t*<sub>*n*</sub>, *l*<sub>*n*</sub>), where *t*<sub>*j*</sub> is the sum of *i*<sub>2</sub> through *i*<sub>*j*</sub>. In other words, the breakpoint times are specified in terms of intervals rather than cumulative time. Otherwise, the behavior is like that of `pwzv`. See `pwz` (above) for an explanation of *bias* and *linatk*.

`pwzvr-list(breakpoints, [bias: bias], [linear-attack: linatk])` [SAL]

`(pwzvr-list breakpoints [:bias bias] [:linear-attack linatk])` [LISP]

A version of `pwzvr` that takes a single list of breakpoints as its argument. See `pw1-list` above for the rationale for this “-list” form of `pwzvr`.

### 7.2.2.3 Filter Behaviors

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

`(alpass sound decay hz [minhz])` [LISP]

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)` [SAL]

`(comb sound decay hz)` [LISP]

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)` [SAL]

`(congen gate risetime falltime)` [LISP]

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 `pw1` to build a pulse followed by some zero time:

```
(pw1 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 (pw1 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 `pw1` 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)` [SAL]

`(convolve sound response)` [LISP]

Convolve two signals. The first can be any length, but the total space required is proportional to the length of *response*. The start time, logical stop time, and sample rate of the output match those of the input *sound*. The physical stop time of the result is the physical stop time of *sound* plus the duration of the *response* so that the result sound includes the “tail” of the filter response. The response is linearly interpolated if necessary to have same sample rate as *sound*. The current implementation uses a “fast convolution” algorithm with a maximum FFT size of 64K, which after zero padding allows up to 32K point convolutions. If the impulse response is longer, it is broken into multiple blocks of 32K samples. There is no limit on the length of the impulse response. This is an Order(N x M) algorithm where N and M are the number of 32K sample blocks in the *sound* and *response*, respectively. Further discussion and examples can be found in `nyquist/lib/convolve/convolution.html`.

`feedback-delay(sound, delay, feedback)` [SAL]

`(feedback-delay sound delay feedback)` [LISP]

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 amount 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)` [SAL]

`(lp sound cutoff)` [LISP]

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)` [SAL]

`(tone sound cutoff)` [LISP]

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

`hp(sound, cutoff)` [SAL]

`(hp sound cutoff)` [LISP]

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)` [SAL]

`(atone sound cutoff)` [LISP]

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

`reson(sound, center, bandwidth [, n])` [SAL]

`(reson sound center bandwidth [n])` [LISP]

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 [nyquist/lib/voice/voice\\_synthesis.html](http://nyquist/lib/voice/voice_synthesis.html) for sample code and documentation.

`areson(sound, center, bandwidth [, n])` [SAL]

`(areson sound center bandwidth [n])` [LISP]

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 yields the original signal.

`shape(signal, table, origin)` [SAL]

`(shape signal table origin)` [LISP]

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 `signals` and `tables`.

Further discussion and examples can be found by installing the “distortion” extension with the NyquistIDE Extension Manager. The code will appear in [nyquist/lib/distortion/distortion.html](http://nyquist/lib/distortion/distortion.html). The shape function is also used to map frequency to amplitude to achieve a spectral envelope for Shepard tones (install the “shepard” extension) in [nyquist/lib/shepard/shepard.lsp](http://nyquist/lib/shepard/shepard.lsp).

`biquad(signal, b0, b1, b2, a0, a1, a2)` [SAL]

`(biquad signal b0 b1 b2 a0 a1 a2)` [LISP]

A fixed-parameter biquad filter. All filter coefficients are FLONUMs. 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)` [SAL]

`(biquad-m signal b0 b1 b2 a0 a1 a2)` [LISP]

A fixed-parameter biquad filter with Matlab sign conventions for `a0`, `a1`, and `a2`. All filter coefficients are FLONUMs.

`lowpass2(signal, hz [, q])` [SAL]

`(lowpass2 signal hz [q])` [LISP]

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])` [SAL]

`(highpass2 signal hz [q])` [LISP]

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])` [SAL]

`(bandpass2 signal hz [q])` [LISP]

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])` [SAL]

`(notch2 signal hz [q])` [LISP]

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])` [SAL]

`(allpass2 signal hz [q])` [LISP]

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])` [SAL]

`(eq-lowshelf signal hz gain [slope])` [LISP]

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])` [SAL]

`(eq-highshelf signal hz gain [slope])` [LISP]

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)` [SAL]

`(eq-band signal hz gain width)` [LISP]

A fixed- or variable-parameter, second-order midrange equalization (EQ) filter based on `snd-biquad`, `eq-band-ccc` and `eq-band-vvv`. 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 SOUNDS, 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)` [SAL]

`(lowpass4 signal hz)` [LISP]

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

`lowpass6(signal, hz)` [SAL]

`(lowpass6 signal hz)` [LISP]

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

`lowpass8(signal, hz)` [SAL]

`(lowpass8 signal hz)` [LISP]

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

`highpass4(signal, hz)` [SAL]

`(highpass4 signal hz)` [LISP]

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

`highpass6(signal, hz)` [SAL]

`(highpass6 signal hz)` [LISP]

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

`highpass8(signal, hz)` [SAL]

`(highpass8 signal hz)` [LISP]

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

`tapv(sound, offset, vardelay, maxdelay)` [SAL]

`(tapv sound offset vardelay maxdelay)` [LISP]

A delay line with a variable position tap. Identical to `snd-tapv`. See it for details (page 113).

#### 7.2.2.4 Effects

`nrev(sound, decay, mix)` [SAL]

`(nrev sound decay mix)` [LISP]

`jcrev(sound, decay, mix)` [SAL]

`(jcrev sound decay mix)` [LISP]

`prcrev(sound, decay, mix)` [SAL]

`(prcrev sound decay mix)` [LISP]

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 multichannel. 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])` [SAL]

`(stkchorus sound depth freq mix [delay])` [LISP]

Chorus implemented in STK. The input *sound* can be single or multichannel. 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)` [SAL]

`(pitshift sound shift mix)` [LISP]

A pitch shifter implemented in STK. The input *sound*, a single-channel or multichannel 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)` [SAL]

`(clarinet step breath-env)` [LISP]

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)` [SAL]

`(clarinet-freq step breath-env freq-env)` [LISP]

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*) [SAL]

(`clarinet-all` *step* *breath-env* *freq-env* *vibrato-freq* *vibrato-gain*  
*reed-stiffness* *noise*) [LISP]

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*) [SAL]

(`sax` *step* *breath-env*) [LISP]

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*) [SAL]

(`sax-freq` *step* *breath-env* *freq-env*) [LISP]

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*) [SAL]

(`sax-all` *step* *breath-env* *freq-env* *vibrato-freq* *vibrato-gain*  
*reed-stiffness* *noise* *blow-pos* *reed-table-offset*) [LISP]

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)` [SAL]

`(flute step breath-env)` [LISP]

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)` [SAL]

`(flute-freq step breath-env freq-env)` [LISP]

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 arbitrarily set to 30.

`flute-all(step, breath-env, freq-env, vibrato-freq, vibrato-gain, jet-delay, noise)` [SAL]

`(flute-all step breath-env freq-env vibrato-freq vibrato-gain jet-delay noise)` [LISP]

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)` [SAL]

`(bowed step bowpress-env)` [LISP]

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)` [SAL]

`(bowed-freq step bowpress-env freq-env)` [LISP]

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)` [SAL]

`(mandolin step dur detune)` [LISP]

A physical model of a plucked double-string instrument from STK. The *step* parameter is a FLONUM which 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)` [SAL]

`(wg-uniform-bar step bowpress-env)` [LISP]

`wg-tuned-bar(step, bowpress-env)` [SAL]

`(wg-tuned-bar step bowpress-env)` [LISP]

`wg-glass-harm(step, bowpress-env)` [SAL]

`(wg-glass-harm step bowpress-env)` [LISP]

`wg-tibetan-bowl(step, bowpress-env)` [SAL]

`(wg-tibetan-bowl step bowpress-env)` [LISP]

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 seem to influence the timbral quality of the resulting sound).

`modalbar(preset, step, dur)` [SAL]

`(modalbar preset step dur)` [LISP]

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)` [SAL]

`(sitar step dur)` [LISP]

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

`stk-breath-env(dur, note-on note-off)` [SAL]

`(stk-breath-env dur note-on note-off)` [LISP]

A simple envelope function intended for STK instruments such as CLARINET, where *dur* is the duration, *note-on* is the attack time, and *note-off* is the decay time, all FLONUMs in seconds.

### 7.2.2.6 Phase Vocoder

`phasevocoder(s, map, [fftsize, hopsize, mode])` [SAL]

`(phasevocoder s map [fftsize hopsize mode])` [LISP]

Phase vocoder: the input SOUND *s* is stretched in time according to the control signal (a SOUND) *map*. The *map* parameter must be strictly non-decreasing. It specifies the input time for each time in the output. E.g. if *map* at time 3 is 4, then the output at time 3 will sound like the input at time 4, assuming *s* and *map* start at time 0. In the general case, *s* and *map* must start at the same time. The phase vocoder operation is applied as if *s* and *map* started at time 0, and the result is shifted to start at the same starting time as *s* and *map*. The *fftsize* is a FIXNUM and power of 2, indicating the size of the analysis and synthesis windows in samples. The default is 2048. The *hopsize* is the hop size (in samples, a FIXNUM) of the synthesis stage where overlapping windows are added. The *hopsize* should be a power of 2 that is smaller than the *fftsize*. The default is the *fftsize*/8. Note that the analysis hopsize is determined by the *map*. If the playback speed (slope of *map*) is too great, the analysis hopsize will be large and the sound quality will suffer. You can always make the analysis hopsize smaller by making the synthesis hopsize smaller (at the cost of greater computation). The *mode* is a FIXNUM that defaults to 0, meaning a standard phase vocoder. A value of 1 invokes a phase computation that attempts to reduce phase artifacts by preserving the phase relationships between peaks and nearby bins. A value of 2 invokes a “robot voice” mode that assigns fixed phases and creates a constant pitch (controlled by the *hopsize*) vocoder-like effect. (Thanks to M.C.Sharma for this idea.) A phase vocoder constructs output from overlapped and added windowed IFFT frames. This presents a special problem at the beginning and ending of the output sound. To output a sound starting at time *t*, the first frame is centered at  $t + \text{fftsize}/2$ , so the output at *t* will smoothly rise from zero because all frames are smoothed by a windowing function. Considerations are: (1) The output at time *t* will rise from zero rather than being the sum of multiple overlapping windows; (2) if *t* maps to an input window that begins before the start time of the input sound, the input window will be padded with zeros and may include an abrupt onset if one exists at the start of the input sound.

`pv-time-pitch(s, stretchfn, pitchfn, dur, [fftsize, hopsize, mode])` [SAL]

`(pv-time-pitch s stretchfn pitchfn dur [fftsize hopsize mode])` [LISP]

Combines phase vocoder and resampling to perform simultaneous time stretching and pitch shifting. The input SOUND *s* is stretched according to SOUND *stretchfn* and pitch-shifted according to SOUND *pitchfn*. The approximate output duration should be specified by *dur* (which is used to optimize the sample rate of computed control functions), and the remaining parameters *fftsize*, *hopsize* and *mode* are passed to the phase vocoder (see `phasevocoder` above). The *stretchfn* gives the factor by which the input should be stretched at each point in time; thus, the total duration is the *integral* of this function. The *pitchfn* specifies the amount by which pitch should be shifted at each point in time. For example, where *pitchfn* is 2, the sample rate will be doubled, increasing pitch and frequencies by an octave. The phase vocoder is used to compensate for time stretching caused by resampling, so *stretchfn* and *pitchfn* operate independently.

### 7.2.2.7 More Behaviors

`clip(sound, peak)` [SAL]

`(clip sound peak)` [LISP]

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*. **Note:** Many systems clip output when converting to fixed-point audio, e.g. 16-bit samples. Instead Nyquist simply takes the low-order 16 bits, allowing samples to “wrap around.” This sounds *terrible*, but that is the point: It is hard to miss when your samples go out of range. One use of this function is to clip rather than wrap output. **Warning:** A floating point sample value of 1.0 maps to  $2^{15}$  in 16-bit audio, but the maximum 16-bit sample value is  $2^{15} - 1$ ! If your goal is to clip to the 16-bit range, you should set *peak* to the ratio 32767.0/32768.0. For 24-bit audio, use  $(2^{23} - 1)/(2^{23}$ , etc.

`s-abs(sound)` [SAL]

`(s-abs sound)` [LISP]

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-avg(sound, blocksize, stepsize, operation)` [SAL]

`(s-avg sound blocksize stepsize operation)` [LISP]

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*. The duration of the output is the same (approximately, due to rounding) as that of *sound*. Notice however, that the features of the input will appear earlier in the output by half the window size. For example, a sharp peak in the input will result in a smoothed peak (using `OP-AVERAGE`) one half *blocksize* earlier. You can correct for this shift by inserting one half *blocksize* of silence before *sound*, e.g. if *s* has a sample rate of 44100 Hz, then `snd-avg(seq(s-rest(0.01), cue(s)), 882, 441, OP-AVERAGE)` will shift *s* by 0.01 s to compensate for the shift introduced by a smoothing window of size 0.02 s ( $882/44100$ ). If *sound* is a multichannel sound, return a multichannel sound with `s-avg` applied to each element. 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 [7.2.2.7](#).

`s-sqrt(sound)` [SAL]

`(s-sqrt sound)` [LISP]

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)` [SAL]

`(s-exp sound)` [LISP]

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)` [SAL]

`(s-log sound)` [LISP]

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)` [SAL]

`(s-max sound1 sound2)` [LISP]

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)` [SAL]

`(s-min sound1 sound2)` [LISP]

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])` [SAL]

`(osc-note pitch [duration env loud table])` [LISP]

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)` [SAL]

`(quantize sound steps)` [LISP]

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 multichannel.

`ramp([duration])` [SAL]

`(ramp [duration])` [LISP]

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 7.3 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])` [SAL]

`(rms sound [rate window-size])` [LISP]

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 *sound* is a SOUND or a multichannel sound (in which case the result is a multichannel sound). The *rate* is a FLONUM and *window-size* is a FIXNUM.

`recip(sound)` [SAL]

`(recip sound)` [LISP]

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])` [SAL]

`(s-rest [duration])` [LISP]

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])` [SAL]

`(noise duration)` [LISP]

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\*.

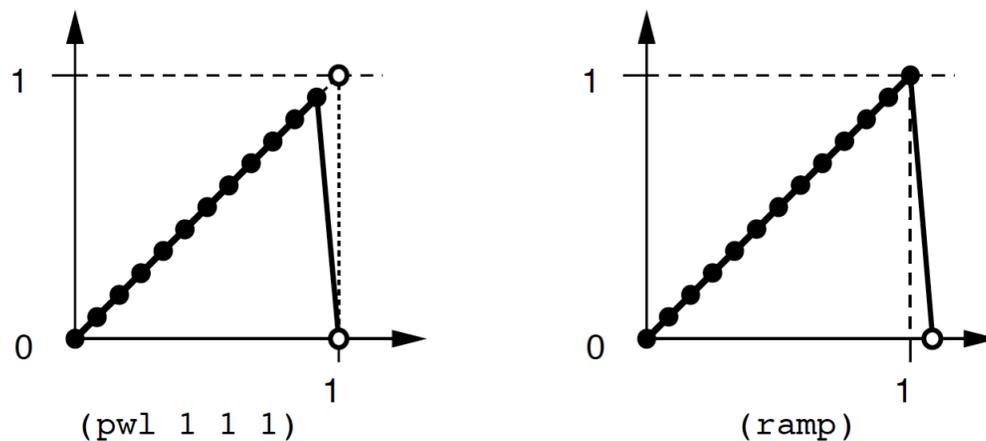


Figure 7.3: Ramps generated by `pwl` and `ramp` functions. The `pwl` version ramps toward the breakpoint (1, 1), but in order to ramp back to zero at breakpoint (1, 0), the function never reaches an amplitude of 1. If used at the beginning of a `seq` construct, the next sound will begin at time 1. The `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.

`yin(sound, minstep, maxstep, stepsize)` [SAL]

`(yin sound minstep maxstep stepsize)` [LISP]

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 (in units of steps, i.e. middle C = 60). 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)` [SAL]

`(abs-env beh)` [LISP]

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)` [SAL]

`(at time beh)` [LISP]

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)` [SAL]

`(at-abs time beh)` [LISP]

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)` [SAL]

`(continuous-control-warp beh)` [LISP]

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)` [SAL]

`(continuous-sound-warp beh)` [LISP]

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)` [SAL]

`(control-srate-abs srate beh)` [LISP]

Evaluate *beh* with *\*control-srate\** set to sample rate *srate*. **Note:** there is no “relative” version of this function.

`extract(start, stop, beh)` [SAL]

`(extract start stop beh)` [LISP]

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)` [SAL]

`(extract-abs start stop beh)` [LISP]

Returns a sound which is the portion of *beh* between *start* and *stop*, independent of the current *\*warp\**. The result is shifted to start at time zero. Wrapping a call to `extract-abs` in *cue* will shift the result to start at the current time, which is `local-to-global(0)`.

`loud(volume, beh)` [SAL]

`(loud volume beh)` [LISP]

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

`loud-abs(volume, beh)` [SAL]

`(loud-abs volume beh)` [LISP]

Evaluates *beh* with *\*loud\** set to *volume*.

`sound-srate-abs(srate, beh)` [SAL]

`(sound-srate-abs srate beh)` [LISP]

Evaluate *beh* with *\*sound-srate\** set to sample rate *srate*. **Note:** there is no “relative” version of this function.

`stretch(factor, beh)` [SAL]

`(stretch factor beh)` [LISP]

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)` [SAL]

`(stretch-abs factor beh)` [LISP]

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)` [SAL]

`(sustain factor beh)` [LISP]

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)` [SAL]

`(sustain-abs factor beh)` [LISP]

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

`transpose(amount, beh)` [SAL]

`(transpose amount beh)` [LISP]

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

`transpose-abs(amount, beh)` [SAL]

`(transpose-abs amount beh)` [LISP]

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)` [SAL]

`(warp fn beh)` [LISP]

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)` [SAL]

`(warp-abs fn beh)` [LISP]

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. See also the `trigger` function, described in Section 9.1.4, which uses a `SOUND` to trigger instances of a behavior.

`seq(beh1 [, beh2, ...])` [SAL]

`(seq beh1 [beh2 ...])` [LISP]

Evaluates the first behavior *beh<sub>1</sub>* according to the current time in the environment 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.

Nyquist programs should generally avoid storing sounds in global variables because that tends to cause sound samples to be retained in memory. A related problem can occur with `seq` and `seqrep`. With these functions, behaviors are not evaluated until needed, but what if the behaviors depend on local variables as parameters? Ordinarily, the variables would be long gone and garbage collected by the time the behavior is evaluated. So, `seq` and `seqrep` capture all local variables and their bindings (values) in *closures*. This solves the problem of retaining local variables and values until they are needed by the behaviors, but it has a problem similar to global variables in that *any sounds captured in the closure cannot be released until the last behavior in the sequence is evaluated*.

Here is an example:

```
load "reverb"

function long-tone() return buzz(15, c4, lfo(5, 10))

function rev-with-tail(s, rt)
  return reverb(seq(s, s-rest(rt * 2)), rt)

play rev-with-tail(pluck(c4), 10)
```

This SAL function uses `seq` to append `rt * 2` seconds of silence to sound `s` before passing it to `reverb` ensuring that the reverb tail will not be cut off immediately at the end of `s`. Behavior 2, `s-rest(rt * 2)`, is wrapped in a closure which also captures the binding of `s` to the input sound (returned from `long-tone()`) and that of `rt` to 10. If there were no closure, then when `rev-with-tail` returns a sound to play, the variable `s` would be freed by garbage collection, and the only remaining reference to `long-tone()` would be internal to the `reverb` sound computation. `reverb` disposes of samples as soon as they are computed and used, so the total memory requirements would be minimal. In this case, however, since `s` is captured in a closure, as `reverb` demands computation of samples from `long-tone()`, `reverb` releases its claim on the samples, but `s` does not, so all the samples are retained in memory until `s-rest(rt * 2)` is evaluated and the closure (and `s`) are freed.

For 10 seconds of sound (such as `long-tone()`), this is not a big problem. In fact, if you use the

default settings for `autonorm`, Nyquist will store about 20 seconds of sound in memory anyway just to do some look-ahead for normalization, but if `s` represents an hour-long stereo recording, or if there are many other sounds bound in local variables and captured in closures, the memory overhead can be too great. The solution is to override the default scopes and bindings to achieve the variable lifetimes we want. At any time, we can set a variable to `nil`, freeing the previous value to allow garbage collection. In LISP, we can write `(prog1 s (setf s nil))` to evaluate `s`, then free `s` by setting it to `nil`. The original value of `s` is returned from the `prog1` expression. In SAL, we can use the same trick, using `setf` rather than the typical SAL assignment operator (`=`).

To change the example to be memory efficient, we change the definition of `rev-with-tail` to:

