Fugue Reference Manual

Version 1.0

Roger B. Dannenberg
19 August 1991

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

Table of Contents

1. Introduction and Overview 3
- **1.1. Installation** 3
- **1.2. Examples** 4
  - **1.2.1. Waveforms** 4
  - **1.2.2. Sequences** 5
  - **1.2.3. Envelopes** 5
- **1.3. Pre-defined Constants** 6

2. Behavioral Abstraction 9
- **2.1. The Environment** 9
- **2.2. Sequential Behavior** 10
- **2.3. Simultaneous Behavior** 10
- **2.4. Sounds vs. Behaviors** 11
- **2.5. The At Transformation** 11
- **2.6. Nested Transformations** 12
- **2.7. Defining Behaviors** 12

3. More Examples 13
- **3.1. Stretching Sampled Sounds** 13
- **3.2. Saving Sound Files** 14
- **3.3. Frequency Modulation** 14

4. Fugue Functions 17
- **4.1. Sounds** 17
  - **4.1.1. What is a Sound?** 17
  - **4.1.2. Creating Sounds** 18
  - **4.1.3. Accessing Sounds** 18
  - **4.1.4. Low-Level Manipulation Primitives** 19
  - **4.1.5. Other Low-Level Primitives** 20
  - **4.1.6. Miscellaneous Operations** 22
- **4.2. Behaviors** 22
  - **4.2.1. Using Previously Created Sounds** 22
  - **4.2.2. Sound Synthesis** 23
- **4.3. Transformations** 24
- **4.4. Combination and Time Structure** 25

Appendix I. ICMC Conference Paper on Fugue 27

Appendix II. Lazy Evaluation 33
- **II.1. Data Types** 35
- **II.2. The Sample Structure** 35
- **II.3. The Node Structure** 35
- **II.4. The Sound Structure** 36

Appendix III. C Functions And Example 39
- **III.1. Constructors** 39
- **III.2. Destructors** 39
- **III.3. Manipulators** 40
- **III.4. fugue.c** 40
- **III.5. Examples** 41

Appendix IV. Intgen 43
  - **IV.0.1. Extending Xlisp** 43
- **IV.1. Header file format** 44
- **IV.2. Using #define'd macros** 45
- **IV.3. Lisp Include Files** 46
- **IV.4. Example** 46

IV.5. More Details 46
Appendix V. XLISP: An Object-oriented Lisp 47
V.1. Introduction 48
V.2. A Note From The Author 48
V.3. XLISP Command Loop 49
V.4. Break Command Loop 49
V.5. Data Types 49
V.6. The Evaluator 50
V.7. Lexical Conventions 50
V.8. Readtables 51
V.9. Lambda Lists 52
V.10. Objects 53
V.11. The "Object" Class 54
V.12. The "Class" Class 54
V.13. SYMBOLS 55
V.14. Evaluation Functions 56
V.15. Symbol Functions 57
V.16. Property List Functions 58
V.17. Array Functions 59
V.18. List Functions 59
V.19. Destructive List Functions 62
V.20. Predicate Functions 63
V.21. Control Constructs 65
V.22. Looping Constructs 67
V.23. The Program Feature 68
V.24. Debugging and Error Handling 69
V.25. Arithmetic Functions 70
V.26. Bitwise Logical Functions 72
V.27. String Functions 72
V.28. Character Functions 74
V.29. Input/Output Functions 76
V.30. The Format Function 77
V.31. File I/O Functions 77
V.32. String Stream Functions 78
V.33. System Functions 79
V.34. File I/O Functions 80
V.34.1. Input from a File 80
V.34.2. Output to a File 81
V.34.3. A Slightly More Complicated File Example 81
Index 83

List of Figures

Figure 1: An envelope generated by the env function. 5

List of Figures

Figure 1: An envelope generated by the env function. 5

Preface

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

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

Several people have contributed to Fugue. Chris Fraley wrote the original implementation as a student of Roger Dannenberg. George Polly augmented the original version with some new functions. Peter Velikonja and Dean Rubine were early users, and their bravery in dealing with a fragile system let to the discovery of many bugs. The latest version of Fugue has undergone a considerable amount of rewriting, debugging, and enhancement by this author.

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

1. Introduction and Overview

Fugue 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, Fugue handles both in a single integrated system. Fugue is also flexible and easy to use because it is based on an interactive Lisp interpreter.

With Fugue, 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.