```
function rev-with-tail(s, rt)
  return reverb(seq(prog1(s, setf(s, nil)), s-rest(rt * 2)), rt)
```

Note in this example that the variable `s` loses its reference to the `long-tone()` sound right after it is evaluated by `prog1`, so now `reverb` has the only reference to this sound, and it can free samples from `long-tone()` as they are consumed. (In case you are wondering, the actual mechanism is that when `reverb` frees samples, their reference count goes to zero since no other reference exists to the possibly shared list of samples. There is no other reference because the reference previously bound to `s` is released by the garbage collector.)

It should be noted that the variable `rt` is retained in the closure and remains accessible there, so when `s` terminates, `s-rest(rt)` will evaluate as expected, using the value that `rt` acquired way back when `rev-with-tail` was called. This illustrates why the local variables must be saved in closures at the time `seq` is called.

`seqrep(var, limit, beh)` [SAL]

`(seqrep (var limit) beh)` [LISP]

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([beh1, beh2, ...])` [SAL]

`(sim [beh1 beh2 ...])` [LISP]

Returns a sound which is the sum of the given behaviors evaluated with the 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 as many channels as the argument with the most channels. 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 LISP `+` 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. Notice that `sim` is not transitive due to coercion rules: Using SAL syntax, `sim(a, sim(b, c))` may not produce the same result as `sim(sim(a, b), c)`.

`simrep(var, limit, beh)` [SAL]

`(simrep var limit beh)` [LISP]

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)`.

`set-logical-stop(beh, time)` [SAL]

`(set-logical-stop beh time)` [LISP]

Returns a sound with *time* as the logical stop time.

`sum(a [, b, ...])` [SAL]

`(sum a [b ...])` [LISP]

Returns the sum of *a*, *b*, ..., allowing mixed addition of sounds, multichannel sounds and numbers. Identical to *sim*. In SAL, use the infix “+” operator. See `sim` just above for more detail.

`mult(a [, b, ...])` [SAL]

`(mult a [b ...])` [LISP]

Returns the product of *a*, *b*, ..., allowing mixed multiplication of sounds, multichannel sounds and numbers.

`diff(a, b)` [SAL]

`(diff a b)` [LISP]

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

`timed-seq(score)` [SAL]

`(timed-seq score)` [LISP]

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* [SAL]

(play *sound*) [LISP]

Play the sound through the DAC. In XLISP, it is a function. In SAL, you can invoke play as either a command or a function:

```
play pluck(c4)
;; or call the function #play like this:
exec #play(pluck(c4))
```

The play command or function writes a file and plays it. The *sound* is any expression that evaluates to a SOUND. Typically, this should be function call, in which case the samples are computed incrementally and not retained in main memory (an advantage for large sounds). If the expression is a variable containing a SOUND (which may or may not be fully evaluated yet), the SOUND is fully evaluated and all samples are retained in the variable. The play function is defined in the file `system.lsp`. The variable `*default-sf-dir*` names a directory into which to save a sound file, and `*default-sound-file*` names the file. If `*default-sound-file*` contains a slash (/), it is treated as an absolute path or path relative to the current directory, ignoring `*default-sf-dir*`. 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 (using SAL syntax):

```
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 (using SAL syntax):

```
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 (or clicking the Mark button in the NyquistIDE) 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)` [SAL]

`(play-file filename)` [LISP]

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()` [SAL]

`(autonorm-on)` [LISP]

Enable automatic adjustment of a scale factor applied to sounds computed using the `play` command.

`autonorm-off()` [SAL]

`(autonorm-off)` [LISP]

Disable automatic adjustment of a scale factor applied to sounds computed using the `play` command.

`sound-on()` [SAL]

`(sound-on)` [LISP]

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

`sound-off()` [SAL]

`(sound-off)` [LISP]

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

`s-save(expression, [maxlen, filename, progress], format: format, mode: mode, bits: bits, swap: flag, play: play)` [SAL]

`(s-save expression [maxlen filename progress] :format format :mode mode :bits bits :swap flag :play play)` [LISP]

Evaluates the *expression*, which should result in a sound or an array of sounds, and writes the result to the given *filename*. (If omitted, `*default-sound-file*` is used instead.) 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*: [SAL] or *:play* [LISP] 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 optionally given by *maxlen*, which allows writing the initial part of a very long or infinite sound. Progress is indicated by printing the sample count after writing each 10 seconds of frames. If *progress* is specified and greater than 10,000, progress is printed at this specified frame count increment. A header is written according to *format*, samples are encoded according to *mode*, using *bits* bits/sample, and bytes are swapped if *flag* is not NIL. Defaults for these are `*default-sf-format*`, `*default-sf-mode*`, and

`*default-sf-bits*`. The default for *flag* 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-WaveX`

Microsoft Wave with WAVEFORMATEX format header.

`snd-head-flac`

FLAC lossless compressed audio.

`snd-head-ogg`

OGG-VORBIS compressed audio.

`snd-head-*`

See `sndfnint.lsp` in the `nyquist/runtime` directory for more formats. The current list includes `paf`, `svx`, `nist`, `voc`, `w64`, `mat4`, `mat5`, `pvf`, `xi`, `htk`, `sds`, `avr`, `sd2`, and `caf`.

## 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` in the `nyquist/runtime` for more modes. The current list includes `double`, `gsm610`, `dwww`, `dpcm`, and `msadpcm`.

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) [SAL]
```

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

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:

- `offset` – the amount of time (in seconds) to skip from the beginning of the file. The default is 0.0.
- `sr` – 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.
- *chans* – 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\**.
- *n* – the number of bits per sample. See *s-save* for details. Default is *\*default-sf-bits\**.
- *flag* – (T or NIL) swap byte order of each sample. Default is NIL.

If there is an error, for example if *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\**<sup>2</sup>. The list assigned to *\*rslt\** is of the form: (*format channels mode bits swap samplerate duration flags*), which are defined as follows:

- *format* – the header format. See *s-save* for details. Access this element of *\*rslt\** by calling `snd-read-format(*rslt*)`.
- *channels* – the number of channels. Access this element of *\*rslt\** by calling `snd-read-channels(*rslt*)`.
- *mode* – the sample representation, e.g. PCM or float. Access this element of *\*rslt\** by calling `snd-read-mode(*rslt*)`. See *s-save* for details.
- *bits* – the number of bits per sample. Access this element of *\*rslt\** by calling `snd-read-bits(*rslt*)`.
- *swap* – 1 if byte-swapping is needed, 0 otherwise. Access this element of *\*rslt\** by calling `snd-read-swap(*rslt*)`.
- *samplerate* – the sample rate, expressed as a FLONUM. Access this element of *\*rslt\** by calling `snd-read-srate(*rslt*)`.
- *duration* – the duration of the sound, in seconds. Access this element of *\*rslt\** by calling `snd-read-dur(*rslt*)`.
- *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-type` (*format*), `snd-head-channels`, `snd-head-mode`, `snd-head-bits`, `snd-head-srate`, and `snd-head-dur`. For example,
 

```
(let ((flags (s-read-flags *rslt*)))
    (not (zerop (logand flags snd-head-srate))))
```

 tells whether the sample rate was specified in the file. See also *sf-info* below. Access this element of *\*rslt\** by calling `snd-read-flags(*rslt*)`.

<sup>2</sup>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.

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

`(s-add-to expression maxlen filename [offset progress])` [LISP]

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). Progress is indicated by printing the sample count after writing each 10 seconds of frames. If *progress* is specified and greater than 10,000, progress is printed at this specified frame count increment. 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, progress])` [SAL]

`(s-overwrite expression maxlen filename [offset progress])` [LISP]

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.) Progress is indicated by printing the sample count after writing each 10 seconds of frames. If *progress* is specified and greater than 10,000, progress is printed at this specified frame count increment. The file must be a recognized sound file with a header (not a raw sound file).

`sf-info(filename)` [SAL]

`(sf-info filename)` [LISP]

Prints information about a sound file. The parameter *filename* is a string. The file is assumed to 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)` [SAL]

`(soundfilename name)` [LISP]

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` to translate filenames.

`s-plot(sound [, dur, n])` [SAL]

`(s-plot sound [dur n])` [LISP]

Plots sound in a window. This function was designed to run a plot program on a Unix workstation, but now is primarily used with NyquistIDE, which has self-contained plotting. Normally, time/value pairs in ascii are written to `points.dat` and system-dependent code (or the NyquistIDE 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*`:

`*default-plot-file*`

The file containing the data points, defaults to "points.dat".

`spec-plot(sound [, offset: offset, dur: dur, res: res, bw: bw, db: db])` [SAL]

`(spec-plot sound [:offset offset :dur dur :res res :bw bw :db db])` [LISP]

Computes and plots the short-time magnitude spectrum of *sound* (a SOUND or STRING). If *sound* is a filename (a STRING), a sound is read from the file. The bandwidth of each bin is given by the *res* (resolution) keyword parameter, which defaults to `*spec-plot-res*`. Since one is often interested in lower frequencies, the *bw* (bandwidth keyword parameter limits the range of bins which are plotted, and defaults to `*spec-plot-bw*`. The output can be presented on a dB scale by setting the *db* keyword parameter to `t` (true). The default is to use a linear scale. The *offset* (in seconds) skips initial samples before taking a frame of samples from *sound*. The *dur* keyword parameter (in seconds) limits the number of samples taken from *sound*. If *dur* is shorter than the FFT window size, the sound is zero-padded. If *dur* is longer than the FFT window size, the extra samples are ignored. The magnitude spectrum is sent to `s-plot` as a signal. The sample rate of the signal is set so that the plot labels on the horizontal axis represent Hz (not bin numbers.) To align bins with grid lines, one normally specifies *res* values in round numbers, e.g. 10 or 20. To achieve an arbitrary bin size, `spec-plot` resamples *sound* to a carefully computed sample rate that, after a power-of-2-sized FFT, yields the desired bin size.

`spec-print(file sound [, offset: offset, dur: dur, res: res, bw: bw, threshold: threshold])` [SAL]

`(spec-print file sound [:offset offset :dur dur :res res :bw bw :threshold threshold])` [LISP]

Computes and prints an ASCII plot of the short-time magnitude spectrum of *sound* using `sa-print` (see 10.1). The *file* can be `T` to print to the console or a file open for writing. Other parameters are as in `spec-plot` (see above) except for *threshold*, which is passed to `sa-print`. Note that *bw* is passed to `sa-print` as *cutoff*. The global variables `*spec-plot-res*` and `*spec-plot-bw*` apply to `spec-print` as well as `spec-plot`. Important: The printed and plotted magnitudes are *normalized* to a peak value of 1 (see `sa-normalize`).

`*spec-plot-res*`

The default bin resolution (separation in Hz between adjacent bins) for `spec-plot`. Defaults to 20 Hz. You can override this by using a keyword parameter when you call `spec-plot` or `spec-print`, or for convenience, you can change this variable which will affect all future calls to `spec-plot` where the keyword parameter is omitted.

**\*spec-plot-bw\***

The default highest frequency to include in `spec-plot`. Defaults to 8000 Hz. You can override this by using a keyword parameter when you call `spec-plot` or `spec-print`, or for convenience, you can change this variable which will affect all future calls to `spec-plot` where the keyword parameter is omitted.

**\*spec-plot-db\***

The default output from `spec-plot` displays magnitude on a linear scale, but there is an option to display on a dB scale. You can change the default behavior by setting this variable to `t` (true), or you can override the default in any call to `spec-plot` using a keyword parameter.

`s-print-tree(sound)` [SAL]

`(s-print-tree sound)` [LISP]

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, sr, duration)` [SAL]

`(snd-const value t0 sr duration)` [LISP]

Returns a sound with constant *value*, starting at *t0* with the given *duration*, at the sample rate *sr*. You might want to use `pw1` (see Section 7.2.2.2) instead.

`snd-read(filename, offset, t0, format, channels, mode, bits, swap, sr, dur)` [SAL]

`(snd-read filename offset t0 format channels mode bits swap sr dur)` [LISP]

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, progress) [SAL]
```

```
(snd-save expression maxlen filename format mode bits swap play progress) [LISP]
```

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. Progress is indicated by printing the sample count after writing each 10 seconds of frames. If *progress* is specified and greater than 10,000, progress is printed at this specified frame count increment. 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, progress) [SAL]
```

```
(snd-overwrite expression maxlen filename offset progress) [LISP]
```

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. Progress is indicated by printing the sample count after writing each 10 seconds of frames. If *progress* is specified and greater than 10,000, progress is printed at this specified frame count increment. 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. 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)` [SAL]

`(snd-coterm s1 s2)` [LISP]

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 ...)` [SAL]

`(snd-from-array ...)` [LISP]

See page 63.

`snd-white(t0, sr, d)` [SAL]

`(snd-white t0 sr d)` [LISP]

Generate white noise, starting at *t0*, with sample rate *sr*, and duration *d*. You probably want to use `noise` (see Section 7.2.2.7).

`snd-zero(t0, sr)` [SAL]

`(snd-zero t0 sr)` [LISP]

Creates a sound that is zero everywhere, starts at *t0*, and has sample rate *sr*. The logical stop time is immediate, i.e. also at *t0*. You probably want to use `pw1` (see Section 7.2.2.2) instead.

## 7.6.2 Signal Operations

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

`snd-abs(sound)` [SAL]

`(snd-abs sound)` [LISP]

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 7.2.2.7.)

`snd-sqrt(sound)` [SAL]

`(snd-sqrt sound)` [LISP]

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 7.2.2.7.)

`snd-add(sound1, sound)` [SAL]

`(snd-add sound1 sound)` [LISP]

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)` [SAL]

`(snd-offset sound offset)` [LISP]

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)` [SAL]

`(snd-avg sound blocksize stepsize operation)` [LISP]

Use `s-avg` instead (see Section 7.2.2.7). The `s-avg` function extends `snd-avg` to multichannel input sounds.

`snd-clip(sound, peak)` [SAL]

`(snd-clip sound peak)` [LISP]

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 7.2.2.7).

`snd-compose(f, g)` [SAL]

`(snd-compose f g)` [LISP]

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 [nyquist/demos/pitch\\_change.htm](http://nyquist/demos/pitch_change.htm).

`snd-tapv(sound, offset, vardelay, maxdelay)` [SAL]

`(snd-tapv sound offset vardelay maxdelay)` [LISP]

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*) [SAL]

(`snd-tapf` *sound* *offset* *vardelay* *maxdelay*) [LISP]

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*) [SAL]

(`snd-copy` *sound*) [LISP]

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*) [SAL]

(`snd-down` *srate* *sound*) [LISP]

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*) [SAL]

(`snd-exp` *sound*) [LISP]

Compute the exponential of each sample of *sound*. Use `s-exp` instead (see Section 7.2.2.7).

`snd-follow`(*sound*, *floor*, *risetime*, *falltime*, *lookahead*) [SAL]

(`snd-follow` *sound* *floor* *risetime* *falltime* *lookahead*) [LISP]

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 samples) 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 through 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)` [SAL]

`(snd-gate sound lookahead risetime falltime floor threshold)` [LISP]

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)` [SAL]

`(snd-inverse signal start srate)` [LISP]

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)` [SAL]

`(snd-log sound)` [LISP]

Compute the natural logarithm of each sample of *sound*. Use `s-log` instead (see Section 7.2.2.7).

`peak(expression, maxlen)` [SAL]

`(peak expression maxlen)` [LISP]

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 38 for examples.

`snd-max(expression, maxlen)` [SAL]

`(snd-max expression maxlen)` [LISP]

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)` [SAL]

`(snd-maxv sound1 sound2)` [LISP]

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 7.2.2.7).

`snd-normalize(sound)` [SAL]

`(snd-normalize sound)` [LISP]

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)` [SAL]

`(snd-oneshot sound threshold ontime)` [LISP]

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)` [SAL]

`(snd-prod sound1 sound2)` [LISP]

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)` [SAL]

`(snd-pwl t0 sr lis)` [LISP]

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 (FIXNUMs, 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)` [SAL]

`(snd-quantize sound steps)` [LISP]

Quantizes a sound. See Section 7.2.2.7 for details.

`snd-recv(sound)` [SAL]

`(snd-recv sound)` [LISP]

Compute the reciprocal of each sample of *sound*. Use `recv` instead (see Section 7.2.2.7).

`snd-resample(f, rate)` [SAL]

`(snd-resample f rate)` [LISP]

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)` [SAL]

`(snd-resamplev f rate g)` [LISP]

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)` [SAL]

`(snd-scale scale sound)` [LISP]

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

`snd-shape(signal, table, origin)` [SAL]

`(snd-shape signal table origin)` [LISP]

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)` [SAL]

`(snd-up srate sound)` [LISP]

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)` [SAL]

`(snd-xform sound sr time start stop scale)` [LISP]

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*, (2) 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 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` or `sound` to create a transformable sound (see Section 7.2.1).

`snd-yin(sound, minstep, maxstep, rate)` [SAL]

`(snd-yin sound minstep maxstep rate)` [LISP]

Identical to `yin`. See Section 7.2.2.7.

### 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)` [SAL]

`(snd-alpass sound delay feedback)` [LISP]

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)` [SAL]

`(snd-alpasscv sound delay feedback)` [LISP]

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)` [SAL]

`(snd-alpassvv sound delay feedback maxdelay)` [LISP]

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)` [SAL]

`(snd-areson sound hz bw normalization)` [LISP]

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*) [SAL]

(`snd-aresoncv` *sound hz bw normalization*) [LISP]

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*) [SAL]

(`snd-aresonvc` *sound hz bw normalization*) [LISP]

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*) [SAL]

(`snd-aresonvv` *sound hz bw normalization*) [LISP]

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*) [SAL]

(`snd-atone` *sound hz*) [LISP]

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*) [SAL]

(`snd-atonev` *sound hz*) [LISP]

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*) [SAL]

(`snd-biquad` *sound b0 b1 b2 a1 a2 z1init z2init*) [LISP]

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*) [SAL]

(`snd-chase` *sound risetime falltime*) [LISP]

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*) [SAL]

(*snd-congen gate risetime falltime*) [LISP]

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*) [SAL]

(*snd-convolve sound response*) [LISP]

Convolve *sound* by *response* using a fast convolution algorithm. The *sound* can be any length, but the *response* is computed and stored in memory. The required computation 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*) [SAL]

(*snd-delay sound delay feedback*) [LISP]

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*) [SAL]

(*snd-delaycv sound delay feedback*) [LISP]

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*) [SAL]

(*snd-reson sound hz bw normalization*) [LISP]

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 taken from *sound*. You should use *reson* instead (see Section 7.2.2.3).

`snd-resoncv`(*sound*, *hz*, *bw*, *normalization*) [SAL]

(`snd-resoncv` *sound hz bw normalization*) [LISP]

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*) [SAL]

(`snd-resonvc` *sound hz bw normalization*) [LISP]

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*) [SAL]

(`snd-resonvv` *sound hz bw normalization*) [LISP]

This function is identical to `snd-reson` except both *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*) [SAL]

(`snd-stkchorus` *sound delay depth freq mix*) [LISP]

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). You should use `stkchorus` instead (see Section 7.2.2.4).

`snd-stkpitshift`(*sound*, *shift*, *mix*) [SAL]

(`snd-stkpitshift` *sound shift mix*) [LISP]

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). You should use `pitshift` instead (see Section 7.2.2.4).

`snd-stkrev`(*rev-type*, *sound*, *decay*, *mix*) [SAL]

(`snd-stkrev` *rev-type sound decay mix*) [LISP]

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 JCreverb, and two selects PRCreverb. 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 that of *sound*. You should use `nrev`, `jcrev` or `prcrev` instead (see Section 7.2.2.4).

`snd-tone`(*sound*, *hz*) [SAL]

(`snd-tone` *sound hz*) [LISP]

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 taken from *sound*. You should use `lp` instead (see Section 7.2.2.3).

`snd-tonev`(*sound*, *hz*) [SAL]

(`snd-tonev` *sound* *hz*) [LISP]

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 `1p` 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)` [SAL]

`(snd-amosc sound step sr hz t0 am phase)` [LISP]

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)` [SAL]

`(snd-fmosc s step sr hz t0 fm phase)` [LISP]

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)` [SAL]

`(snd-fmfb t0 hz sr index dur)` [LISP]

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)`

`(snd-fmfbv t0 hz sr index)` [LISP]

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)` [SAL]

`(snd-buzz n sr hz t0 fm)` [LISP]

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)` [SAL]

`(snd-pluck sr hz t0 d final-amp)` [LISP]

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)` [SAL]

`(snd-osc s step sr hz t0 d phase)` [LISP]

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)` [SAL]

`(snd-partial sr hz t0 env)` [LISP]

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)` [SAL]

`(snd-sine t0 hz sr d)` [LISP]

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-sampler(s, step, start, sr, hz, t0, fm, npoints)` [SAL]

`(snd-sampler s step start sr hz t0 fm npoints)` [LISP]

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 sound *s* is the source sound to be looped, and *step* (a FLONUM) is the nominal fundamental frequency (in steps, not Hz) of *s*. The *start* (a FLONUM) is the time in seconds at which to start the loop, *sr* (a FLONUM) is the desired sample rate of the output, *hz* is the nominal fundamental frequency of the output, *t0* (a FLONUM) is the starting time of the output, and *fm* (a SOUND) is frequency modulation that is added to *hz* to determine the output fundamental frequency. The parameter *npoints* (a FIXNUM) 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. This function implements a typical sampling synthesis algorithm, looping and resampling the input according to the ratio between the desired fundamental frequency (which is the sum of *hz* and *fm*) and the nominal fundamental of the looped sound (which is assumed to be given by *step*). You should use `sampler` instead (see Section 7.2.2.1).

`snd-siosc(tables, sr, hz, t0, fm)` [SAL]

`(snd-siosc tables sr hz t0 fm)` [LISP]

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 Phase Vocoder Functions

`snd-phasevocoder(s, map, fftsize, hopsiz mode)` [SAL]

`code{(snd-phasevocoder s map fftsize hopsiz mode)` [LISP]}

Phase vocoder: Identical to `phasevocoder` except that *fftsize*, *hopsiz* and *mode* are not optional. Specify -1 to get the default values for *fftsize* and *hopsiz*. Specify 0 for the default value of *mode*. You should use `phasevocoder` instead (see Section 7.2.2.6).

## 7.6.6 Physical Model Functions

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

`snd-bandedwg(freq, bowpress-env, preset, sr)` [SAL]

`(snd-bandedwg freq bowpress-env preset sr)` [LISP]

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)` [SAL]

`(snd-bowed freq bowpress-env sr)` [LISP]

A bowed string instrument implemented in STK. The *freq* is a FLONUM in Hertz, *bowpress-env* is a SOUND that ranges from zero 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)` [SAL]

`(snd-bowed-freq freq bowpress-env freq-env sr)` [LISP]

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)` [SAL]

`(snd-clarinet freq breath-env sr)` [LISP]

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)` [SAL]

`(snd-clarinet-freq freq breath-env freq-env sr)` [LISP]

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)` [SAL]

`(snd-clarinet-all freq vibrato-freq vibrato-gain freq-env breath-env  
reed-stiffness noise sr)` [LISP]

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)` [SAL]

`(snd-flute freq breath-env sr)` [LISP]

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)` [SAL]  
`(snd-flute-freq freq breath-env freq-env sr)` [LISP]  
 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)` [SAL]  
`(snd-flute-all freq vibrato-freq vibrato-gain freq-env breath-env jet-delay noise sr)` [LISP]  
 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)` [SAL]  
`(snd-mandolin t0 freq dur body-size detune sr)` [LISP]  
 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)` [SAL]  
`(snd-modalbar t0 freq preset dur sr)` [LISP]  
 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)` [SAL]  
`(snd-sax freq breath-env sr)` [LISP]  
 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)` [SAL]  
`(snd-sax-freq freq freq-env breath-env sr)` [LISP]  
 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)` [SAL]  
`(snd-sax-all freq vibrato-freq vibrato-gain freq-env breath-env reed-stiffness noise blow-pos reed-table-offset sr)` [LISP]  
 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)` [SAL]

`(snd-sitar t0 freq dur sr)` [LISP]

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).

## 7.6.7 Sequence Support Functions

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

`snd-seq(sound, closure)` [SAL]

`(snd-seq sound closure)` [LISP]

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 directly.

`snd-multiseq(array, closure)` [SAL]

`(snd-multiseq array closure)` [LISP]

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*. The sample rates and number of channels returned from the closure must match the first multi-channel sound in the sequence. Do not call this function directly.

## 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:

`*~=tolerance*`

The tolerance used by the SAL "approximately equal" operator (`~=`) to determine if two numbers are approximately equal.

`*table*`

Default table used by `osc` and other oscillators. Initially set to the sinusoid in `*sine-table*`.

`*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 C.15.

**\*clipping-error\***

If the peak absolute amplitude value of a sound saved or played exceeds **\*clipping-threshold\***, an XLISP error is raised. See **\*clipping-threshold\*** for more detail.

**\*clipping-threshold\***

See **\*clipping-error\*** for a description of this variable. **\*clipping-threshold\*** is initialized to 127/128. This number is conservative, and it is possible to slightly exceed this value, even with 8-bit files without actual clipping (consider rounding. Also, floating point format files will not clip even when the amplitude exceeds 1.0. Note that a "clipping" threshold of 1.0 is optimistic: 1.0 corresponds to a 16-bit integer value of 32,768 ( $2^{15}$ ), but the maximum positive 16-bit integer is 32,767. Thus, a positive sample of 1.0 will clip when written or played as 16-bit audio.

**\*control-srate\***

Part of the environment, establishes the control sample rate. See Section 3.1 for details.

**\*default-plot-file\***

The default file for plot data (written by Nyquist, read by NyquistIDE). Default value is "points.dat".

**\*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.lisp`.

**\*lpslider-cutoff\***

The cutoff frequency used by `lpslider`, described in Section 9.1.3. The default value is 20Hz. If unbound, this variable is set when you load `sliders.lisp`.

**\*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.

**\*saw-table\***

A single cycle sawtooth wave intended for use by table-lookup oscillators such as `osc`. This is the waveform used by `osc-saw`. Note that some aliasing will normally occur when using this waveform, which is not band-limited.

**\*sine-table\***

A single cycle sinusoid intended for use by table-lookup oscillators such as `osc`.

**\*snd-device\***

Normally, this is set by `NyquistIDE`. Select the audio output device by setting this to a `FIXNUM` to select a device by index number or to a `STRING` to select a device by name. If a `STRING` is provided, the first device whose name contains the `STRING` (as a substring) is selected. (See `*snd-list-devices*` below.) If the value is unbound or `NULL`, `*snd-device-default*` is used (see below).

**\*snd-device-default\***

If `*snd-device*` is unbound or null, this variable can select the audio output device in the same way as `*snd-device*`. Normally, this variable would be set in an initialization file that provides a default value that can be overridden by the `NyquistIDE` preferences. If both `*snd-device*` and `*snd-device-default*` are null, the default `PortAudio` device is used.

**\*snd-list-devices\***

List all audio output devices (as text output) when a sound is played. By inspecting this list, one can determine possible values for `*snd-device*` and see what device is being selected by `Nyquist`. One cannot (currently) obtain the list of devices as an `XLISP` value. The default behavior is to list the devices and some instructions only the first time any sound is played. Setting `*snd-list-devices*` to a value (`t` or `nil`) before playing a sound will override this behavior.

**\*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.

**\*tri-table\***

A single cycle triangle wave intended for use by table-lookup oscillators such as `osc`. This is the waveform used by `osc-tri`. Note that some aliasing will normally occur when using this waveform, which is not band-limited.

**XLISP variables**

See Section C.15 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 1.8 for definitions of predefined constants for loudness, duration, and pitch.

## 9 Interactive Nyquist

Nyquist is not intended for real-time performance, but it has some features that allow you to adjust parameters interactively. The basic idea is that there is an array of 1000 floating point values, called sliders, that can be accessed while synthesizing sounds in Nyquist. The `slider` unit generator returns a signal that copies the current value of a slider value. You can change the slider value while playing sounds using either Open Sound Control or the `NyquistIDE`.

Sounds are normally computed on demand. So the result returned by `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.)

When you load `sliders.lsp`, which defines a number of slider functions, two important settings may be changed. First `autonorm-off` is called. The problem with auto-normalization is that it works by computing 1 million samples ahead of real time to determine a normalization factor. If Nyquist computes ahead, it will be unable to respond to control changes until the million samples (about 20 seconds) have been played. Secondly, `snd-set-latency` is used to set the audio latency to 0.02s (20 milliseconds). Normally, Nyquist uses a generous 0.3s latency which allows Nyquist to stop computing audio and run garbage collection without breaks in the audio output. At 20ms, interactivity is greatly enhanced because changes do not sit in audio buffers for 300ms, but you may notice some break-up in the audio, especially when garbage collection takes place. The latency can be changed to any value you like after you load `sliders.lsp`.

In addition to reading sliders as continually changing `SOUNDS`, you can get the slider value as a Lisp `FLONUM` (a floating point number) using `get-slider-value`. 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.

Other unit generators exist to instantiate behaviors and to stop sounds according to slider values. These will be described below.

## 9.1 Interactive Control with the NyquistIDE

To control sounds interactively using the NyquistIDE, you first create a control panel, then populate the panel with sliders and buttons. These will send values into the sliders array in the Nyquist process. To control a sound, you use built-in functions to retrieve real-time values from the sliders array as sound is being played. Further discussion and examples can be found in `nyquist/lib/sliders/slider-demos.sal`.

### 9.1.1 Creating a Control Panel

A control panel is created with `make-slider-panel`, which takes a panel name and color as parameters. Control panels *can only be created by executing code in Nyquist*. There is no way to configure control panels directly using the NyquistIDE. Control panels can be deleted interactively using the close button on the panel or through code by calling `close-slider-panel`.

`make-slider-panel`(*name*, *color*) [SAL]

(`make-slider-panel` *name color*) [LISP]

Create a control panel in the IDE. The title of the control panel window is given by *name*, a STRING. The color of the panel is given by *color*, a FIXNUM from 0 through 12. The color 0 is gray. Other colors are implementation-dependent, but different numbers give distinguishable colors. You must load `sliders.lsp` to access this function.

`close-slider-panel`(*name*) [SAL]

(`close-slider-panel` *name*) [LISP]

Close a control panel in the NyquistIDE named by *name*, a STRING. Any embedded controls (sliders or buttons) are also destroyed. You must load `sliders.lsp` to access this function.

### 9.1.2 Creating Controls

You can create slider and button controls. A slider control adjusts a floating point value in the sliders array and accessible as a time-varying signal (a SOUND) or as a FLONUM. A button control sets a floating point value in the sliders array to zero (0.0) or one (1.0). The value changes when the left mouse button is pressed over the button control and changes back when the mouse button is released.

`make-slider`(*name*, *init*, *low*, *high*) [SAL]

(`make-slider` *name init low high*) [LISP]

Create a slider in the most recently created control panel. (Thus, you should populate a control panel with sliders and buttons before creating another control panel.) Sliders have a label specified by *name*, a STRING, an initial value specified by *init*, a FLONUM, a minimum value specified by *low*, a FLONUM, and a maximum value specified by *high*, a FLONUM. Sliders are added to the current panel in order from top to bottom. You must load `sliders.lsp` to access this function.

`make-button(name [, normal])` [SAL]  
`(make-button name [normal])` [LISP]  
 Create a button in the most recently created control panel. Buttons have a label specified by *name*, a STRING, and a "normal value" specified by 0 or 1 (a FIXNUM). If 0 is specified, the value controlled by the button is 0 when the button is released and 1 when the button is pressed. If 1 is specified, the normal value is 1 changing to 0 when the button is pressed. Buttons are added to the control panel in order from top to bottom. You must load `sliders.lsp` to access this function.

### 9.1.3 Accessing Control Values

Each control created by `make-slider` or `make-button` is assigned a slider index from 10 to 999. Control changes are passed via hidden text input from the NyquistIDE to the nyquist process, where the values are converted to floats and stored in the slider array. You can then access these values with either `slider`, `lpslider`, or `get-slider-value`.

`slider(number [, dur])` [SAL]  
`slider(name [, dur])` [SAL]  
`slider(panel, name [, dur])` [SAL]  
`(slider number [dur])` [LISP]  
`(slider name [dur])` [LISP]  
`(slider panel name [dur])` [LISP]  
 Create a SOUND that reads signal values from the slider array. In the first form, the first parameter is the index (a FIXNUM) of the value in the slider array. In the second form, the slider value will be controlled by the NyquistIDE control created by `make-slider` or `make-button` using the same *name*, a STRING. The control must be in the most recently created panel. In the third form, the panel named *panel* is searched for the control named *name* to determine the value. In all cases, the optional *dur*, a FLONUM, is used to determine the duration of the sound. This duration is scaled by the environment in the usual way. You must load `sliders.lsp` to access this function.

`lpslider(number [, dur])` [SAL]  
`lpslider(name [, dur])` [SAL]  
`lpslider(panel, name [, dur])` [SAL]  
`(lpslider number [dur])` [LISP]  
`(lpslider name [dur])` [LISP]  
`(lpslider panel name [dur])` [LISP]  
 Create a SOUND based on the value of an interactive control. This function is exactly like `slider`, except the sound is low-pass filtered to avoid sudden jumps when the control value is adjusted. The low-pass filter cutoff is determined by `*lpslider-cutoff*`, which is initialized to 20Hz when `slider.lsp` is loaded. You must load `sliders.lsp` to access this function.

`get-slider-value(number)` [SAL]  
`get-slider-value(name)` [SAL]  
`get-slider-value(panel, name)` [SAL]  
`(get-slider-value number)` [LISP]  
`(get-slider-value name)` [LISP]  
`(get-slider-value panel name)` [LISP]  
 Get a value (a FLONUM) stored in the slider array. The array is accessed directly if the parameter is *number*, a FIXNUM. The index can be determined by searching the most recently created control panel by *name*, a

STRING, if only *name* is given. If both *panel* and *name* are given (STRINGS), the named panel is searched for the named control. The result is a FLONUM. You must load `sliders.lsp` to access this function.

`snd-slider(index, t0, srate, duration)` [SAL]

`(snd-slider index t0 srate duration)` [LISP]

Create a sound controlled by the slider named by *index* (an integer index into the array of sliders). 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. Normally, you would call `slider` (see above) rather than this low-level function.

### 9.1.4 Starting and Stopping Sounds

All Nyquist sounds have a duration. Even the `slider` unit generator has a duration and terminates at the end of that duration. In most cases, you will instead want interactive functions to run until you interactively ask them to stop. The `stop-on-zero` function terminates when an input signal (typically a slider) goes to zero. This can be used to terminate a complex interactively controlled sound. For example, here is a tone that terminates when the Stop button is pressed:

```
exec make-button("Stop", 1)
```

```
function can-stop()
  play (hzosc(1000) * stop-on-zero(slider("Stop"))) ~ 100
```

The stretch factor of 100 will cause this tone to play for 100 seconds. However, if the button named "Stop" is pressed, it will change value from the "normal" value 1 to 0. When the signal goes to zero, `stop-on-zero` will terminate. The multiplication then immediately terminates (because anything multiplied by zero will be zero; termination is just an efficient way to return zeros from now on). Since the multiplication is the top-level sound being played, the play stops.

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 can be used, for example, to create a sequence of sound events interactively. For example:

```
exec make-button("Trigger", 0)
```

```
function trigger-me()
  play trigger(slider("Trigger", 100), pluck(c3))
```

Here, a button control changes value from 0 to 1 when the button is pressed. This value is retrieved by the `slider` function, which runs for 100 seconds. When the button and hence the `slider` goes from 0 to 1, the behavior, `pluck(c3)` is instantiated. Many instances can be triggered by the button.

`stop-on-zero(s)` [SAL]

`(stop-on-zero s)` [LISP]

Return a SOUND that is identical to *s*, a SOUND, except the returned sound terminates when *s* first goes to zero. When a sound terminates, it remains at zero. A SOUND multiplication terminates when either parameter terminates, so multiplying by `stop-on-zero` is a way to terminate a sound interactively. (See the example above.) You must load `sliders.lsp` to access this function.

`trigger(s, beh)` [SAL]

`(trigger s beh)` [LISP]

Returns a sound which is the sum of zero or more possibly overlapping 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 start time of the result is the start time of *s*, and zero samples will be generated until the first instance of *beh*. The sample rate of the result is `*sound-srate*`, and all instances of *beh* must have the same `*sound-srate*` sample rate. The behaviors (instances of *beh*) must be (monophonic) SOUNDS. The stop time of the result is the maximum stop time of *s* and all sounds returned by instances of the behavior. This function is particularly designed to allow behaviors to be invoked in real time. See the `trigger-me` function definition shown above.

An implementation note: There is no way to have `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 `trigger` in the first place.

Warning: The *beh* argument of `trigger` is converted to a closure that captures the current environment, including any variables in scope. The following example illustrates a problem where *s* is a local variable:

```
(defun example (snd) (trigger snd (pluck c4)))
```

or in SAL:

```
function example(snd) return trigger(snd, pluck(c4))
```

The problem here is that *snd* will be captured by the closure built from `pluck(c4)`. As `trigger` begins to evaluate *snd* looking for zero crossings, the samples from *snd* will be retained in memory because *snd* is retained in the closure. A solution is the following:

```
(defmacro unbind (sym) `(let ((x ,sym)) (setf ,sym nil) x))
(defun example (snd) (trigger (unbind snd) (pluck c4)))
```

or in SAL:

```
;; unbind must be defined as above and loaded from a .lsp file
function example(snd) return trigger(unbind(snd), pluck(c4))
```

In this code, *snd* is still retained by the constructed closure, but it is set to `nil` by `unbind`, which returns the *value* of *snd*. Thus, the only surviving reference to the original value of *snd* is held by `trigger`, which frees samples immediately after computing them. No samples are retained in memory.

`snd-stoponzero(s)` [SAL]

`(snd-stoponzero s)` [LISP]

This function is identical to `stop-on-zero`. You should use `stop-on-zero` instead.

`snd-trigger(s, closure)` [SAL]

`(snd-trigger s closure)` [LISP]

This is a low-level support function for `trigger`. The *closure* takes a starting time and returns a SOUND. See `trigger` above for more details. Use `trigger` as described above and do not call this function directly.

## 9.2 Using Open Sound Control

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.

The `slider` and `get-slider-value` functions, described above, can be used to access these values within Nyquist. Each of these functions can take a slider array index to specify which value to use. Since `make-slider` allocates slider indices starting at 10, it is recommended that you control sliders 0 through 9 via OSC. If you change a slider array value via OSC that is already controlled by a graphical slider in the NyquistIDE, the graphical slider will not be updated or synchronized to the OSC value. (And there is no current way to send OSC command to the NyquistIDE.)

Note: Open Sound Control must be enabled by calling `osc-enable(t)`. If this fails under Windows, see the installation instructions in `sys/win/README.txt` regarding `SystemRoot`.

`osc-enable(flag)` [SAL]

`(osc-enable flag)` [LISP]

Enable or disable Open Sound Control. (See Section 9.2.) 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 `slider` and `get-slider-value` functions. 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`. *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.

## 9.2.1 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.

## 9.2.2 Python3 OSC Interface Demo

In `nyquist/demos/osc` are two programs that demonstrate controlling Nyquist sounds with Python in real time. Open `nyquist/demos/osc/getosc.sal` in the NyquistIDE and read the comments. There are several functions you can run or play while executing the Python program `nyquist/demos/osc/nyquistosc.py` from a command line (terminal) window.

## 9.2.3 Test Programs in C

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.

## 9.2.4 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 the 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 `nyquist/lib/osc/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.

## 10 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 into 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-iff`, and turn the sequence back into audio.

With `snd-fft` and `snd-iff`, 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 `nyquist/lib/fft/fft_tutorial.htm`, which is part of the standard Nyquist release. The documentation for `snd-fft` and `snd-iff` follows.

`snd-fft(sound, length, skip, window)` [SAL]

`(snd-fft sound length skip window)` [LISP]

This function performs an FFT on the first samples in *sound* and returns a Lisp array of FLONUMs. 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) which must be a power of 2. 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. The last array element corresponds to the real coefficient at the Nyquist frequency.

`snd-iff(time, sr, iterator, skip, window)` [SAL]

`(snd-iff time sr iterator skip window)` [LISP]

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 *sr*. 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 and less than the frame (FFT) size. 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 power of 2. The length is implied by the first 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-iff`.

## 10.1 Spectral Processing

There are a number of functions defined to make spectral processing easier in XLISP and SAL. The general approach, as described above, is to create an iterator object that returns spectral frames. To avoid using the XLISP object system directly, a more functional interface is defined, especially for SAL users. The `sa-init` function creates an iterator, and `sa-next` retrieves spectral frames. Various functions are also provided to transform these into amplitude (magnitude) spectra, plot them and perform other operations.

Some examples that use these spectral processing functions can be found in the Nyquist extension "fftsal" (use the NyquistIDE's Window : Nyquist Extensions menu item to download it; it will then be in your `nyquist/lib/fftsal` directory. You can find descriptions of the examples in `nyquist/lib/fftsal/spectral-process.lsp` and `nyquist/lib/fftsal/spectral-process.sal`.

```
sa-init(resolution: hz, fft-dur: dur, skip-period: skip, window: window-type, input:
input) [SAL]
(sa-init :resolution hz :fft-dur dur :skip-period skip :window window-type :input input)
[LIISP]
```

Creates a spectral-analysis object that can be used to obtain spectral data from a sound. All keyword parameters are optional except `input`. The `resolution` keyword parameter gives the width of each spectral bin in Hz. It may be `nil` or not specified, in which case the resolution is computed from `fft-dur`. The actual resolution may be finer than the specified resolution because `fft` sizes are rounded to a power of 2. The `fft-dur` is the width of the FFT window in seconds. The actual FFT size will be rounded up to the nearest power of two in samples. If `nil`, `fft-dur` will be calculated from `resolution`. If both `fft-size` and `resolution` are `nil` or not specified, the default value is 1024 samples, corresponding to a duration of `1024 / signal-sample-rate`. If both `resolution` and `fft-dur` are specified, the `resolution` parameter will be ignored. Note that `fft-dur` and `resolution` are reciprocals. The `skip-period` specifies the time interval in seconds between successive spectra (FFT windows). Overlapping FFTs are possible. The default value overlaps windows by 50%. Non-overlapped and widely spaced windows that ignore samples by skipping over them entirely are also acceptable. The `window` specifies the type of window. The default is raised cosine (Hann or "Hanning") window. Options include `:hann`, `:hanning`, `:hamming`, `:none` or `nil`, where `:none` and `nil` mean a rectangular window. The `input` can be a string (which specifies a sound file to read) or a Nyquist SOUND to be analyzed. The return value is an XLISP object that can be called to obtain parameters as well as a sequence of spectral frames. Normally, you will set a variable to this result and pass the variable to `sa-next`, described below.

`sa-info(sa-obj)` [SAL]  
`(sa-info sa-obj)` [LISP]  
 Prints information about an *sa-obj*, which was created by `sa-init` (see above). The return value is `nil`, but information is printed.