Fugue runs under any Unix environment and it produces sound files as output. If you can play a sound file by typing a command to a Unix shell, then you can get Fugue to play sounds for you. Fugue is currently configured to run on a NeXT machine, using the built-in sound system to play Fugue output.

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

There are several articles about the design of Fugue and the problems that it solves. The shortest of these, which appeared in the Proceedings of the ICMC 1989, is included as Appendix I. The July 1991 issue of IEEE Computer gives more complete coverage. In this manual, I will give some examples to show how Fugue can be used and describe in detail the available functions. Appendix III describes the implementation and how to extend it.

1.1. Installation

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

```
zcat fugue.tar.Z | tar xf -
cd fugue
make
```

The first line creates a `fugue` directory and some subdirectories. Assuming the make completes successfully, you can run fugue as follows:

```
cd test
../fugue
```

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

Note: Fugue looks for the file `init.lsp` in the current directory. If you look in the `init.lsp` in `test`, you will notice two things. First, `init.lsp` loads `fugue.lsp` from the fugue directory, and second, `init.lsp` defines the function `play`. It assumes you have a unix command `play` that will play a 16-bit mono, 22050 Hz sample rate file with no headers. A play program for the NeXT machine is included in the distribution, but you will have to make it and set up your paths so that it will be found.

1.2. Examples

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

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

1.2.1. Waveforms

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

The next step is to add several harmonics together. The function `mkwave` calls upon `build-harmonic` to generate a total of four harmonics with amplitudes 1.0, 0.5, 0.25, and 0.12. These are scaled (by `s-scale`) and added (by `s-add`) to create a waveform which is bound to `table`. ***Note:*** *The functions* `s-add` *and* `s-scale` *should be used with care, and normally apply only to special non-time-domain signals like wave tables. Other examples will illustrate how to perform similar operations on the more usual time-domain signals.*

A complete Fugue waveform is a list consisting of a sound, a pitch, and `t`, indicating a periodic waveform. The pitch gives the nominal pitch of the sound. (This is implicit in a single cycle wave table, but a sampled sound may have many cycles.) This dotted pair is formed in the last line of `mkwave`: we compute the pitch by dividing the sample rate by the table length to get Hertz, and then convert this to a pitch number.

```
(defun mkwave ()
  (let ((table
         (s-add (s-scale (build-harmonic 1.0) 1.0)
          (s-add (s-scale (build-harmonic 2.0) 0.5)
           (s-add (s-scale (build-harmonic 3.0) 0.25)
            (s-scale (build-harmonic 4.0) 0.12))))))
    (setf *wave*
     (list table
           (hz-to-step (/ *SOUND-SRATE* 2048.0))
           t)) ))
```

The last step of this example is to build the wave. The following code calls `mkwave` only if `*wave*` is undefined. Since `*wave*` is set by `mkwave`, `mkwave` will not be called again