`sa-next(sa-obj)` [SAL]  
`(sa-next sa-obj)` [LISP]  
 Fetches the next spectrum from *sa-obj*, which was created by `sa-init` (see above). The return value is an array of FLONUMs representing the discrete complex spectrum.

`sa-magnitude(frame)` [SAL]  
`(sa-magnitude frame)` [LISP]  
 Computes the magnitude (amplitude) spectrum from a frame returned by `sa-frame`. The  $i^{\text{th}}$  bin is stored at index  $i$ . The size of the array is the FFT size / 2 + 1.

`sa-normalize(frame [, max])` [SAL]  
`(sa-normalize frame [max])` [LISP]  
 Normalize a copy of *frame*, a magnitude (amplitude) spectrum returned by `sa-magnitude`. If *max* (a FLONUM) is provided, the spectrum will be normalized to have a maximum value of *max*, which defaults to 1.

`sa-plot(sa-obj, frame)` [SAL]  
`(sa-plot sa-obj frame)` [LISP]  
 Plots a magnitude (amplitude) spectrum from *frame* returned by `sa-magnitude`. The *sa-obj* parameter should be the same value used to obtain the frame.

`sa-print(file, sa-obj, frame, cutoff: cutoff, threshold: threshold)` [SAL]  
`(sa-print sa-obj file frame :cutoff cutoff :threshold threshold)` [LISP]  
 Prints an ASCII plot of *frame*, a magnitude (amplitude) spectrum returned by `sa-magnitude` (or `sa-normalize`). The *file* is either a file opened for writing or T to print to the console. The caller is responsible for closing the file (eventually). The *sa-obj* parameter should be the same value used to obtain the frame. If *cutoff*, a FLONUM, is provided, only the spectrum below cutoff (Hz) will be printed. If *threshold*, a FLONUM, is provided, the output may elide bins with values below the threshold.

`sa-get-bin-width(sa-obj)` [SAL]  
`(sa-get-bin-width sa-obj)` [LISP]  
 Returns the width of a frequency bin as a FLONUM in Hz (also the separation of bin center frequencies). The center frequency of the  $i^{\text{th}}$  bin is  $i * \text{bin-width}$ .

`sa-get-fft-size(sa-obj)` [SAL]  
`(sa-get-fft-size sa-obj)` [LISP]  
 Returns a FIXNUM, the size of the FFT, a power of 2.

`sa-get-fft-dur(sa-obj)` [SAL]  
`(sa-get-fft-dur sa-obj)` [LISP]  
 Returns a FIXNUM, the duration of the FFT window.

`sa-get-fft-window(sa-obj)` [SAL]  
`(sa-get-fft-window sa-obj)` [LISP]  
 Returns a symbol representing the type of window used, `:hann`, `:hamming` or `:none`.

`sa-get-skip-period(sa-obj)` [SAL]  
`(sa-get-skip-period sa-obj)` [LISP]  
 Returns the skip size in seconds (a FLONUM).

`sa-get-fft-skip-size(sa-obj)` [SAL]  
`(sa-get-fft-skip-size sa-obj)` [LISP]  
 Returns the skip size in samples (a FIXNUM).

`sa-get-sample-rate(sa-obj)` [SAL]

`(sa-get-sample-rate sa-obj)` [LISP]

Returns the sample rate of the sound being analyzed (a FLONUM) in Hz.

## 11 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 14.) A tutorial on using the Adagio representation and MIDI can be found in `nyquist/lib/midi/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 `nyquist/lib/midi/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 14) 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.

### 11.1 The SEQ Type

Nyquist has a special data type to store MIDI data called a SEQ (short for "sequence"). You can create an empty SEQ object, read data into it from a MIDI or Adagio file, write a SEQ as MIDI or Adagio, and insert notes into a SEQ object. You can also convert SEQ objects into sound by providing functions to handle different MIDI messages. Further discussion and examples can be found in `lib/midi/midi_tutorial.htm`.

`seq-create()` [SAL]

`(seq-create)` [LISP]

Creates a SEQ data object.

`seq-read(seq, file)` [SAL]

`(seq-read seq file)` [LISP]

Reads into a SEQ data object from an ASCII text file in Adagio format. The first parameter is the SEQ object to read. The *file* is a file opened for reading. It must be closed after `seq-read` returns.

`seq-read-smf(seq, midi-file)` [SAL]

`(seq-read seq midi-file)` [LISP]

Reads into a SEQ data object from a Standard MIDI File. The first parameter is the SEQ object to read. The *midi-file* is a binary file opened in *binary mode* for reading. It must be closed after `seq-read-smf` returns.

`seq-write(seq, file, absolute)` [SAL]

`(seq-write seq file absolute)` [LISP]

Writes a SEQ data object to an ASCII text file in Adagio format. The first parameter is the SEQ object to write. The *file* is a file opened for writing. It must be closed after `seq-read` returns. The *absolute* parameter should be true to write absolute times of events and false (NIL) to write relative times.

`seq-write-smf(seq, midi-file)` [SAL]

`(seq-write-smf seq midi-file)` [LISP]

Writes a SEQ data object to a binary file in Standard MIDI File format. The first parameter is the SEQ object to write. The *midi-file* is a *binary* file opened for writing. The file is closed by `seq-write-smf`.

`seq-insert-note(seq, time, line, chan, pitch, dur, loud)` [SAL]

`(seq-insert-note seq time line chan pitch dur loud)` [LISP]

Inserts a new note into the SEQ object *seq*. The note will be at *time* (a FIXNUM in milliseconds), with *chan* (MIDI channel) given by a FIXNUM starting at 0, duration given by *dur* (a FIXNUM in milliseconds), *pitch* (a FIXNUM with MIDI key number), and MIDI velocity given by *loud*, a FIXNUM from 1 to 127. The *line* field is intended as a source code line number for Adagio files and can be any FIXNUM.

`seq-insert-ctrl(seq, time, line, ctrltype, chan, ctrlnum, value)` [SAL]

`(seq-insert-ctrl seq time line ctrltype chan ctrlnum value)` [LISP]

Inserts a new control into the SEQ object *seq*. Here, “control” refers to a variety of MIDI messages, including program change, pitch bend, and aftertouch in addition to control changes. The control will be at *time* (a FIXNUM in milliseconds). The *ctrltype* (FIXNUM) uses MIDI status bytes for convenience: `seq-ctrl-tag` (11) for a MIDI control change, `seq-prgm-tag` (12) for a MIDI program change, `seq-cpress-tag` (13) for a MIDI channel pressure change, or `seq-bend-tag` (14) for a MIDI pitch bend change. The *chan* (MIDI channel) is given by a FIXNUM starting at 0. If *ctrltype* is 11 (control change), then *ctrlnum* is the MIDI controller number, otherwise this parameter is ignored. The *value* is the MIDI data value (FIXNUM from 0 to 127). In the case of pitch bend, this value is a FIXNUM from 0 to 255 representing the upper 8 bits of the unsigned 14-bit MIDI pitch bend value. (To convert from a scale of -1 to 1, use  $\text{round}(x * 127 + 128)$ .) This gives about 10-cent resolution on pitch bends of plus or minus one octave, which is normally good enough, but should be better. The *line* field is intended as a source code line number for Adagio files and can be any FIXNUM.

To read data from a sequence, you normally begin by making a shallow copy, but you can probably just call `seq-reset` to initialize the iterator. Then call `seq-get` to get the current event, and `seq-next` to advance to the next event:

`seq-copy(seq)` [SAL]

`(seq-copy seq)` [LISP]

Returns a shallow copy of *seq*. Each copy of a sequence contains an independent iterator providing sequential access to the events in the sequence.

`seq-reset(seq)` [SAL]

`(seq-reset seq)` [LISP]

Reset the sequence iterator to the beginning of the sequence.

`seq-get(seq)` [SAL]

`(seq-get seq)` [LISP]

Get the current event in the sequence. The returned value is a list of FIXNUMs in the form (*eventtype time line chan value1 value2 dur*) as follows:

- *eventtype* - The type of event: `seq-done-tag` (0) indicates end of sequence, `seq-other-tag` (1) indicates the event is not supported in Nyquist, `seq-note-tag` (2) indicates the event is a note, `seq-ctrl-tag` (11) indicates a MIDI control change, `seq-prgm-tag` (12) indicates a MIDI program

change, `seq-cpress-tag` (13) indicates a MIDI channel pressure change, and `seq-bend-tag` (14) indicates a MIDI pitch bend change.

- `seq-ctrl-tag` (3) indicates the event is a MIDI control change
- *time* - Time of the event in milliseconds.
- *line* - The Adagio file line number for the event (may not be defined when data is imported from a MIDI file or elsewhere).
- *chan* - The MIDI channel (zero-based).
- *value1* - The note pitch or controller number, depending upon the event type, or zero for program change, channel pressure or pitch bend. This value is zero-based, e.g. the first controller number is 0.
- *value2* - The note velocity, program number, controller value, channel pressure, or pitch bend (see `seq-insert-ctrl` above for details on pitch bend values.) Program numbers are zero based.
- *dur* - The duration in milliseconds for notes.

Functions to access events returned by `seq-get` are defined in `runtime/seqfnint.lsp`.

`seq-next(seq)` [SAL]

`(seq-next seq)` [LISP]

Advance the sequence iterator to the next event in the sequence unless the iterator is at the last (`seq-done-tag`) event.

## 11.2 Adagio Score Language

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 11.5 starting on page 151 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<sup>1</sup>.
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.

## 11.3 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<sup>th</sup>'s), but this can be changed to a millisecond (1000<sup>th</sup>'s) using the !MSEC command and reset to centiseconds with !CSEC (see Section 11.6.1). In the descriptions below, the term “time unit” will be used to mean whichever convention is currently in effect.

### 11.3.1 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 11.3.4.) Time is measured relative to the time of the most recent Tempo or Rate command. (See the examples in Section 11.5 for some clarification of this point.)

### 11.3.2 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 do 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.

<sup>1</sup>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.

An alternate notation for pitch is  $Pn$ , 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.

### 11.3.3 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 11.3.10). 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  $Un$ , where  $n$  is an integer indicating 100<sup>th</sup>'s of a second (or 1000<sup>th</sup>'s), see Section 11.6.1. 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

### 11.3.4 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:

NO	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 NO and is explained in Section 11.6.2. Articulation effects such as *staccato* can be produced using N, but it is more convenient to use the articulation attribute described in Section 11.3.6.

### 11.3.5 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 11.3.9 for an example.

### 11.3.6 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.

### 11.3.7 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 dynamic markings are translated into numbers as follows:

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

### 11.3.8 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.

### 11.3.9 Timbre (MIDI Program)

A MIDI program (synthesizer preset) can be selected using the attribute  $Zn$ , 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.

### 11.3.10 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 note 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.

### 11.3.11 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 11.6.4).

## 11.4 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.”

## 11.5 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 NO
R Q Z15 V2

A4 H V1 NO
G3 V2

B4 Q V1 NO
F3 V2

C4 Q V1 NO
E3 V2

```

Here, each pair of lines represents two simultaneous notes. The NO 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.

## 11.6 Advanced Features

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

### 11.6.1 Time Units and Resolution

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

```

!CSEC          centisecond time units = 100th
!MSEC          millisecond time units = 1000th

```

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

### 11.6.2 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 NO 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

TO 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
```

### 11.6.3 Control Change Commands

Any control change can be specified using the syntax “ $\sim n (v)$ ”, 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 11.4 on page 151 for rules on whether or not a command will play a note.

## 11.6.4 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.

### 11.6.5 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 24<sup>th</sup> of a beat subdivision (for example, exactly on a beat). If not, the tempo change will take place on the nearest exact 24<sup>th</sup> of a beat subdivision. This may be earlier or later than the requested time.

### 11.6.6 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<sup>2</sup>.

```
* 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
```

### 11.6.7 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!

<sup>2</sup>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.

```

!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

```

### 11.6.8 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

```

### 11.6.9 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 NO.

### 11.6.10 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.

## 12 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 `nyquist/lib/lpc/lpc_tutorial.html`, 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  $A(z) = 1 + a_1z^{-1} + a_2z^{-2} + \dots + a_pz^{-p}$ . The *FILTER-COEFS* array has the form  `#(ap ap-1 ... a3 a2 a1)`.

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.

### 12.1 LPC Classes and Functions

`make-lpanal-iterator(sound, framedur, skiptime, npoles)` [SAL]

`(make-lpanal-iterator sound framedur skiptime npoles)` [LISP]

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)` [SAL]  
`(make-lpc-file-iterator filename)` [LISP]  
 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)` [SAL]  
`(save-lpc-file lpc-iterator filename)` [LISP]  
 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?])` [SAL]  
`(show-lpc-data lpc-iterator iniframe endframe [poles?])` [LISP]  
 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)` [SAL]  
`(allpoles-from-lpc snd lpc-frame)` [LISP]  
 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`.

`nth-frame(lpc-iterator, numframe)` [SAL]  
`(nth-frame lpc-iterator numframe)` [LISP]  
 Get the  $n^{\text{th}}$  frame from an lpc iterator by skipping  $n$  frames and getting the next one. Typical use is to construct a filter using the  $n^{\text{th}}$  frame from an LPC data file using something like this to filter `snd`: `allpoles-from-lpc(snd, nth-frame(make-lpc-file-iterator(filename), n))`.

`lpreson(snd, lpc-iterator, skiptime)` [SAL]  
`(lpreson snd lpc-iterator skiptime)` [LISP]  
 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)` [SAL]  
`(lpc-frame-rms1 frame)` [LISP]  
 Get the energy of the input signal from a frame.

`lpc-frame-rms2(frame)` [SAL]  
`(lpc-frame-rms2 frame)` [LISP]  
 Get the energy of the residual from a frame.

`lpc-frame-err(frame)` [SAL]  
`(lpc-frame-err frame)` [LISP]  
 Get the square root of `RMS1/RMS2` from a frame.

`lpc-frame-filter-coefs(frame)` [SAL]  
`(lpc-frame-filter-coefs frame)` [LISP]  
 Get the filter coefficients from a frame.

## 12.2 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)` [SAL]

`(snd-lpanal samps npoles)` [LISP]

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)` [SAL]

`(snd-allpoles snd lpc-coefs gain)` [LISP]

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)` [SAL]

`(snd-lpreson snd lpc-iterator skiptime)` [LISP]

This function is identical to `lpreson` (see above).

## 13 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.

### 13.1 Debugging

Probably the most important debugging tool is the backtrace. There are two kinds of backtrace: one for SAL, and one for Lisp.

SAL mode is actually just an XLISP function (`sal`) that reads input and evaluates it. When SAL encounters an error, it normally prints a trace of the SAL stack (all the active functions written in SAL), exits the current command, and reads the next command.

If you call XLISP functions from SAL, including most Nyquist sound processing functions, and an error occurs within these XLISP functions, you will only see the SAL function that called the XLISP functions listed in the stack trace. Sometimes you need more details.

When Nyquist encounters an error when it is not running SAL, it normally 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.

To get this XLISP backtrace behavior when SAL encounters an error, you need to have `*breakenable*` set while SAL is running. The best way to do this is to run within the NyquistIDE program, open the Preferences dialog, and choose the desired settings, e.g. “Enable XLISP break on SAL error.”

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 representation 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-rate(100)
set-control-rate(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 everyting 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.

## 13.2 Useful Functions

`grindef(name)` [SAL]

`(grindef name)` [LISP]

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)` [SAL]

`(args name)` [LISP]

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)` [SAL]

`(incf symbol)` [LISP]

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)` [SAL]  
`(decf symbol)` [LISP]  
 Decrement *symbol* by one. (See `incf`, above.)

`push(val, lis)` [SAL]  
`(push val lis)` [LISP]  
 Push *val* onto *lis* (a Lisp list). This is a macro that is equivalent to writing (in Lisp) `(setf lis (cons val lis))`.

`pop(lis)` [SAL]  
`(pop lis)` [LISP]  
 Remove (pop) the first item from *lis* (a Lisp list). This is a macro that is equivalent to writing (in Lisp) `(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, expr1, expr2, ...)` [SAL]  
`(while test expr1 expr2 ...)` [LISP]  
 A conventional “while” loop. If *test* is true, evaluate expressions (*expr1*, *expr2*, 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. In SAL, the loop statement is preferred.

`when(test, action)` [SAL]  
`(when test action)` [LISP]  
 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 somewhat 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])` [SAL]  
`(require-from fnsymbol filename [path])` [LISP]  
 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()` [SAL]  
`(current-path)` [LISP]  
 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)` [SAL]

`(real-random from to)` [LISP]

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

`power(x, y)` [SAL]

`(power x y)` [LISP]

Returns *x* raised to the *y* power.

## 14 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 are:

- pattern objects, used to generate interesting sequences of parameter values,
- random number generators, used to create random sequences from different distributions (contained in the `distributions.lsp` library)
- a standard representation for "note lists" called *scores* and functions to render them as sounds,
- the `score-gen` macro which helps to generate scores from patterns,
- score manipulation functions to select, transform, shift, and perform other operations on scores.

### 14.1 Xmusic Basics

Xmusic is inspired by and based on Common Music by Rick Taube. 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 (the base language for Nyquist).

The Xmusic libraries in Nyquist offer an interesting subset of the tools in Common Music. One important feature of Xmusic is that it is integrated with all of the Nyquist synthesis functions, so you can use Xmusic patterns and scores to control fine details of sound synthesis.

### 14.2 Xmusic Patterns

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))
```

In this example, `pitch-source` is an object of class `cycle-class` which inherits from `pattern-class`.

Because SAL is not an object-oriented language, these classes and their methods are not directly accessible from SAL. Instead, Xmusic defines a functional interface, e.g. `make-cycle` creates an instance of `cycle-class`, and the `next` function, introduced below, retrieves the next value from any instance of `pattern-class`. Using LISP syntax, you can have full access to the methods of all objects (see the source code in `xm.lisp`, but the functional interface described here should be sufficient and it is recommended that you limit your access to this interface.

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. Second, the pattern is accessed (multiple times) using `next`.

`next(pattern-object [, #t])` [SAL]

`(next pattern-object [t])` [LISP]

Returns the next element from a pattern generator object. If the optional second argument is true (default value is false), then an entire period is returned as a list.

## 14.2.1 Nested Patterns

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*.

## 14.2.2 Periods

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) (60 62 64). Notice that the cycle starts from the beginning after only 3 items and the fourth is never reached in this case. The `for:` parameter *could itself be a pattern*, in which case the a new cycle length would be computed at the beginning of every cycle.

A variation on this restructures the stream of items into groups of 3:

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

This prints the lists (60 62 64) (65 60 62). Note that in this case the cycle pattern is producing default cycles of length 4 because there is no `for:` specification, and these are regrouped by the length pattern.

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))
loop repeat 9 exec format(t, "~A ", next(cycle-3)) end
```

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.

### 14.2.3 General Parameters for Creating Pattern objects

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.

### 14.2.4 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)` [SAL]

`(make-cycle items :for for :name name :trace trace)` [LISP]

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.) The list is replaced and restarted every period of the cycle, which defaults to the whole list, but may be specified by the `for:` keyword. If *items* is a pattern, a new period of *items* becomes the list from which items are generated for each cycle pattern. Note that the *items* list may be truncated by the use of `for:.` Different groupings and repetitions can be obtained by nesting `make-cycle` within `make-length` and/or `make-copier` patterns.

### 14.2.5 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)` [SAL]

`(make-line items :for for :name name :trace trace)` [LISP]

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.)

### 14.2.6 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)` [SAL]

`(make-random items :for for :name name :trace trace)` [LISP]

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 (optional) relative probability of selecting this item, *mincount* is the (optional) minimum number of repetitions when this item is selected, and *maxcount* is the (optional) 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.

## 14.2.7 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)` [SAL]

`(make-palindrome items :elide elide :for for :name name :trace trace)` [LISP]

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.

## 14.2.8 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)` [SAL]

`(make-heap items :for for :max max :name name :trace trace)` [LISP]

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.

### 14.2.9 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)` [SAL]

`(make-accumulation items :name name :trace trace)` [LISP]

Return the prefixes of items, e.g. the first element followed by the first and second elements, then the first three, until the entire list is returned. 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`.

### 14.2.10 copier

The `copier-class` makes copies of periods from a sub-pattern. For example, three periods of `make-copier` (`make-length` (`make-cycle` (b c}), 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, `make-length` is used to re-group the cycle into periods 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)` [SAL]

`(make-copier sub-pattern :repeat repeat :merge merge :for for :name name :trace trace)` [LISP]

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.

### 14.2.11 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*) [SAL]

(`make-accumulate` *sub-pattern* *:for for* *:max maximum* *:min minimum*  
*:name name* *:trace trace*) [LISP]

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. Minimum and maximum patterns are reevaluated every cycle. Note that this is similar in name but not in function to `make-accumulation`.

### 14.2.12 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*) [SAL]

(`make-sum` *x y* *:for for* *:name name* *:trace trace*) [LISP]

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*.

### 14.2.13 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*) [SAL]

(`make-product` *x y* *:for for* *:name name* *:trace trace*) [LISP]

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*.

### 14.2.14 eval

The `eval`-class evaluates an expression to produce each output item. The default output period length is 1 if the expression is an expression; otherwise, the expression must be a pattern that returns expressions to be evaluated, and the output periods match the periods of the expression pattern.

`make-eval(expr, for: for, name: name, trace: trace)` [SAL]

`(make-eval expr :for for :name name :trace trace)` [LISP]

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. Note that *expr* is evaluated as a Lisp expression, which is a list consisting of the function name followed by parameter expressions (which may also be lists). In the simplest case (recommended), create a SAL function with no parameters that performs the computation. For example, to create a pattern in SAL that calls the parameterless function `rrandom`, write `make-eval({rrandom})`. In SAL, `{rrandom}` creates a list with the (unevaluated) symbol `rrandom`, equivalent to `list(quote(rrandom))`.

### 14.2.15 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 length 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)` [SAL]

`(make-length pattern length-pattern :name name :trace trace)` [LISP]

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.

### 14.2.16 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 period 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)` [SAL]

`(make-window pattern length-pattern skip-pattern :name name :trace trace)` [LISP]

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.

### 14.2.17 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)
```

[SAL]

```
(make-markov rules past :produces produces :for for :name name :trace trace) [LISP]
```

Generate a sequence of items from a Markov process. The *rules* parameter is a list of rules where each rule 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 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]) [SAL]
```

```
(markov-create-rules sequence order [generalize]) [LISP]
```

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

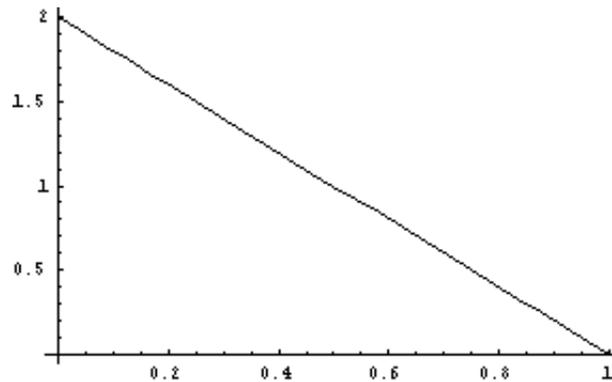


Figure 14.1: The Linear Distribution,  $g = 1$ .

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.

## 14.3 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. Further discussion and examples can be found in `nyquist/demos/probability_distributions.htm`.

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`( $g$ ) [SAL]

(`linear-dist`  $g$ ) [LISP]

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 14.1.) The linear distribution is useful for generating for generating time and pitch intervals.

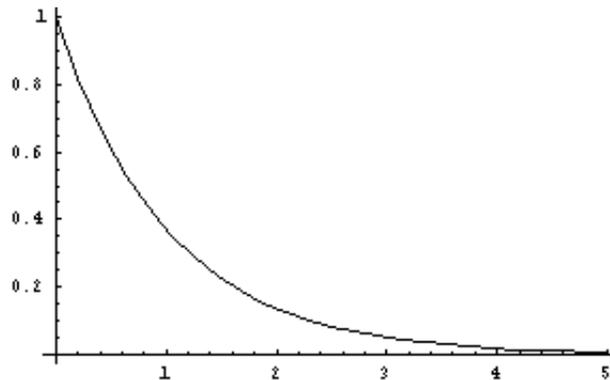


Figure 14.2: The Exponential Distribution,  $\delta = 1$ .

`exponential-dist(delta [, high])` [SAL]

`(exponential-dist delta [high])` [LISP]

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 14.2. 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.

`gamma-dist(nu [, high])` [SAL]

`(gamma-dist nu [high])` [LISP]

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.

`bilateral-exponential-dist(xmu, tau [, low, high])` [SAL]

`(bilateral-exponential-dist xmu tau [low high])` [LISP]

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.

`cauchy-dist(tau [, low, high])` [SAL]

`(cauchy-dist tau [low high])` [LISP]

Returns a FLONUM from the Cauchy distribution, a symmetric 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.

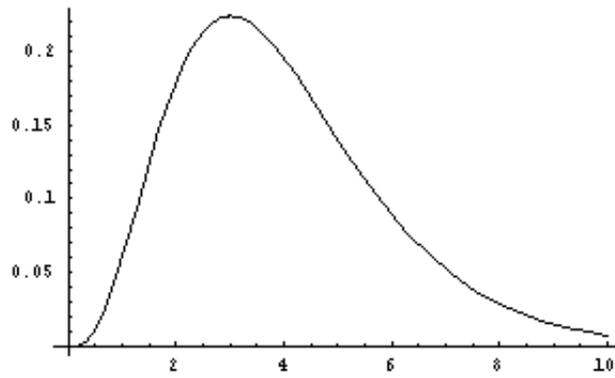


Figure 14.3: The Gamma Distribution,  $nu = 4$ .

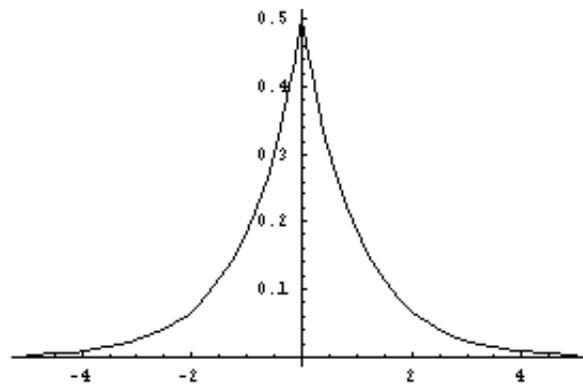


Figure 14.4: The Bilateral Exponential Distribution.

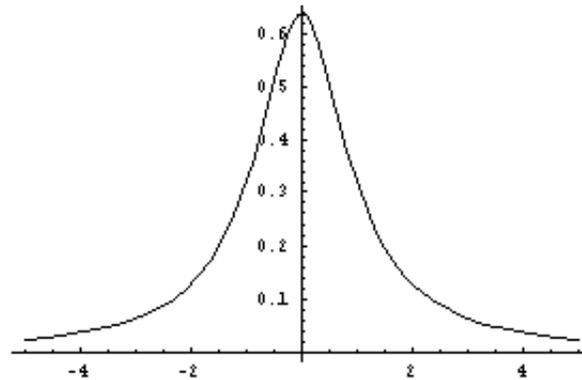
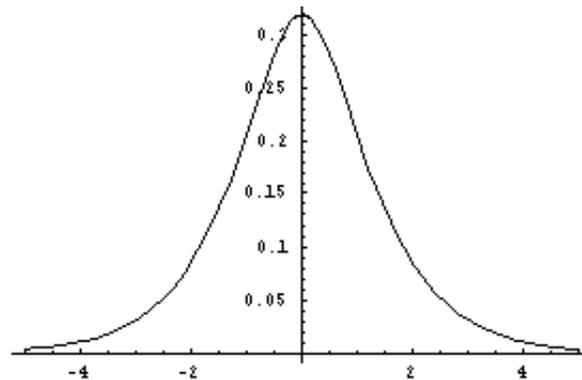
Figure 14.5: The Cauchy Distribution,  $\tau = 1$ .

Figure 14.6: The Hyperbolic Cosine Distribution.

`hyperbolic-cosine-dist` (*low*, *high*) [SAL]

(`hyperbolic-cosine-dist` [*low* *high*]) [LISP]

Returns a FLONUM value from the hyperbolic cosine distribution, a symmetric distribution with its peak at zero. The *low* and *high* parameters give optional artificial bounds on the minimum and maximum output values, respectively.

`logistic-dist` (*alpha*, *beta* [, *low*, *high*]) [SAL]

(`logistic-dist` *alpha* *beta* [*low* *high*]) [LISP]

Returns a FLONUM value from the logistic distribution, which is symmetric 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.

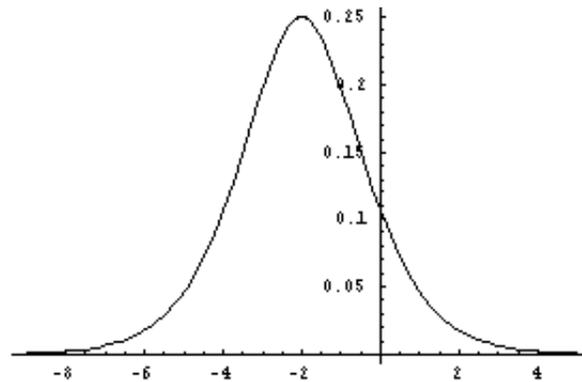
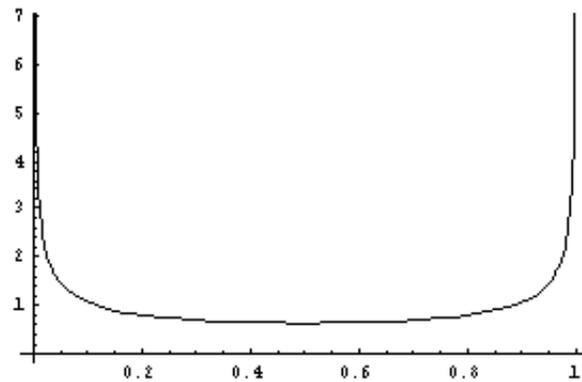
Figure 14.7: The Logistic Distribution,  $\alpha = 1$ ,  $\beta = 2$ .

Figure 14.8: The Arc Sine Distribution.

`arc-sine-dist()` [SAL]

`(arc-sine-dist)` [LISP]

Returns a FLONUM value from the arc sine distribution, which outputs values between 0 and 1. It is symmetric about the mean of  $1/2$ , but is more likely to generate values closer to 0 and 1.

`gaussian-dist(xmu, sigma [, low, high])` [SAL]

`(gaussian-dist xmu sigma [low high])` [LISP]

Returns a FLONUM value from the Gaussian or Gauss-Laplace distribution, a linear function of the normal distribution. It is symmetric 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.

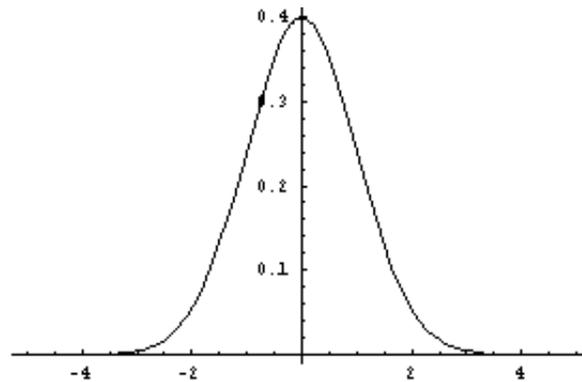


Figure 14.9: The Gauss-Laplace (Gaussian) Distribution,  $xmu = 0$ ,  $sigma = 1$ .

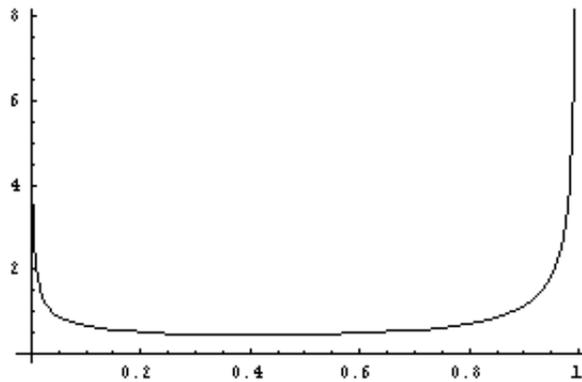


Figure 14.10: The Beta Distribution,  $alpha = .5$ ,  $beta = .25$ .

`beta-dist(a, b)` [SAL]

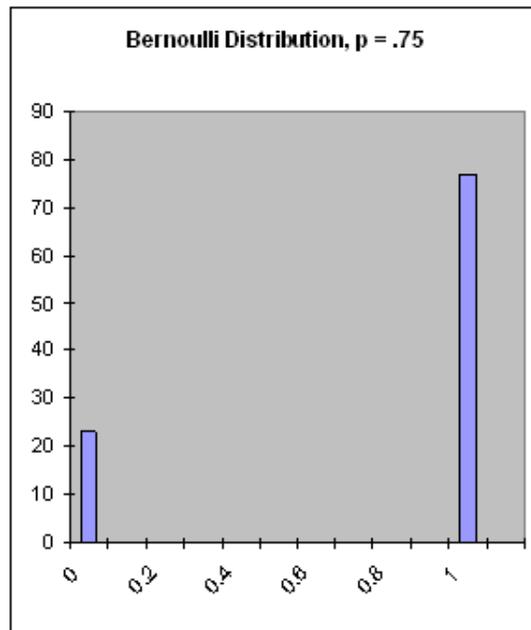
`(beta-dist a b)` [LISP]

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 symmetric about  $1/2$  when  $a = b$ .

`bernoulli-dist(px1 [, x1, x2])` [SAL]

`(bernoulli-dist px1 [x1 x2])` [LISP]

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 viewed as a success while  $x2$  is viewed as a failure.

Figure 14.11: The Bernoulli Distribution,  $p = .75$ .

`binomial-dist(n, p)` [SAL]

`(binomial-dist n p)` [LISP]

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)` [SAL]

`(geometric-dist p)` [LISP]

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)` [SAL]

`(poisson-dist delta)` [LISP]

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.

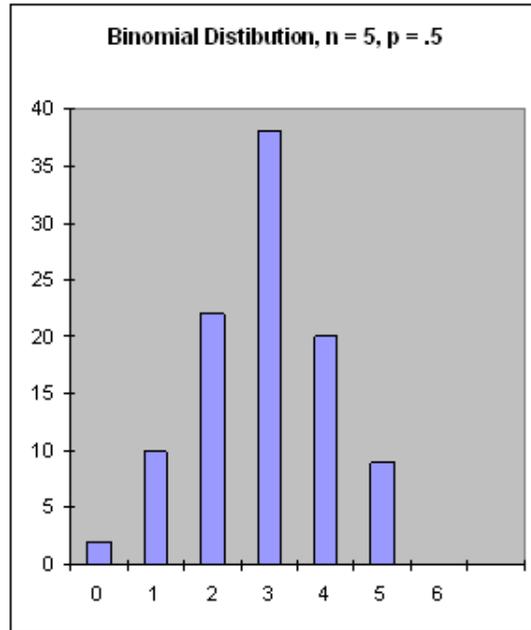


Figure 14.12: The Binomial Distribution,  $n = 5, p = .5$ .

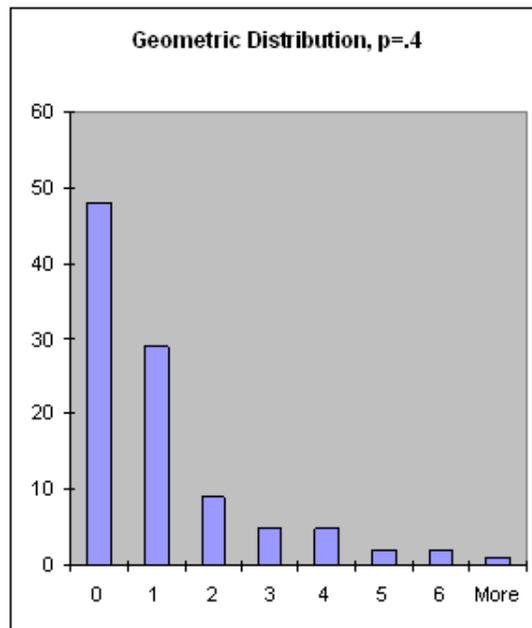
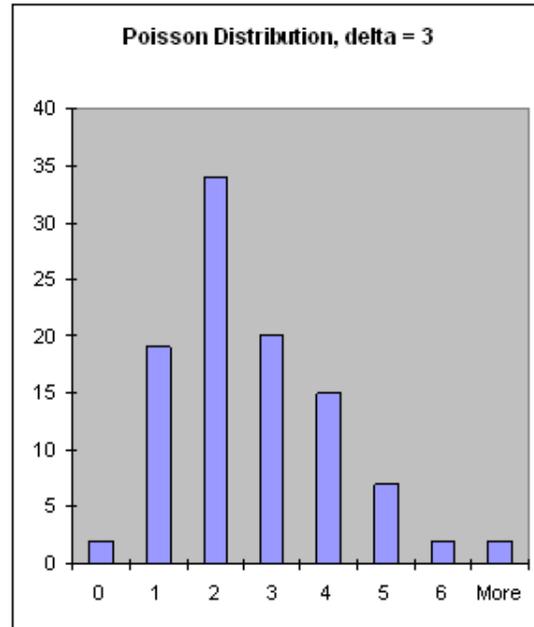


Figure 14.13: The Geometric Distribution,  $p = .4$ .

Figure 14.14: The Poisson Distribution,  $\delta = 3$ .

## 14.4 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.

### 14.4.1 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 C 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:

```
k-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.

### 14.4.2 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, ...)` [SAL]

`(score-gen :k1 e1 :k2 e2 ...)` [LISP]

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 questions and answers below.

*How many notes will be generated?* The keyword parameter `score-len`: specifies an upper bound on the number of notes. (Note: in LISP syntax, keywords are always *preceded* by colons, so you would write `:score-len` instead.) 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:start` 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:start` 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.

### 14.4.3 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)` [SAL]

`(event-time event)` [LISP]  
Retrieve the time field from an event.

`event-set-time(event, time)` [SAL]

`(event-set-time event time)` [LISP]  
Construct a new event where the time of *event* is replaced by *time*.

`event-dur(event)` [SAL]

`(event-dur event)` [LISP]  
Retrieve the duration (i.e. the stretch factor) field from an event.

`event-set-dur(event, dur)` [SAL]

`(event-set-dur event dur)` [LISP]  
Construct a new event where the duration (or stretch factor) of *event* is replaced by *dur*.

`event-expression(event)` [SAL]

`(event-expression event)` [LISP]  
Retrieve the expression field from an event.

`event-set-expression(event, dur)` [SAL]

`(event-set-expression event dur)` [LISP]  
Construct a new event where the expression of *event* is replaced by *expression*.

`event-end(event)` [SAL]

`(event-end event)` [LISP]  
Retrieve the end time of *event*, its time plus its duration.

`expr-has-attr(expression, attribute)` [SAL]

`(expr-has-attr expression attribute)` [LISP]  
Test whether a score event *expression* has the given *attribute*.

`expr-get-attr(expression, attribute [, default])` [SAL]  
`(expr-get-attr expression attribute [default])` [LISP]  
 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)` [SAL]  
`(expr-set-attr expr attribute value)` [LISP]  
 Construct a new expression identical to *expr* except that the *attribute* has *value*.

`event-has-attr(event, attribute)` [SAL]  
`(event-has-attr event attribute)` [LISP]  
 Test whether a given score *event*'s expression has the given *attribute*.

`event-get-attr(event, attribute, [default])` [SAL]  
`(event-get-attr event attribute [default])` [LISP]  
 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)` [SAL]  
`(event-set-attr event attribute value)` [LISP]  
 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, where 1 (not zero) denotes the first note. If indices are in range, the number of note selected is `to-index - from-index + 1`. Out-of-range indices are ignored. 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. (Note: in LISP syntax, colons *precede* the keyword, so use `:from-index`, `:to-index`, `:from-time`, and `:to-time`.)

`score-sorted(score)` [SAL]  
`(score-sorted score)` [LISP]  
 Test if *score* is sorted.

`score-sort(score [, copy-flag])` [SAL]  
`(score-sort score [copy-flag])` [LISP]  
 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)` [SAL]  
`(score-shift score offset :from-index i :to-index j :from-time x :to-time y)` [LISP]  
 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 decreased if necessary to the minimum time of any event that is moved to an earlier time (by a negative *offset*), and the end time of the score is increased if necessary to the maximum end time of any event that is moved to a later time. If all shifted events remain within the score's begin-to-end range, the begin and end times are not changed. 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) [SAL]
```

```
(score-stretch score factor :dur dur-flag :time time-flag
  :from-index i :to-index j :from-time x :to-time y) [LISP]
```

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) [SAL]
```

```
(score-transpose score keyword amount :from-index i :to-index j
  :from-time x :to-time y) [LISP]
```

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 transpose up by a whole step, write (score-transpose 2 :pitch *score*). The original score is not modified, and a new score is returned. If *keyword* is :pitch and a corresponding parameter value is a list, each element of the list is incremented by *amount*. This special case is in keeping with the convention of timed-seq in which score events with lists for the :pitch attribute are expanded into "chords" by instantiating an event for each element (pitch) in the list (chord).

```
score-scale(score, keyword, amount, from-index: i, to-index: j,
  from-time: x, to-time: y) [SAL]
```

```
(score-scale score keyword amount :from-index i :to-index j
  :from-time x :to-time y) [LISP]
```

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) [SAL]
```

```
(score-sustain score factor :from-index i :to-index j :from-time x :to-time y) [LISP]
```

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) [SAL]
```

```
(score-voice score replacement-list :from-index i :to-index j
  :from-time x :to-time y) [LISP]
```

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, ...)` [SAL]

`(score-merge score1 score2 ...)` [LISP]

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, ...)` [SAL]

`(score-append score1 score2 ...)` [LISP]

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, extract: ex)` [SAL]

`(score-select score predicate :from-index i :to-index j`

`:from-time x :to-time y :reject flag :extract ex)` [LISP]

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. Note that `from-index: i` and `to-index: j` are 1-based: A value of 1 refers to the first note, etc., because the zeroth element of the score is a SCORE-BEGIN-END event. For consistency, if a SCORE-BEGIN-END event is missing from *score*, one is inserted before any further processing. The selected notes are combined to form a new score. By default, *ex*, the value of the `extract:` keyword, is false, and the begin and end markers are retained from *score*. Conceptually, the score and its timing are retained as an object, but only some of its constituent sound events are retained. Otherwise, if *ex* is true (non-nil), the begin and end times are computed based on the selected (extracted) events: The begin time is `from-time x`, if present, and otherwise the time of the first event (if any), and otherwise the begin time of *score*. Similarly, the end time is the `to-time y`, if present, and otherwise the end time of the last event (if any), and otherwise the new start time. 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 begin and end times of *score* are retained and the `extract` parameter (*ex*) is ignored. In all cases, the original score is not modified, and a new score is returned.