when the file is reloaded:

```
(if (not (boundp '*wave*)) (mkwave))
```

1.2.2. Sequences

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

```
(defun note (pitch dur) (osc pitch dur 0 *wave*))

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

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

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

1.2.3. Envelopes

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

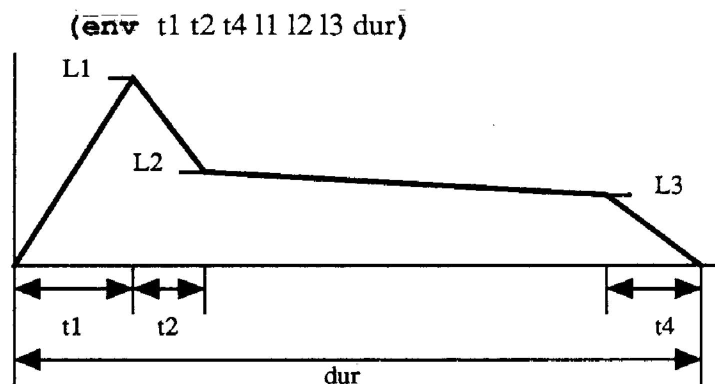

Figure 1: An envelope generated by the env function.

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

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


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

The end of this example shows the use of `stretch` to modify durations. There are several transformations supported by Fugue, and transformations of abstract behaviors is perhaps ***the*** fundamental idea behind Fugue. The next section is devoted to explaining this concept, and further elaboration can be found in Appendix I.

1.3. Pre-defined Constants

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

- **Dynamics**

  `lppp` = 0.33
  `lpp` = 0.5
  `lp` = 0.75
  `lmp` = 0.90
  `lmf` = 0.11
  `lf` = 1.33
  `lff` = 2.00
  `lfff` = 3.00
  `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**

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

2. Behavioral Abstraction

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

2.1. The Environment

The transformation environment consists of a set of special Lisp variables. These variables should be considered read-only and should *never* be set directly by the programmer. Instead, they are automatically maintained 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 `*volume*` may change timbre more than amplitude, and `*transpose*` may be ignored by percussion instruments:

`*time*` Logical starting time. *Note:* this is the abstract or perceptual starting time of a behavior. The actual or physical starting time can be earlier or later. As examples, a pickup-note to a phrase or the rise-time of a note may occur *before* `*time*`.

`*volume*` Loudness, expressed as a linear factor. The default (nominal) loudness is 1.0.

`*transpose*` Pitch transposition, expressed in semitones.

`*stretch*` Amount by which to stretch in time.

`*duty*` The ''duty factor'', or amount by which to separate or overlap sequential notes. For example, staccato might be expressed with a `*duty*` of 0.5, while very legato playing might be expressed with a `*duty*` of 1.2.

Specifically, `*duty*` stretches the duration of notes (articulation) without affecting the inter-onset time (the rhythm).

`*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.

`*stop*` Stop time of clipping region. By analogy to `*start*`, no sound should be generated after this time.

`*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.

`*sound-srate*` Sample rate of musical sounds. This environment element provides the default sample rate for musical sounds.

2.2. Sequential Behavior

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

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

This expression is evaluated as follows: first, `*time*` is set to `0`, and `(note c4 q)` is evaluated. A sound is returned and saved. The sound has an ending time, which in this case will be `1.0` because the duration `q` is `1.0`. This ending time, `1.0`, is assigned to `*time*`, and the second note is evaluated. The second note will start at `*time*` which is now `1.0`. The sound that is returned is now added to the first sound to form a composite sound, whose duration will be `2.0`. `*time*` is restored to `0.0`.

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 at the current `*time*`. Evaluate each successive behavior with `*time*` set to the ending time of the previous behavior. Restore `*time*` to its original value and return a sound which is the sum of the results.

2.3. Simultaneous Behavior

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

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

will play both notes starting at the same time.

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

2.4. Sounds vs. Behaviors

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

```
; load a sound
;
(setf a-snd (sf-load
              "/usr0/rbd/fugue/test/ex/demo-snd.nh"
              22050.0))

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

One might think that the following would then work:

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

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

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

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

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

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

2.5. The At Transformation

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

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

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

2.6. Nested Transformations

Transformations can be combined using nested expressions. For example,

```
(sim (cue a-snd)
     (loud 2.0 (at 3.0 (cue a-snd))))
```

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

Transformations can also be applied to groups of behaviors:

```
(loud 2.0 (sim (at 0.0 (cue a-snd))
               (at 0.7 (cue a-snd))))
```

2.7. Defining Behaviors

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

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

(play (snds 0.1))
(play (loud 2.5 (stretch 0.9 (snds 0.3))))
```

In the last line, `snds` is transformed: the transformations will apply to the `cue` behaviors within `snds`. The `loud` transformation will scale the sounds by `2.5`, and `stretch` will apply to the shift (`at`) amounts `0.0`, `0.7`, `1.0`, and `(+ 1.2 dly)`. The sounds themselves (copies of `a-snd`) will not be stretched because `cue` never stretches sounds.

Section 4.3 describes the full set of transformations.

3. More Examples

This chapter explores Fugue through additional examples. The reader may wish to browse through these and move on to Chapter 4, which is a reference section describing Fugue functions.

3.1. Stretching Sampled Sounds

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

```
; if demo4.lsp was not loaded, load sound sample:
;
(if (not (boundp 'a-snd))
    (setf a-snd
          (sf-load "/usr0/rbd/fugue/test/ex/demo-snd.nh"
                   22050.0)))

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

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

(play (down))

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

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

Notice the use of the `sound` behavior as opposed to `cue`. The `cue` behavior shifts and scales its sound according to `*time*` and `*volume*`, but it does not change the duration or resample the sound. In contrast, `sound` not only shifts and scales its sound, but it also stretches it by resampling according to the `*stretch*` factor in the environment. (The `*transpose*` element of the environment is ignored by both `cue` and `sound`.)