`score-set-begin(score, time)` [SAL]

`(score-set-begin score time)` [LISP]

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)` [SAL]

`(score-get-begin score)` [LISP]

Return the begin time of the *score*.

`score-set-end(score, time)` [SAL]

`(score-set-end score time)` [LISP]

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)` [SAL]  
`(score-get-end score)` [LISP]  
 Return the end time of the *score*.

`score-must-have-begin-end(score)` [SAL]  
`(score-must-have-begin-end score)` [LISP]  
 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 original score is not modified.

`score-filter-length(score, cutoff)` [SAL]  
`(score-filter-length score cutoff)` [LISP]  
 Remove notes that extend beyond the *cutoff* time. This is similar to `score-select`, but 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)` [SAL]  
`(score-repeat score n)` [LISP]  
 Make a sequence of *n* copies of *score*. Each copy is shifted so that its 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)` [SAL]  
`(score-stretch-to-length score length)` [LISP]  
 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)` [SAL]  
`(score-filter-overlap score)` [LISP]  
 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, [lines])` [SAL]  
`(score-print score [lines])` [LISP]  
 Print a score with one note per line. Returns `nil`. If *lines* (optional FIXNUM) is given, print a maximum of that many lines (but the minimum is at least 3). The format is first *lines*-2 score events, the line "...", and the last score event.

`score-play(score)` [SAL]  
`(score-play score)` [LISP]  
 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)` [SAL]  
`(score-adjacent-events score function :from-index i :to-index j :from-time x :to-time y)` [LISP]  
 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)` [SAL]  
`(score-apply score function :from-index i :to-index j :from-time x :to-time y)` [LISP]  
 Replace each note in the score with the result of `(function time dur expression)` (in Lisp) or `function(time, dur, expression)` (in SAL), 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)` [SAL]  
`(score-indexof score function :from-index i :to-index j :from-time x :to-time y)` [LISP]  
 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)` [SAL]  
`(score-last-indexof score function :from-index i :to-index j :from-time x :to-time y)` [LISP]  
 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)` [SAL]  
`(score-randomize-start score amt :from-index i :to-index j :from-time x :to-time y)` [LISP]  
 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.

#### 14.4.4 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 11. It is also useful sometimes to read notes from Standard MIDI Files into Xmusic scores and vice versa.

Using `score-read-smf` and `score-read`, 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.

The `score-from-seq` function allows you to obtain control changes and other MIDI messages.

`score-read-smf(filename)` [SAL]  
`(score-read-smf filename)` [LISP]  
 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-read(filename)` [SAL]

`(score-read filename)` [LISP]

Read an Adagio file from *filename*. Return an Xmusic score, or `nil` if the file could not be opened. See Chapter 11 for details on Adagio, a text-based score language. See `score-read-smf` for details on handling program changes.

`score-from-seq(seq, prog: pflag, synths: synths,  
bend: bend, cpress: cpress, ctrl: ctrls)` [SAL]

`(score-from-seq seq :prog pflag :synths synths  
:bend bend :cpress cpress :ctrl ctrls)` [LISP]

Produce a score from a sequence (see Section 11.1) type *seq*. The optional *pflag*, if non-null, will insert program changes as event `:prog` attributes in score events. The *bend* and *cpress* (channel pressure) values may be `:onset` to introduce `:bend` or `:cpress` attributes into score events, or `:contin` to encode these MIDI messages as SOUNDS available through event `:contin` attributes. The *ctrl* parameter is a list where the first element is either `:onset` or `:contin` and the remaining elements are controller numbers to be encoded. In all cases `:contin` values appear in score events as an object. You can access the SOUNDS that encode control changes using the functions `ctrlfn-bend(contin)`, `ctrlfn-cpress(contin)` or `ctrlfn-ctrl(contin, number)`, which will return `const(0)` if no corresponding MIDI messages were found. (As a special case the default for controller 7 (volume pedal) is `const(1)`). See `lib/midi/midi_tutorial.htm` for more details and code examples. To test for the presence of MIDI messages and avoid the construction of `const(0)`, use `ctrlfn-bend?(contin)`, `ctrlfn-cpress?(contin)` or `ctrlfn-ctrl?(contin, number)`.

`score-write-smf(score, filename, [programs as-adagio])` [SAL]

`(score-write-smf score filename [programs as-adagio])` [LISP]

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.* The *as-adagio* optional parameter should normally be omitted. If non-`nil`, the file is written in Adagio format, but if you want to do that, call `score-write` instead. Xmusic scores do not specify tempo, so the MIDI file is written with a fixed tempo of 100bpm. If you create scores or stretch scores so that each beat is exactly 0.6s (100bpm), sequencers and score editors will quantize your scores correctly. Otherwise, the *timing* will be correct, but for example a score with one note every second will be notated as 1 note every  $1 \frac{2}{3}$  beats.

`score-write(score, filename, [programs, absolute])` [SAL]

`(score-write score filename [programs absolute])` [LISP]

Write an Adagio format file to *filename* with notes in *score*, using absolute times if *absolute* is true, otherwise write relative times (the default). See Chapter 11 for details on Adagio, a text-based score language. See `score-write-smf` for details on MIDI program changes.

## 14.4.5 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)` [SAL]

`(add-to-workspace symbol)` [LISP]

Adds a global variable to the workspace. The *symbol* should be a (quoted) symbol.

`save-workspace()` [SAL]

`(save-workspace)` [LISP]

All global variables in the workspace are saved to `workspace.lsp` (in the current directory), overwriting the previous file.

`describe(symbol [, description])` [SAL]

`(describe symbol [description])` [LISP]

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)` [SAL]

`(add-action-to-workspace symbol)` [LISP]

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`.

## 14.4.6 Utility Functions

This chapter concludes with details of various utility functions for score manipulation.

`patternp(expression)` [SAL]

`(patternp expression)` [LISP]

Test if *expression* is an Xmusic pattern.

`params-transpose(params, keyword, amount)` [SAL]

`(params-transpose params keyword amount)` [LISP]

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)` [SAL]

`(params-scale params keyword amount)` [LISP]

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)` [SAL]

`(interpolate x x1 y1 x2 y2)` [LISP]

Linearly interpolate (or extrapolate) between points  $(x1, y1)$  and  $(x2, y2)$  to compute the *y* value corresponding to *x*.

`intersection(a, b)` [SAL]

`(intersection a b)` [LISP]

Compute the set intersection of lists *a* and *b*.

`union(a, b)` [SAL]

`(union a b)` [LISP]

Compute the set union of lists *a* and *b*.

`set-difference(a, b)` [SAL]

`(set-difference a b)` [LISP]

Compute the set of all elements that are in *a* but not in *b*.

`subsetp(a, b)` [SAL]

`(subsetp a b)` [LISP]

Returns true iff *a* is a subset of *b*, that is, each element of *a* is a member of *b*.

## 15 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. Many of these functions and files are not included in the main distribution of Nyquist. Instead, they are extensions (see Section 2.4). Extensions are just like any other library except you must first install them with the Extension Manager in the NyquistIDE, and each extension is in a separate folder within the `lib` folder.

For example, to get statistics functions, load `"statistics"`, but to get the phase vocoder test functions, first install the `pvoc` extension, and then load `"pvoc/phasevocoder.sal"`. In many cases, functions within an extension will “autoload,” e.g. after you install the `labels` extension, the first time you call `read-labels` or `write-labels`, `labels.sal` will be loaded automatically to define these functions, as if the function had been loaded all along. Extensions also update the function call quick reference and completion feature in the NyquistIDE.

Many libraries are documented in one of the following sections, but see the Extension Manager for possible new additions. Extensions have documentation, which you can find using the Extension Manager.

### 15.1 Statistics

The file `statistics.lsp` defines a class and functions to compute simple statistics, histograms, correlation, and some other tests. See the source code for complete details.

### 15.2 Plots

The Nyquist IDE has a simple facility to plot signals. For more advanced plotting, you can use `gnuplot.sal` (load `"gnuplot.sal"`) to generate plots for `gnuplot`, a separate, but free program. See the source for details.

### 15.3 Labeling Audio Events, Marking Audio Times, Displaying Marked Audio Times

The `labels.sal` program can convert lists to label files and label files to lists. Label files can be loaded along with audio in Audacity to show metadata. See the source for details.

## 15.4 Linear Regression

See the `regression` extension and its `regression.sal` for simple linear regression functions.

## 15.5 Vector Math, Linear Algebra

See `vectors.lisp` and `load "vectors"` for a simple implementation of vector arithmetic and other vector functions. These “vectors” are implemented as lists, but there are functions to convert to and from arrays.

## 15.6 JSON Input and Output

JSON is widely used in web technology, scientific computing and many other areas. See `json.sal` and `load "json"` for a simple implementation of JSON input and output. On input, JSON dictionaries are represented as association lists and JSON arrays are represented as XLisp arrays. Data can be retrieved from dictionaries using `json-field`. If you are writing in Lisp, note that you can call `(sal-load "json")` to load and compile `json.sal` so that you can call its functions from Lisp.

For output, you can either write a structure consisting of nested arrays and dictionaries (association lists), or you can call a sequence of functions to, for example, begin writing an array, write each element of the array and finish writing the array. This way, you can write JSON as the data is computed rather than building a monolithic structure in Lisp and then writing it.

Note that Lisp is already able to read and write structured data, and if everyone used Lisp and SAL, there might not be any need for JSON. Nyquist scores are an example of data you can save to files in a text format and read back in. So JSON is not recommended except for interoperation with other non-Lisp systems.

## 15.7 Piano Synthesizer

The piano synthesizer (library name is `pianosyn.lisp`) generates realistic piano tones using a multiple wavetable implementation by Zheng (Geoffrey) Hua and Jim Beauchamp, University of Illinois. Detailed acknowledgements print when you load the file. Further information and example code can be found in `nyquist/lib/piano/piano.htm`. There are several useful functions in this library. These functions auto-load the `pianosyn.lisp` file if it is not already loaded:

```
piano-note(duration, step, dynamic) [SAL]
```

```
(piano-note duration step dynamic) [LISP]
```

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*) [SAL]

(`piano-note-2` *step* *dynamic*) [LISP]

Similar to `piano-note` except the duration is nominally 1.0.

`piano-midi`(*midi-file-name*) [SAL]

(`piano-midi` *midi-file-name*) [LISP]

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*) [SAL]

(`piano-midi2file` *midi-file-name* *sound-file-name*) [LISP]

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*.

## 15.8 Dynamics Compression

These functions in the `compress` extension implement 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 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-threshold*,  
*limit: limit*, *transition: transition*, *verbose: verbose*) [SAL]

(`compress-map` *compress-ratio* *compress-threshold* *expand-ratio* *expand-threshold*  
:*limit limit* :*transition transition* :*verbose verbose*) [LISP]

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-threshold* 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.

If `verbose` is true (this is the default), the map is printed, showing, for each dB value below zero of this input, what is the gain (in dB) indicated by the output. Only regions where the map is changing are printed because at lower values, the dB gain is constant.

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, min: mindb)` [SAL]

`(db-average input :min mindb)` [LISP]

Compute the average amplitude of `input` in dB. The result is a sound at a rate of about 40Hz based on RMS of `input` such that 0 (dB) corresponds to a sinusoid with a peak amplitude of 1. This is the same dB estimate that is used in the `compress` function. If `mindb` is specified and non-nil, the result samples below this value will be replaced by `mindb`. For example, if you plot the `db-average` curve of an instrument with a range from -3 dB down to -30 dB, but silences exist at -90 dB, then most of the interesting values will be squeezed into the top 1/3 of a plot. Instead, if you specify `min` to be -30, then the interesting range will span the entire plot (and silences will show up at -30 dB, the value of `min`).

`compress(input, map, rise-time, fall-time [, lookahead])` [SAL]

`(compress input map rise-time fall-time [lookahead])` [LISP]

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])` [SAL]

`(agc input range rise-time fall-time [lookahead])` [LISP]

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`.

## 15.9 Clipping Softener

The `clipsoften` extension was written to improve the quality of poorly recorded speech. In recordings of speech, extreme clipping generates harsh high frequency noise. This can sound particularly 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)` [SAL]

`(soften-clipping snd cutoff)` [LISP]

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).

## 15.10 Graphical Equalizer

The library (`load "grapeq.lsp"`) works with the NyquistIDE's Equalizer window (see Section 2.7, but this library can be used directly in Nyquist programs 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)` [SAL]

`(nband-range input gains lowf highf)` [LISP]

A graphical equalizer applied to *input* (a SOUND). The gain controls and number of bands is given by *gains*, an ARRAY of SOUNDS (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 FLONUMs).

`nband(input, gains)` [SAL]

`(nband input gains)` [LISP]

A graphical equalizer, identical to `nband-range` with a range of 20 to 20,000 Hz.

## 15.11 Sound Reversal

The `reverse` extension implements functions to play sounds in reverse.

`s-reverse(snd)` [SAL]

`(s-reverse snd)` [LISP]

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, sr: sr, dur: dur, nchans: chans, format: format, mode: mode, bits: n, swap: flag)` [SAL]

`(s-read-reverse filename :time-offset offset :sr sr :dur dur`

`:nchans chans :format format :mode mode :bits n :swap flag)` [LISP]

This function is identical to `s-read` (see Section 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.

## 15.12 Time Delay Functions

The `time-delay-fns.lsp` library implements chorus, phaser, and flange effects.

`phaser(snd)` [SAL]

`(phaser snd)` [LISP]

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)` [SAL]

`(flange snd)` [LISP]

A flange effect applied to *snd*. To vary the rate and other parameters, see the source code.

`stereo-chorus(snd, delay: delay, depth: depth, rate1: rate1, rate2: rate2  
saturation: saturation)` [SAL]

`(stereo-chorus snd :delay delay :depth depth :rate1 rate1 :rate2 rate2  
:saturation saturation)` [LISP]

A chorus effect applied to *snd*, a SOUND (monophonic). The output is a stereo sound with out-of-phase chorus effects applied separately for the left and right channels. See the chorus function below for a description of the optional parameters. The *rate1* and *rate2* parameters are *rate* parameters for the left and right channels.

`chorus(snd, delay: delay, depth: depth, rate: rate, saturation: saturation, phase: phase)`  
[SAL]

`(chorus snd :delay delay :depth depth :rate rate :saturation saturation :phase phase)`  
[LISP]

A chorus effect applied to *snd*. All parameters may be arrays as usual. The chorus is implemented as a variable delay modulated by a sinusoid shifted by *phase* degrees (a FLONUM) oscillating at *rate* Hz (a FLONUM). The sinusoid is scaled by *depth* (a FLONUM). The delayed signal is mixed with the original, and *saturation* (a FLONUM) gives the fraction of the delayed signal (from 0 to 1) in the mix. Default values are *delay* 0.03, *depth* 0.003, *rate* 0.3, *saturation* 1.0, and *phase* 0.0 (degrees).

## 15.13 Multiple Band Effects

The `bandfx` extension 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. Further discussion and examples can be found in `nyquist/lib/bandfx/bandfx.lsp`.

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 multichannel 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 ([dannenbergs@cs.cmu.edu](mailto:dannenbergs@cs.cmu.edu)) if you are interested in working on this library.

`apply-banded-delay(s, lowp, highp, num-bands, lowd, highd, fb, wet)` [SAL]

`(apply-banded-delay s lowp highp num-bands lowd highd fb wet)` [LISP]

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)` [SAL]

`(apply-banded-bass-boost s lowp highp num-bands num-boost gain)` [LISP]

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)` [SAL]

`(apply-banded-treble-boost s lowp highp num-bands num-boost gain)` [LISP]

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.

## 15.14 Granular Synthesis

Some granular synthesis functions are implemented in the `gran` extension. 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])` [SAL]

`(sf-granulate filename grain-dur grain-dev ioi ioi-dev pitch-dev  
[file-start file-end])` [LISP]

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`.)

## 15.15 Chowning FM Voices

John Chowning developed voice synthesis methods using FM to simulate resonances for his 1981 composition "Phone." He later recreated the synthesis algorithms in Max, and Jorge Sastre ported these to SAL. See `fm-voices-chowning` extension and documentation for more details.

## 15.16 Atonal Melody Composition

Jorge Sastre contributed the `atonal-melodies` extension that generates atonal melodies. You can find links to an example score and audio file in the code and also at <http://algocompbook.com/examples.html>.

## 15.17 MIDI Utilities

The `midishow.lsp` library has functions that can print the contents of MIDI files. This intended as a debugging aid.

`midi-show-file(file-name)` [SAL]

`(midi-show-file file-name)` [LISP]

Print the contents of a MIDI file to the console.

`midi-show(the-seq [, out-file])` [SAL]

`(midi-show the-seq [out-file])` [LISP]

Print the contents of the sequence *the-seq* to the file *out-file* (whose default value is the console.)

## 15.18 Reverberation

The `reverb.lsp` library implements artificial reverberation.

`reverb(snd, time)` [SAL]

`(reverb snd time)` [LISP]

Artificial reverberation applied to *snd* with a decay time of *time*.

## 15.19 DTMF Encoding

The `dtmf` extension implements DTMF encoding. DTMF is the “touch tone” code used by telephones.

`dtmf-tone(key, len, space)` [SAL]

`(dtmf-tone key len space)` [LISP]

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 tone is given by *len* (a FLONUM) and the tone is followed by silence of duration *space* (a FLONUM).

`speed-dial(thelist)` [SAL]

`(speed-dial thelist)` [LISP]

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.

## 15.20 Dolby Surround(R), Stereo and Spatialization Effects

The `spatial` extension 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 and extension documentation for more information.

`stereoize(snd)` [SAL]

`(stereoize snd)` [LISP]

Convert a mono sound, *snd*, to stereo. Four bands of equalization and some delay are used to create a stereo effect.

`widen(snd, amt)` [SAL]

`(widen snd amt)` [LISP]

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)` [SAL]

`(span snd amt)` [LISP]

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)` [SAL]

`(swapchannels snd)` [LISP]

Swap left and right channels in *snd*, a stereo sound.

`prologic(l, c, r, s)` [SAL]

`(prologic l c r s)` [LISP]

Encode four monaural SOUNDS 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)` [SAL]

`(pl-left snd)` [LISP]

Produce a Dolby-encoded (stereo) signal with *snd*, a SOUND, encoded as the front left channel.

`pl-center(snd)` [SAL]

`(pl-center snd)` [LISP]

Produce a Dolby-encoded (stereo) signal with *snd*, a SOUND, encoded as the front center channel.

`pl-right(snd)` [SAL]

`(pl-right snd)` [LISP]

Produce a Dolby-encoded (stereo) signal with *snd*, a SOUND, encoded as the front right channel.

`pl-rear(snd)` [SAL]

`(pl-rear snd)` [LISP]

Produce a Dolby-encoded (stereo) signal with *snd*, a SOUND, encoded as the rear, or surround, channel.

`pl-pan2d(snd, x, y)` [SAL]

`(pl-pan2d snd x y)` [LISP]

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 SOUNDS. 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)` [SAL]

`(pl-position snd x y config)` [LISP]

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 FLONUMs or SOUNDS. 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)` [SAL]

`(pl-doppler snd r)` [LISP]

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).

## 15.21 Drum Machine

The drum machine software in the `plight` extension deserves further explanation. There is documentation associated with the extension and available in the NyquistIDE Extension Manager. To use the software, install the extension and load the code by evaluating:

```
load "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 `nyquist/lib/plight/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 `nyquist/lib/plight/plight/drums.lsp` file contains an example function `plight-drum-example` to play some drums. There is also the file `nyquist/lib/plight/beats.props` to serve as an example of how to specify sound files and beat patterns.

`drum(tracknum, patternnum, bpm)` [SAL]

`(drum tracknum patternnum bpm)` [LISP]

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)` [SAL]

`(drum-loop snd duration numtimes)` [LISP]

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)` [SAL]

`(length-of-beat bpm)` [LISP]

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.

## Bibliography

- [1] R. B. Dannenberg and C. L. Fraley. "Fugue: Composition and Sound Synthesis with Lazy Evaluation and Behavioral Abstraction." In *Proceedings of the International Computer Music Conference*, T. Wells and D. Butler (eds.), San Francisco: International Computer Music Association, 1989, pp. 76-79.
- [2] M. Simoni and R. B. Dannenberg. *Algorithmic Composition: A Guide to Composing Music with Nyquist*. Ann Arbor: University of Michigan Press. 2013.
- [3] D. S. Touretzky. *LISP: A Gentle Introduction to Symbolic Computation*. New York: Harper and Row. 1984.

## Appendix A 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.)

### A.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.

### A.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)`.

## A.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.

## A.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.

## A.5 Writing a .alg File

**WARNING:** Translation relies heavily on string substitution, which

is fragile. In particular, variables with names that are substrings of other variables will cause problems. For example if you declare STATE variables "phase" and "iphase", then the translator will globally substitute "phase\_reg" for "phase", converting "phase" to "phase\_reg" and "iphase" to "iphase\_reg". Then it will substitute "iphase\_reg" for "iphase" which will convert the existing "iphase\_reg" to "iphase\_reg\_reg". This will be confusing and will not compile. (End of WARNING)

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.lisp`.) 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.

## A.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 pseudo-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 auxiliary 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 auxiliary 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 deallocated. 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 flag)**

controls when 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. If inline interpolation is enabled, Nyquist generates various inner loops and selects the appropriate one at run-time. (This can cause a combinatorial explosion if there are multiple sound arguments.) If inline interpolation is not enabled, much less code is generated and interpolation is performed as necessary by instantiating a separate signal processing operation. The value of *flag* is YES to generate inline interpolation, NO to disable inline interpolation, and NIL to take the default set by the global variable \*INLINE-INTERPOLATION\*. The default is also taken if no *INLINE-INTERPOLATION* attribute is specified.

**(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 performing 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.

## A.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.

## A.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 B 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.

### B.1 Overview

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`.

### B.1.1 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.

## B.2 Header file format

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

```
type-name routine-name (); /* LISP: (func-name type1 type2 ...) */
```

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.

## B.3 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 \*.

## B.4 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.lisp* and append

```
(any lisp expression)
```

to *name.lisp*, 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.lisp* file is generated. *Note: the LISP-SRC comment must be on a new line.*

## B.5 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) */
```

## B.6 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 C    **XLISP: An Object-oriented Lisp**

**Version 2.0**

February 6, 1988

by

**David Michael Betz**

127 Taylor Road

Peterborough, NH 03458

Copyright (c) 1988, by David Michael Betz

All Rights Reserved

Permission is granted for unrestricted non-commercial use

## **C.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.

## C.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.

## C.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.

## C.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.

## C.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.

## C.6 Data Types

There are several different data types available to XLISP programmers.

- lists - are linked lists of `cons` cells, where a `cons` cell is a pair of references called the `car` or `head` and `cdr` or `tail`. The empty list is the null reference denoted by `nil`. (There is no `cons` cell.) A list of 1 element is a `cons` cell whose `car` is the element and whose `cdr` is a list of no elements, i.e. `nil`. A list of 2 elements is a `cons` cell whose `car` is the first element and whose `cdr` is a list containing the second element, and so on.
- symbols - unique strings that serve as names of variables and names of functions. All symbols are in a global symbol table.
- strings - non-unique, immutable character strings.
- integers
- characters - note that characters are not strings.
- floats

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

## C.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.

## C.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

## C.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 occurrence 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>` == literal of type character
- `#\Newline` == newline character
- `#\Space` == space character
- `#\Tab` == tab character
- `#| ... |#` == a comment
- `#:<symbol>` == an uninterned symbol
- `'<expr>` == (backquote <expr>)
- `,<expr>` == (comma <expr>)
- `,@<expr>` == (comma-at <expr>)

## C.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

## C.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.

## C.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 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

## C.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.

## C.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.

## C.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.

## C.16 Evaluation Functions

`eval(expr)` [SAL]

(`eval expr`) [LISP] – evaluate an xlist expression  
*expr* – the expression to be evaluated  
 returns – the result of evaluating the expression

`apply(fun, args)` [SAL]

(`apply fun args`) [LISP] – 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 ...)` [SAL]

(`funcall fun arg ...`) [LISP] – 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)` [SAL]

(`quote expr`) [LISP] – return an expression unevaluated  
*expr* – the expression to be quoted (quoted)  
 returns – *expr* unevaluated

(`function expr`) [LISP] – get the functional interpretation. Note that in SAL, the function can be accessed as

`#function.`  
*expr* – the symbol or lambda expression (quoted)  
 returns – the functional interpretation

backquote(*expr*) [SAL]  
 (backquote *expr*) [LISP] – fill in a template  
   *expr* – the template  
   returns – a copy of the template with comma and comma-at  
   expressions expanded

lambda(*args*, *expr* ...) [SAL]  
 (lambda *args expr* ...) [LISP] – 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*) [SAL]  
 (get-lambda-expression *closure*) [LISP] – get the lambda expression  
   *closure* – the closure  
   returns – the original lambda expression

macroexpand(*form*) [SAL]  
 (macroexpand *form*) [LISP] – recursively expand macro calls  
   *form* – the form to expand  
   returns – the macro expansion

macroexpand-1(*form*) [SAL]  
 (macroexpand-1 *form*) [LISP] – expand a macro call  
   *form* – the macro call form  
   returns – the macro expansion

## C.17 Symbol Functions

(set *sym expr*) [LISP] – set the value of a symbol. Note that in SAL, the function can be accessed as #set.  
   *sym* – the symbol being set  
   *expr* – the new value  
   returns – the new value

setq([*sym*, *expr*] ...) [SAL]  
 (setq [*sym expr*] ...) [LISP] – set the value of a symbol. Note that in SAL, the set command is normally used.  
   *sym* – the symbol being set (quoted)  
   *expr* – the new value  
   returns – the new value

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

setf([*place*, *expr*] ...) [SAL]  
 (setf [*place expr*] ...) [LISP] – 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 ...*) [LISP] – define a function  
(defmacro *sym fargs expr ...*) [LISP] – 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*]) [SAL]  
(gensym [*tag*]) [LISP] – generate a symbol  
*tag* – string or number  
returns – the new symbol

intern(*pname*) [SAL]  
(intern *pname*) [LISP] – make an interned symbol  
*pname* – the symbol’s print name string  
returns – the new symbol

make-symbol(*pname*) [SAL]  
(make-symbol *pname*) [LISP] – make an uninterned symbol  
*pname* – the symbol’s print name string  
returns – the new symbol

symbol-name(*sym*) [SAL]  
(symbol-name *sym*) [LISP] – get the print name of a symbol  
*sym* – the symbol  
returns – the symbol’s print name

symbol-value(*sym*) [SAL]  
(symbol-value *sym*) [LISP] – get the value of a symbol  
*sym* – the symbol  
returns – the symbol’s value

symbol-function(*sym*) [SAL]  
(symbol-function *sym*) [LISP] – get the functional value of a symbol  
*sym* – the symbol  
returns – the symbol’s functional value

symbol-plist(*sym*) [SAL]  
(symbol-plist *sym*) [LISP] – get the property list of a symbol  
*sym* – the symbol  
returns – the symbol’s property list

`hash(sym, n)` [SAL]  
(`hash sym n`) [LISP] – compute the hash index for a symbol  
*sym* – the symbol or string  
*n* – the table size (integer)  
returns – the hash index (integer)

## C.18 Property List Functions

`get(sym, prop)` [SAL]  
(`get sym prop`) [LISP] – get the value of a property  
*sym* – the symbol  
*prop* – the property symbol  
returns – the property value or `nil`

`putprop(sym, val, prop)` [SAL]  
(`putprop sym val prop`) [LISP] – 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)` [SAL]  
(`remprop sym prop`) [LISP] – remove a property  
*sym* – the symbol  
*prop* – the property symbol  
returns – `nil`

## C.19 Array Functions

`aref(array, n)` [SAL]  
(`aref array n`) [LISP] – get the *n*th element of an array  
*array* – the array  
*n* – the array index (integer)  
returns – the value of the array element

`make-array(size)` [SAL]  
(`make-array size`) [LISP] – make a new array  
*size* – the size of the new array (integer)  
returns – the new array

`vector(expr ...)` [SAL]  
(`vector expr ...`) [LISP] – make an initialized vector  
*expr* – the vector elements  
returns – the new vector

## C.20 List Functions

`car(expr)` [SAL]  
(`car expr`) [LISP] – return the car of a list node  
*expr* – the list node  
returns – the car of the list node

`cdr(expr)` [SAL]  
(`cdr expr`) [LISP] – return the cdr of a list node  
*expr* – the list node  
returns – the cdr of the list node

`cxxr(expr)` [SAL]  
(`cxxr expr`) [LISP] – all cxxr combinations

`cxrxr(expr)` [SAL]  
(`cxrxr expr`) [LISP] – all cxrxr combinations

`cxxxxr(expr)` [SAL]  
(`cxxxxr expr`) [LISP] – all cxxxxr combinations

`first(expr)` [SAL]  
(`first expr`) [LISP] – a synonym for car

`second(expr)` [SAL]  
(`second expr`) [LISP] – a synonym for cadr

`third(expr)` [SAL]  
(`third expr`) [LISP] – a synonym for caddr

`fourth(expr)` [SAL]  
(`fourth expr`) [LISP] – a synonym for caddr

`rest(expr)` [SAL]  
(`rest expr`) [LISP] – a synonym for cdr

`cons(expr1, expr2)` [SAL]  
(`cons expr1 expr2`) [LISP] – construct a new list node  
*expr1* – the car of the new list node  
*expr2* – the cdr of the new list node  
returns – nil

`list(expr ...)` [SAL]  
(`list expr ...`) [LISP] – create a list of values  
*expr* – expressions to be combined into a list  
returns – the new list

`append(expr ...)` [SAL]  
(`append expr ...`) [LISP] – append lists  
*expr* – lists whose elements are to be appended  
returns – the new list

`reverse(expr)` [SAL]  
(`reverse expr`) [LISP] – reverse a list  
*expr* – the list to reverse  
returns – a new list in the reverse order

`last(list)` [SAL]  
(`last list`) [LISP] – return the last list node of a list  
*list* – the list  
returns – the last list node in the list

`member(expr, list, test: test, test-not: test-not)` [SAL]  
 (member *expr list* &key :test :test-not) [LISP] – 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, test: test, test-not: test-not)` [SAL]  
 (assoc *expr alist* &key :test :test-not) [LISP] – 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, test: test, test-not: test-not)` [SAL]  
 (remove *expr list* &key :test :test-not) [LISP] – 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)` [SAL]  
 (remove-if *test list*) [LISP] – 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)` [SAL]  
 (remove-if-not *test list*) [LISP] – remove elements that fail test  
     *test* – the test predicate  
     *list* – the list  
     returns – copy of list with non-matching elements removed

`length(expr)` [SAL]  
 (length *expr*) [LISP] – 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)` [SAL]  
 (nth *n list*) [LISP] – 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)` [SAL]  
 (nthcdr *n list*) [LISP] – 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 ...)` [SAL]  
`(mapc fcn list1 list ...)` [LISP] – 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 ...)` [SAL]  
`(mapcar fcn list1 list ...)` [LISP] – 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 ...)` [SAL]  
`(mapl fcn list1 list ...)` [LISP] – 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 ...)` [SAL]  
`(maplist fcn list1 list ...)` [LISP] – 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, test: test, test-not: test-not)` [SAL]  
`(subst to from expr &key :test :test-not)` [LISP] – 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, test: test, test-not: test-not)` [SAL]  
`(sublis alist expr &key :test :test-not)` [LISP] – 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

## C.21 Destructive List Functions

`rplaca(list, expr)` [SAL]  
`(rplaca list expr)` [LISP] – 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)` [SAL]

`(rplacd list expr)` [LISP] – 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 ...)` [SAL]

`(nconc list ...)` [LISP] – destructively concatenate lists  
*list* – lists to concatenate  
 returns – the result of concatenating the lists

`delete(expr, list, test: test, test-not: test-not)` [SAL]

`(delete expr list &key :test :test-not)` [LISP] – 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)` [SAL]

`(delete-if test list)` [LISP] – delete elements that pass test  
*test* – the test predicate  
*list* – the list  
 returns – the list with matching elements deleted

`delete-if-not(test, list)` [SAL]

`(delete-if-not test list)` [LISP] – delete elements that fail test  
*test* – the test predicate  
*list* – the list  
 returns – the list with non-matching elements deleted

`sort(list, test)` [SAL]

`(sort list test)` [LISP] – sort a list  
*list* – the list to sort  
*test* – the comparison function  
 returns – the sorted list

Note: The comparison function should have two parameters and return true if the first parameter should come before the second parameter in the sorted result. For a list of numbers, built-in comparison functions can be used, e.g. `(sort '(3 2 1) #'<)` returns `(1 2 3)`. To sort a list of lists by the first element of each list, you can write a function to access the keys and compare them, e.g. `(sort '((3 c) (2 b) (1 a)) #'(lambda (x y) (< (car x) (car y))))` returns `((1 A) (2 B) (3 C))`. In SAL, you could write function `my-sort(a, b) return first(a) < first(b) print sort({{3 c} {2 b} {1 a}}, quote(my-sort))`, which will print `{1 A} {2 B} {3 C}`.

## C.22 Predicate Functions

`atom(expr)` [SAL]

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

`symbolp(expr)` [SAL]  
(`symbolp expr`) [LISP] – is this a symbol?  
*expr* – the expression to check  
returns – `t` if the expression is a symbol, `nil` otherwise

`numberp(expr)` [SAL]  
(`numberp expr`) [LISP] – is this a number?  
*expr* – the expression to check  
returns – `t` if the expression is a number, `nil` otherwise

`null(expr)` [SAL]  
(`null expr`) [LISP] – is this an empty list?  
*expr* – the list to check  
returns – `t` if the list is empty, `nil` otherwise

`not(expr)` [SAL]  
(`not expr`) [LISP] – is this false?  
*expr* – the expression to check  
return – `t` if the value is `nil`, `nil` otherwise

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

`endp(list)` [SAL]  
(`endp list`) [LISP] – is this the end of a list  
*list* – the list  
returns – `t` if the value is `nil`, `nil` otherwise

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

`integerp(expr)` [SAL]  
(`integerp expr`) [LISP] – is this an integer?  
*expr* – the expression to check  
returns – `t` if the value is an integer, `nil` otherwise

`floatp(expr)` [SAL]  
(`floatp expr`) [LISP] – is this a float?  
*expr* – the expression to check  
returns – `t` if the value is a float, `nil` otherwise

`stringp(expr)` [SAL]  
(`stringp expr`) [LISP] – is this a string?  
*expr* – the expression to check  
returns – `t` if the value is a string, `nil` otherwise

`characterp(expr)` [SAL]  
(`characterp expr`) [LISP] – is this a character?  
*expr* – the expression to check  
returns – `t` if the value is a character, `nil` otherwise

`arrayp(expr)` [SAL]  
(`arrayp expr`) [LISP] – is this an array?  
*expr* – the expression to check  
returns – `t` if the value is an array, `nil` otherwise

`streamp(expr)` [SAL]  
(`streamp expr`) [LISP] – is this a stream?  
*expr* – the expression to check  
returns – `t` if the value is a stream, `nil` otherwise

`objectp(expr)` [SAL]  
(`objectp expr`) [LISP] – is this an object?  
*expr* – the expression to check  
returns – `t` if the value is an object, `nil` otherwise

`filep(expr)` [SAL]  
(`filep expr`) [LISP] – is this a file?  
*expr* – the expression to check  
returns – `t` if the value is an object, `nil` otherwise  
This is not part of standard XLISP nor is it built-in. Nyquist defines it though.

`boundp(sym)` [SAL]  
(`boundp sym`) [LISP] – is a value bound to this symbol?  
*sym* – the symbol  
returns – `t` if a value is bound to the symbol, `nil` otherwise

`fboundp(sym)` [SAL]  
(`fboundp sym`) [LISP] – 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)` [SAL]  
(`minusp expr`) [LISP] – is this number negative?  
*expr* – the number to test  
returns – `t` if the number is negative, `nil` otherwise

`zerop(expr)` [SAL]  
(`zerop expr`) [LISP] – is this number zero?  
*expr* – the number to test  
returns – `t` if the number is zero, `nil` otherwise

`plusp(expr)` [SAL]  
(`plusp expr`) [LISP] – is this number positive?  
*expr* – the number to test  
returns – `t` if the number is positive, `nil` otherwise

`evenp(expr)` [SAL]  
(`evenp expr`) [LISP] – is this integer even?  
*expr* – the integer to test  
returns – `t` if the integer is even, `nil` otherwise

`oddp(expr)` [SAL]  
(`oddp expr`) [LISP] – is this integer odd?  
*expr* – the integer to test  
returns – `t` if the integer is odd, `nil` otherwise

`eq(expr1, expr2)` [SAL]  
 (`eq expr1 expr2`) [LISP] – are the expressions identical (pointer equality)? Numbers and strings are generally not `eq`, e.g. (`eq 256 256`), (`eq 1.0 1.0`), and (`eq "a" "a"`) are false.  
*expr1* – the first expression  
*expr2* – the second expression  
 returns – `t` if they are equal, `nil` otherwise

`eq1(expr1, expr2)` [SAL]  
 (`eq1 expr1 expr2`) [LISP] – are the expressions of equal value? (`eq1` tests for identical objects (pointer equality) except for numbers. Two numbers can be `eq1` even if they are stored in different locations. However, a `FIXNUM` is never `eq1` to a `FLONUM`, i.e. (`eq1 1 1.0`) is false.)  
*expr1* – the first expression  
*expr2* – the second expression  
 returns – `t` if they are equal, `nil` otherwise

`equal(expr1, expr2)` [SAL]  
 (`equal expr1 expr2`) [LISP] – are the expressions equal? Arrays are not equal unless they are the same array (pointer equality), but numbers and strings are compared by value, and lists are tested recursively for equal content. A `FIXNUM` is never equal to a `FLONUM`.  
*expr1* – the first expression  
*expr2* – the second expression  
 returns – `t` if they are equal, `nil` otherwise

## C.23 Control Constructs

(`cond pair ...`) [LISP] – 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 ...)` [SAL]  
 (`and expr ...`) [LISP] – 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 ...)` [SAL]  
 (`or expr ...`) [LISP] – 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]`) [LISP] – 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` (default is `nil`)

returns – the value of the selected expression.

Note that the SAL conditional expression syntax is `?(test, iftrue-expression, iffalse-expression)`, but `#if` may be used instead of `?`. Either form may omit the third argument, which defaults to `nil`.

`when(texpr, expr ...)` [SAL]

(`when texpr expr ...`) [LISP] – 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 ...)` [SAL]

(`unless texpr expr ...`) [LISP] – 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 ...`) [LISP] – 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 ...`) [LISP] – create local bindings

(`let* (binding ...) expr ...`) [LISP] – 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 ...`) [LISP] – create local functions

(`labels (binding ...) expr ...`) [LISP] – `flet` with recursive functions

(`macrolet (binding ...) expr ...`) [LISP] – 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 ...)` [SAL]

(`catch sym expr ...`) [LISP] – 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])` [SAL]

(`throw sym [expr]`) [LISP] – 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 ...)` [SAL]

`(unwind-protect expr cexpr ...)` [LISP] – 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

## C.24 Looping Constructs

`(loop expr ...)` [LISP] – basic looping form

*expr* – the body of the loop

returns – never returns (must use non-local exit)

`(do (binding ...) (texpr reexpr ...) expr ...)` [LISP]

`(do* (binding ...) (texpr reexpr ...) expr ...)` [LISP]

*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

*reexpr* – 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 [reexpr]) expr ...)` [LISP] – loop through a list

*sym* – the symbol to bind to each list element

*expr* – the list expression

*reexpr* – the result expression (the default is `nil`)

*expr* – the body of the loop (treated like an implicit `prog`)

`(dotimes (sym expr [reexpr]) expr ...)` [LISP] – 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

*reexpr* – the result expression (the default is `nil`)

*expr* – the body of the loop (treated like an implicit `prog`)

## C.25 The Program Feature

(prog (*binding* ...) *expr* ...) [LISP] – the program feature  
 (prog\* (*binding* ...) *expr* ...) [LISP] – 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* ...) [SAL]  
 (block *name* *expr* ...) [LISP] – named block  
*name* – the block name (symbol)  
*expr* – the block body  
 returns – the value of the last expression

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

return-from(*name* [, *value*]) [SAL]  
 (return-from *name* [*value*]) [LISP] – return from a named block  
*name* – the block name (symbol)  
*value* – the value to return (defaults to nil)  
 returns – never returns

tagbody(*expr* ...) [SAL]  
 (tagbody *expr* ...) [LISP] – block with labels  
*expr* – expression(s) to evaluate or tags (symbols)  
 returns – nil

go(*sym*) [SAL]  
 (go *sym*) [LISP] – go to a tag within a tagbody or prog  
*sym* – the tag (quoted)  
 returns – never returns

(progv *slist* *vlist* *expr* ...) [LISP] – 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* ...) [SAL]  
 (prog1 *expr1* *expr* ...) [LISP] – 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* ...) [SAL]  
 (prog2 *expr1* *expr2* *expr* ...) [LISP] – 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 ...)` [SAL]  
 (`progn expr ...`) [LISP] – execute expressions sequentially  
*expr* – the expressions to evaluate  
 returns – the value of the last expression (or `nil`)

## C.26 Debugging and Error Handling

`trace(sym)` [SAL]  
 (`trace sym`) [LISP] – add a function to the trace list  
*sym* – the function to add (quoted)  
 returns – the trace list

`untrace(sym)` [SAL]  
 (`untrace sym`) [LISP] – remove a function from the trace list  
*sym* – the function to remove (quoted)  
 returns – the trace list

`error(msg [, arg])` [SAL]  
 (`error msg [arg]`) [LISP] – signal a non-correctable error  
*msg* – the error message string  
*arg* – the argument expression (printed after the message)  
 returns – never returns

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

`break([[bmsg [, arg]])` [SAL]  
 (`break [bmsg [arg]]`) [LISP] – 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)` [LISP] – clean-up after an error  
 returns – never returns

`(top-level)` [LISP] – clean-up after an error and return to the top level  
 returns – never returns

`(continue)` [LISP] – continue from a correctable error  
 returns – never returns

`(errset expr [pflag])` [LISP] – 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

(*backtrace* [*n*]) [LISP] – print *n* levels of trace back information  
*n* – the number of levels (defaults to all levels)  
returns – *nil*

(*evalhook* *expr ehook ahook* [*env*]) [LISP] – 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*) [SAL]  
(*profile* *flag*) [LISP] – turn profiling on or off.  
*flag* – *nil* turns profiling off, otherwise on  
returns – the previous state of profiling.  
This is not part of standard XLISP.