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

with no modulation input, and the result is a sine tone. The duration of the modulation determines the duration of the generated tone (when the modulation signal ends, the oscillator stops).

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

To summarize, `pwl` always expects an odd number of parameters. The resulting function is stretched according to `*stretch*`, and shifted according to `*time*`. Now, here is the example:

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

The same idea can be applied to a non-sinusoidal carrier. Here, we assume that `*fm-voice*` is predefined:

```
; do the same thing with a non-sine table
;
(play (fmosc cs2 (pwl 0.5 (step-to-hz cs2) 0.501)
                 0 *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:

with no modulation input, and the result is a sine tone. The duration of the modulation determines the duration of the generated tone (when the modulation signal ends, the oscillator stops).

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

To summarize, `pwl` always expects an odd number of parameters. The resulting function is stretched according to `*stretch*`, and shifted according to `*time*`. Now, here is the example:

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

The same idea can be applied to a non-sinusoidal carrier. Here, we assume that `*fm-voice*` is predefined:

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

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

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


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

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

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

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

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

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

4. Fugue Functions

This chapter provides a language reference manual for Fugue. Operations are categorized by functionality and abstraction level. Fugue 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.

The "low-level" primitives directly operate on sounds, but know nothing of environmental variables (such as `*time*`, `*stretch*`, etc.). The names of most of these low-level functions start with "`s-`". In general, programmers should avoid any function with the "`s-`" 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 operations are distinguished by the "`snd-`" prefix. To summarize:

no prefix: operation on behaviors
`snd-` prefix: commonly used operation on sounds
`s-` prefix: avoid using these

Fugue uses both linear frequency and equal-temperament pitch numbers to specify repetition rates. Frequency is always specified in cycles per second (hz), and pitch numbers, also referred to as "key numbers" (thanks to MIDI) are floating point numbers such that 48 = Middle C, 49 = C#, 49.23 is C# plus 23 cents, etc. The mapping from pitch number to frequency is the standard exponential conversion, and fractional pitch numbers are allowed: $frequency = 440 \times 2^{(pitch-69)/12}$

4.1. Sounds

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

4.1.1. 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, Fugue sounds are asymmetric: it is possible to

compute sequences forward in time by aligning the logical start of each sound with the `logical-stop` of the previous one, but one cannot compute "backwards", aligning the logical end of each sound with the logical start of its successor. The root of this asymmetry is the fact that when we invoke a behavior, we say when to start, and the result of the behavior tells us its logical duration. There is no way to invoke a behavior with a direct specification of when to stop[1].

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

4.1.2. Creating Sounds

The basic operations that create sounds are:

`(s-create)`
: Returns a sound which is silence. The duration is zero.

`(s-constant` *value duration*`)`
: Returns a sound of *duration*, with the constant *value* at the sample rate *srate*. ***Note.:*** since sounds are assumed to be zero at their start time,

`(s-compose` *array srate*`)`
: Takes a Lisp *array* of integers and/or floats, and converts them into a sound sample with the given *srate*.

`(snd-load` *filename srate*`)`
: Loads a sound file named by the string *filename* from disk. There is no header on the file, only continuous 16-bit words (MSB first). The sample is treated as if it were taken at sample-rate *srate* (in hz).

4.1.3. Accessing Sounds

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

`(snd-access` *sound time*`)`
: Retrieves the value of *sound* at *time*. If *time* is before or after *sound*, `0.0` is returned. The *sound* is linearly interpolated if *time* does not fall on an exact sample time.

`(s-samples` *sound limit*`)`
: Converts the samples into a lisp array. The data is taken directly from the samples, ignoring shifts. For example, if the sound starts at 3.0 seconds, the first sample will refer to time 3.0, not time 0.0. `s-extent` (see below) will tell you the time range for the *sound*. A maximum of *limit* samples is returned.

`(snd-srate` *sound*`)`
: Returns the sample rate of the sound.

`(snd-show` *sound*`)`