## C.27 Arithmetic Functions

*truncate*(*expr*) [SAL]  
(*truncate* *expr*) [LISP] – truncates a floating point number to an integer  
*expr* – the number  
returns – the result of truncating the number

*float*(*expr*) [SAL]  
(*float* *expr*) [LISP] – converts an integer to a floating point number  
*expr* – the number  
returns – the result of floating the integer

(+ *expr* ...) [LISP] – add a list of numbers  
*expr* – the numbers  
returns – the result of the addition

(- *expr* ...) [LISP] – subtract a list of numbers or negate a single number  
*expr* – the numbers  
returns – the result of the subtraction

(\* *expr* ...) [LISP] – multiply a list of numbers  
*expr* – the numbers  
returns – the result of the multiplication

(/ *expr* ...) [LISP] – divide a list of numbers  
*expr* – the numbers  
returns – the result of the division

(1+ *expr*) [LISP] – add one to a number  
*expr* – the number  
returns – the number plus one

- (1- *expr*) [LISP] – subtract one from a number  
*expr* – the number  
returns – the number minus one
- rem(*expr* ...) [SAL]  
(rem *expr* ...) [LISP] – remainder of a list of numbers  
*expr* – the numbers  
returns – the result of the remainder operation
- min(*expr* ...) [SAL]  
(min *expr* ...) [LISP] – the smallest of a list of numbers  
*expr* – the expressions to be checked  
returns – the smallest number in the list
- max(*expr* ...) [SAL]  
(max *expr* ...) [LISP] – the largest of a list of numbers  
*expr* – the expressions to be checked  
returns – the largest number in the list
- abs(*expr*) [SAL]  
(abs *expr*) [LISP] – the absolute value of a number  
*expr* – the number  
returns – the absolute value of the number
- gcd(*n1*, *n2* ...) [SAL]  
(gcd *n1* *n2* ...) [LISP] – compute the greatest common divisor  
*n1* – the first number (integer)  
*n2* – the second number(s) (integer)  
returns – the greatest common divisor
- random(*n*) [SAL]  
(random *n*) [LISP] – compute a random number between 0 and *n*-1 inclusive. If *n* is 0, return 0.  
*n* – the upper bound (integer)  
returns – a random number
- rrandom() [SAL]  
(rrandom) [LISP] – compute a random real number between 0 and 1 inclusive  
returns – a random floating point number
- random-seed(*n*) [SAL]  
(random-seed *n*) [LISP] – seed the random number generator with starting seed *n*. If random-seed is not called, sranddev or some other initialization method will be used by default.  
*n* – the upper bound (integer)  
returns – a random number
- sin(*expr*) [SAL]  
(sin *expr*) [LISP] – compute the sine of a number  
*expr* – the floating point number  
returns – the sine of the number
- cos(*expr*) [SAL]  
(cos *expr*) [LISP] – compute the cosine of a number  
*expr* – the floating point number  
returns – the cosine of the number

`tan(expr)` [SAL]

(`tan expr`) [LISP] – compute the tangent of a number  
*expr* – the floating point number  
 returns – the tangent of the number

`atan(expr [, expr2])` [SAL]

(`atan expr [expr2]`) [LISP] – compute the arctangent  
*expr* – the value of *x*  
*expr2* – the value of *y* (default value is 1.0)  
 returns – the arctangent of *x/y*  
 This is not part of standard XLISP.

`expt(x-expr, y-expr)` [SAL]

(`expt x-expr y-expr`) [LISP] – 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)` [SAL]

(`exp x-expr`) [LISP] – compute *e* to the *x* power  
*x-expr* – the floating point number  
 returns – *e* to the *x* power

`sqrt(expr)` [SAL]

(`sqrt expr`) [LISP] – compute the square root of a number  
*expr* – the floating point number  
 returns – the square root of the number

(`< n1 n2 ...`) [LISP] – test for less than

(`<= n1 n2 ...`) [LISP] – test for less than or equal to

(`= n1 n2 ...`) [LISP] – test for equal to

(`/= n1 n2 ...`) [LISP] – test for not equal to

(`>= n1 n2 ...`) [LISP] – test for greater than or equal to

(`> n1 n2 ...`) [LISP] – test for greater than

*n1* – the first number to compare

*n2* – the second number to compare

returns – `t` if all arguments are numbers and the results of comparing *n1* with *n2*, *n2* with *n3*, etc., are all true.

(FIXNUMS are converted to FLONUMS in mixed-type comparisons.)

## C.28 Bitwise Logical Functions

`logand(expr ...)` [SAL]

(`logand expr ...`) [LISP] – the bitwise and of a list of numbers  
*expr* – the numbers  
 returns – the result of the and operation

`logior(expr ...)` [SAL]

(`logior expr ...`) [LISP] – the bitwise inclusive or of a list of numbers  
*expr* – the numbers  
 returns – the result of the inclusive or operation

`logxor(expr ...)` [SAL]  
 (`logxor expr ...`) [LISP] – the bitwise exclusive or of a list of numbers  
*expr* – the numbers  
 returns – the result of the exclusive or operation

`lognot(expr)` [SAL]  
 (`lognot expr`) [LISP] – the bitwise not of a number  
*expr* – the number  
 returns – the bitwise inversion of number

## C.29 String Functions

`string(expr)` [SAL]  
 (`string expr`) [LISP] – 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, start: start, end: end)` [SAL]  
 (`string-search pat str &key :start :end`) [LISP] – 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  
 This is not part of standard XLISP.

`string-trim(bag, str)` [SAL]  
 (`string-trim bag str`) [LISP] – trim both ends of a string  
*bag* – a string containing characters to trim  
*str* – the string to trim  
 returns – a trimmed copy of the string

`string-left-trim(bag, str)` [SAL]  
 (`string-left-trim bag str`) [LISP] – trim the left end of a string  
*bag* – a string containing characters to trim  
*str* – the string to trim  
 returns – a trimmed copy of the string

`string-right-trim(bag, str)` [SAL]  
 (`string-right-trim bag str`) [LISP] – trim the right end of a string  
*bag* – a string containing characters to trim  
*str* – the string to trim  
 returns – a trimmed copy of the string

`string-upcase(str, start: start, end: end)` [SAL]  
 (`string-upcase str &key :start :end`) [LISP] – 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*, start: *start*, end: *end*) [SAL]  
 (string-downcase *str* &key :start :end) [LISP] – 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*, start: *start*, end: *end*) [SAL]  
 (nstring-upcase *str* &key :start :end) [LISP] – 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*, start: *start*, end: *end*) [SAL]  
 (nstring-downcase *str* &key :start :end) [LISP] – convert to lowercase  
*str* – the string  
 :start – the starting offset  
 :end – the ending offset + 1  
 returns – the converted string (not a copy)

strcat(*expr* ...) [SAL]  
 (strcat *expr* ...) [LISP] – concatenate strings  
*expr* – the strings to concatenate  
 returns – the result of concatenating the strings

subseq(*string*, start [, end]) [SAL]  
 (subseq *string* start [end]) [LISP] – 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*, start1: *start1*, end1: *end1*, start2: *start2*, end2: *end2*) [SAL]  
 (string< *str1* *str2* &key :start1 :end1 :start2 :end2) [LISP]  
 string<=(*str1*, *str2*, start1: *start1*, end1: *end1*, start2: *start2*, end2: *end2*) [SAL]  
 (string<= *str1* *str2* &key :start1 :end1 :start2 :end2) [LISP]  
 string=(*str1*, *str2*, start1: *start1*, end1: *end1*, start2: *start2*, end2: *end2*) [SAL]  
 (string= *str1* *str2* &key :start1 :end1 :start2 :end2) [LISP]  
 string/=(*str1*, *str2*, start1: *start1*, end1: *end1*, start2: *start2*, end2: *end2*) [SAL]  
 (string/= *str1* *str2* &key :start1 :end1 :start2 :end2) [LISP]  
 string>=(*str1*, *str2*, start1: *start1*, end1: *end1*, start2: *start2*, end2: *end2*) [SAL]  
 (string>= *str1* *str2* &key :start1 :end1 :start2 :end2) [LISP]  
 string>(*str1*, *str2*, start1: *start1*, end1: *end1*, start2: *start2*, end2: *end2*) [SAL]  
 (string> *str1* *str2* &key :start1 :end1 :start2 :end2) [LISP]  
*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, start1: start1, end1: end1,
            start2: start2, end2: end2) [SAL]
(string-lessp str1 str2 &key :start1 :end1 :start2 :end2) [LISP]
string-not-greaterp(str1, str2, start1: start1, end1: end1,
                  start2: start2, end2: end2) [SAL]
(string-not-greaterp str1 str2 &key :start1 :end1 :start2 :end2) [LISP]
string-equal(str1, str2, start1: start1, end1: end1,
            start2: start2, end2: end2) [SAL]
(string-equal str1 str2 &key :start1 :end1 :start2 :end2) [LISP]
string-not-equal(str1, str2, start1: start1, end1: end1,
                start2: start2, end2: end2) [SAL]
(string-not-equal str1 str2 &key :start1 :end1 :start2 :end2) [LISP]
string-not-lessp(str1, str2, start1: start1, end1: end1,
                start2: start2, end2: end2) [SAL]
(string-not-lessp str1 str2 &key :start1 :end1 :start2 :end2) [LISP]
string-greaterp(str1, str2, start1: start1, end1: end1,
                start2: start2, end2: end2) [SAL]
(string-greaterp str1 str2 &key :start1 :end1 :start2 :end2) [LISP]
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.

```

## C.30 Character Functions

```

char(string, index) [SAL]
(char string index) [LISP] – 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) [SAL]
(upper-case-p chr) [LISP] – is this an upper case character?
chr – the character
returns – t if the character is upper case, nil otherwise

lower-case-p(chr) [SAL]
(lower-case-p chr) [LISP] – is this a lower case character?
chr – the character
returns – t if the character is lower case, nil otherwise

both-case-p(chr) [SAL]
(both-case-p chr) [LISP] – is this an alphabetic (either case) character?
chr – the character

```

returns – `t` if the character is alphabetic, `nil` otherwise

`digit-char-p( chr )` [SAL]

(`digit-char-p chr`) [LISP] – is this a digit character?

*chr* – the character

returns – the digit weight if character is a digit, `nil` otherwise

`char-code( chr )` [SAL]

(`char-code chr`) [LISP] – get the ascii code of a character

*chr* – the character

returns – the ascii character code (integer)

`code-char( code )` [SAL]

(`code-char code`) [LISP] – get the character with a specified ascii code

*code* – the ascii code (integer)

returns – the character with that code or `nil`

`char-upcase( chr )` [SAL]

(`char-upcase chr`) [LISP] – convert a character to upper case

*chr* – the character

returns – the upper case character

`char-downcase( chr )` [SAL]

(`char-downcase chr`) [LISP] – convert a character to lower case

*chr* – the character

returns – the lower case character

`digit-char( n )` [SAL]

(`digit-char n`) [LISP] – convert a digit weight to a digit

*n* – the digit weight (integer)

returns – the digit character or `nil`

`char-int( chr )` [SAL]

(`char-int chr`) [LISP] – convert a character to an integer

*chr* – the character

returns – the ascii character code

`int-char( int )` [SAL]

(`int-char int`) [LISP] – convert an integer to a character

*int* – the ascii character code

returns – the character with that code

`char<( chr1, chr2 ... )` [SAL]

(`char< chr1 chr2 ...`) [LISP]

`char<=( chr1, chr2 ... )` [SAL]

(`char<= chr1 chr2 ...`) [LISP] `char=( chr1, chr2 ... )` [SAL]

(`char= chr1 chr2 ...`) [LISP]

`char/=( chr1, chr2 ... )` [SAL]

(`char/= chr1 chr2 ...`) [LISP]

`char>=( chr1, chr2 ... )` [SAL]

(`char>= chr1 chr2 ...`) [LISP]

`char>( chr1, chr2 ... )` [SAL]

(`char> chr1 chr2 ...`) [LISP]

*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 ...)` [SAL]  
`(char-lessp chr1 chr2 ...)` [LISP]  
`char-not-greaterp(chr1, chr2 ...)` [SAL]  
`(char-not-greaterp chr1 chr2 ...)` [LISP]  
`char-equal(chr1, chr2 ...)` [SAL]  
`(char-equal chr1 chr2 ...)` [LISP]  
`char-not-equal(chr1, chr2 ...)` [SAL]  
`(char-not-equal chr1 chr2 ...)` [LISP]  
`char-not-lessp(chr1, chr2 ...)` [SAL]  
`(char-not-lessp chr1 chr2 ...)` [LISP]  
`char-greaterp(chr1, chr2 ...)` [SAL]  
`(char-greaterp chr1 chr2 ...)` [LISP]  
*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.

## C.31 Input/Output Functions

`read([stream [, eof [, rflag]])` [SAL]  
`(read [stream [eof [rflag]])` [LISP] – 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])` [LISP] – print an expression on a new line  
*expr* – the expression to be printed  
*stream* – the output stream (default is standard output)  
 returns – the expression

`(display label expr ...)` [LISP] – print expressions and their values  
*label* – a string prefix to print, often the current function name or something to identify the source of this line of output  
*expr* – a list of expressions. Each expression is printed literally without evaluation, then it is evaluated and the value is printed. This helps to generate readable debugging output such as "In foo: A = 4, B = 2.5"  
 returns – the expression  
 Note: Output from `display` can be turned on and off by calling `display-on` or `display-off` as described below. This is not part of standard XLISP nor is it built-in. Nyquist defines it though.

`(display-on)` [LISP] – enable `display` macro  
 returns – *T*  
 Note: This call gives `display` its default definition as described above. This is not part of standard XLISP nor is it built-in. Nyquist defines it though.

(display-off) [LISP] – disable display macro  
 returns – NIL  
 Note: This call redefines display to just evaluate arguments and not print anything.  
 This is not part of standard XLISP nor is it built-in. Nyquist defines it though.

prin1(*expr* [, *stream*]) [SAL]  
 (prin1 *expr* [*stream*]) [LISP] – print an expression  
*expr* – the expression to be printed  
*stream* – the output stream (default is standard output)  
 returns – the expression

princ(*expr* [, *stream*]) [SAL]  
 (princ *expr* [*stream*]) [LISP] – 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*]) [SAL]  
 (pprint *expr* [*stream*]) [LISP] – pretty print an expression  
*expr* – the expressions to be printed  
*stream* – the output stream (default is standard output)  
 returns – the expression

terpri([*stream*]) [SAL]  
 (terpri [*stream*]) [LISP] – terminate the current print line  
*stream* – the output stream (default is standard output)  
 returns – nil

flatsize(*expr*) [SAL]  
 (flatsize *expr*) [LISP] – length of printed representation using prin1  
*expr* – the expression  
 returns – the length

flatc(*expr*) [SAL]  
 (flatc *expr*) [LISP] – length of printed representation using princ  
*expr* – the expression  
 returns – the length

## C.32 The Format Function

format(*stream*, *fmt*, *arg* ...) [SAL]  
 (format *stream* *fmt* *arg* ...) [LISP] – do formatted  
 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

## C.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, direction: direction)` [SAL]

`(open fname &key :direction)` [LISP] – open a file stream  
*fname* – the file name string or symbol  
 :direction – :input or :output (default is :input)  
 returns – a stream

`open-binary(fname, direction: direction)` [SAL]

`(open-binary fname &key :direction)` [LISP] – open a binary file stream  
*fname* – the file name string or symbol  
 :direction – :input or :output (default is :input)  
 returns – a stream

`close(stream)` [SAL]

`(close stream)` [LISP] – close a file stream  
*stream* – the stream  
 returns – `nil`

`setdir(path [, verbose])` [SAL]

`(setdir path [verbose])` [LISP] – set current directory  
*path* – the path of the new directory  
*verbose* – print error message if current directory cannot be changed to *path*  
 returns – the resulting full path, e.g. `(setdir ".")` gets the current working directory, or `nil` if an error occurs  
 This is not part of standard XLISP.

`listdir(path)` [SAL]

`(listdir path)` [LISP] – get a directory listing  
*path* – the path of the directory to be listed  
 returns – list of filenames in the directory  
 This is not part of standard XLISP.

`get-temp-path()` [SAL]

`(get-temp-path)` [LISP] – 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  
 This is not part of standard XLISP.

`get-user()` [SAL]

`(get-user)` [LISP] – 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

This is not part of standard XLISP.

`find-in-xlisp-path(filename)` [SAL]

`(find-in-xlisp-path filename)` [LISP] – 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

This is not part of standard XLISP.

`read-char([stream])` [SAL]

`(read-char [stream])` [LISP] – read a character from a stream

*stream* – the input stream (default is standard input)

returns – the character

`peek-char([flag [, stream]])` [SAL]

`(peek-char [flag [stream]])` [LISP] – 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

`write-char(ch [, stream])` [SAL]

`(write-char ch [stream])` [LISP] – 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]])` [SAL]

`(read-int [stream [length]])` [LISP] – 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]])` [SAL]

`(write-int ch [stream [length]])` [LISP] – write a binary integer to a stream

*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]])` [SAL]  
 (`read-float stream [length]`) [LISP] – 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]])` [SAL]  
 (`write-float ch stream [length]`) [LISP] – 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])` [SAL]  
 (`read-line stream`) [LISP] – read a line from a stream  
*stream* – the input stream (default is standard input)  
 returns – the string

`read-byte([stream])` [SAL]  
 (`read-byte stream`) [LISP] – read a byte from a stream  
*stream* – the input stream (default is standard input)  
 returns – the byte (integer)

`write-byte(byte [, stream])` [SAL]  
 (`write-byte byte stream`) [LISP] – write a byte to a stream  
*byte* – the byte to write (integer)  
*stream* – the output stream (default is standard output)  
 returns – the byte (integer)

## C.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]])` [SAL]  
 (`make-string-input-stream str [start [end]]`) [LISP]  
*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(stream)` [SAL]  
`(make-string-output-stream)` [LISP]  
 returns – an unnamed output stream

`get-output-stream-string(stream)` [SAL]  
`(get-output-stream-string stream)` [LISP]  
*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)` [SAL]  
`(get-output-stream-list stream)` [LISP]  
*stream* – the output stream  
 returns – the output so far as a list  
 Note: the output stream is emptied by this function

## C.35 System Functions

Note: the `load` function first tries to load a file from the current directory. A `.lisp` 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.)

`get-real-time()` [SAL]  
`(get-real-time)` [LISP] – get the time  
 returns – Time as `FLONUM` seconds since Jan 1, 1970  
 Note: Since the time is a large number, you may want to subtract an earlier time to measure a time interval or use a custom format to print enough precision. (See `*float-format*` in Section C.15.)

`get-run-time()` [SAL]  
`(get-run-time)` [LISP] – get the run time based on number of Lisp expression evaluations. Typically, a computer will use one unit of run time in about 10ms, but this can vary either way and depends on CPU speed.  
 returns – Time as a `FIXNUM`.

`get-env(name)` [SAL]  
`(get-env name)` [LISP] – get from an environment variable  
*name* – the name of the environment variable  
 returns – string value of the environment variable, `nil` if variable does not exist

`(load fname &key :verbose :print)` [LISP] – 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)` [SAL]  
`(save fname)` [LISP] – save workspace to a file  
*fname* – the filename string or symbol  
 returns – `t` if workspace was written, `nil` otherwise

`restore(fname)` [SAL]  
(`restore fname`) [LISP] – restore workspace from a file  
*fname* – the filename string or symbol  
returns – `nil` on failure, otherwise never returns

`dribble([fname])` [SAL]  
(`dribble [fname]`) [LISP] – 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()` [SAL]  
(`gc`) [LISP] – force garbage collection  
returns – `nil`

`expand(num)` [SAL]  
(`expand num`) [LISP] – expand memory by adding segments  
*num* – the number of segments to add  
returns – the number of segments added

`alloc(num)` [SAL]  
(`alloc num`) [LISP] – 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()` [SAL]  
(`info`) [LISP] – show information about memory usage.  
returns – `nil`

`room()` [SAL]  
(`room`) [LISP] – show memory allocation statistics  
returns – `nil`

`type-of(expr)` [SAL]  
(`type-of expr`) [LISP] – 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)` [SAL]  
(`peek addrs`) [LISP] – peek at a location in memory  
*addrs* – the address to peek at (integer)  
returns – the value at the specified address (integer)

`poke(addr, value)` [SAL]

`(poke addr value)` [LISP] – poke a value into memory  
*addr* – the address to poke (integer)  
*value* – the value to poke into the address (integer)  
 returns – the value

`bigendianp()` [SAL]

`(bigendianp)` [LISP] – 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.  
 This is not part of standard XLISP.

`address-of(expr)` [SAL]

`(address-of expr)` [LISP] – get the address of an xlist node  
*expr* – the node  
 returns – the address of the node (integer)

`exit()` [SAL]

`(exit)` [LISP] – exit xlist. (Note: in Audacity plug-ins, `exit` is undefined because exiting would terminate Audacity.)  
 returns – never returns

`setup-console()` [SAL]

`(setup-console)` [LISP] – 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)` [SAL]

`(echoenabled flag)` [LISP] – 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.

## C.36 File I/O Functions

### C.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 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.

### C.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.

### C.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))
```