---

[1] Most behaviors will stop at their `*start*` + `*stretch*`, but this is by convention and is not a direct specification.

Print the entire tree structure of the sound. (See Appendix II.)

(`snd-stats` *sound*)
Displays vital statistics about a *sound*. If *sound* is a sample, `snd-stats` will also display the first and last 5 samples in the sound. The return value is *sound*. ***Note:*** since some operations are lazily evaluated, *sound* may point to an expression to be evaluated rather than computed samples. All other functions automatically evaluate a sound when samples are needed except for `snd-stats`. Lazy evaluation can be defeated by applying the `flatten` function, described below.

(`snd-extent` *sound*)
Returns a list of the time at which *sound* starts and the time at which it stops, i.e. the list (*signal-start signal-stop*).

(`snd-logical-stop` *sound*)
Returns the ''perceptual'' or logical stop time of a *sound*. ***Note:*** this is usually the same as the actual stopping time returned by `snd-extent` because all built-in functions set the logical stop time to `signal-stop`. The `set-logical-stop` operation can be used to set the logical stop time. The function `get-logical-stop` is identical to `snd-logical-stop` and should be used when defining behaviors.

(`snd-maxsamp` *sound*)
Returns the value of the maximum absolute value of any sample in *sound*.

4.1.4. Low-Level Manipulation Primitives

Low-level manipulation primitives provide basic operations that are implemented using lazy evaluation. These operations are *not* behavioral abstractions, hence they are immune to transformations, and for that reason should generally be avoided. They are used primarily in the implementation of the built-in behaviors described in the next section.

(`s-apply` *sound scale start stop shift stretch srate*)
Takes the given *sound*, scales its samples by *scale*, extracts the period between *start* and *stop*, shifts this in time by *shift*, and finally stretches this resulting sound by *stretch* (that is, the time shift and the extracted sample are both stretched). The `logical-stop` is shifted and stretched as well. If *srate* is `0.0`, then the sample rate of the sound is kept, otherwise, the sample is re-sampled to the sampling rate of *srate*. This resulting sound is returned.

(`s-scale` *sound factor*)
Returns a sound that is the same as *sound*, except each sample is multiplied by *factor*.

(`s-clip` *sound start stop*)
Returns a sound which is the portion of *sound* between the *start* and *stop* times.

(`s-lclip` *sound delta*)
Returns a sound which is the portion of *sound* with the first *delta* seconds removed.

(`s-rclip` *sound delta*)
Returns a sound which is the portion of *sound* with the last *delta* seconds removed.

(`s-shift` *sound amount*)
Returns a sound which is *sound* shifted in time (forwards or backwards) by *amount*.

(`s-stretch` *sound factor*)
Returns a sound which is *sound* stretched in time by *factor*.

(`s-set-logical-stop` *sound time*)
: Returns a sound which is *sound*, except that the logical stop of the sound occurs at *time*. When defining a behavior, use `set-logical-stop` instead.

4.1.5. Other Low-Level Primitives

In addition to the basic primitives of the previous section, there are a number of "unit generator" style operations. As before, *these are not behaviors*, so they are immune to transformations. Their main purpose is in the implementation of behaviors. For example, the `osc` function is implemented by a call to `s-osc`, after taking into account the current transformation environment.

(`s-add` *sound1 sound2*)
: Returns the sum of the two sounds. In most cases, the corresponding behavior `sim` should be called instead of `s-add`.

(`s-mult` *sound1 sound2*)
: Returns *sound1* multiplied by *sound2*. When the results of two behaviors should be multiplied, the corresponding operation `mult` should be called as a matter of style, even though it is identical to `s-mult`.

(`s-osc` *sound pitch srate freq duration phase periodic*)
: Returns a sound which is the *sound* oscillated for the given *duration* (in seconds), *frequency* (in hz), and *srate* (in hz). *Phase* currently indicates where in *sound* to begin (in radians[2]). *Pitch* is a pitch or key number indicating what pitch *sound* is, so that things can be appropriately resampled. (*sound* may be more than one period, so *pitch* is not redundant.) *Periodic* should be `T` if this sample is to be looped, or `nil` if this represents a non-periodic sample. The behavior `osc` should normally be called instead of `s-osc`.

(`s-amosc` *sound pitch srate pitch modulation phase periodic*)
: Returns a sound which is *sound* oscillated for the duration of sound *modulation*, where *pitch* is the pitch or key number indicating the pitch of *sound*, *srate* is the desired sample rate (in hz), *pitch* is the desired resultant pitch, *modulation* modulates the oscillator output (using multiplication), *phase* is the initial phase in radians, and *periodic* should be `T` if the sample is one period of a waveform, or `nil` if this is a sample that should not be looped. If *periodic* is `nil`, you might be happier just using `s-mult` to multiply the two signals. Even if *periodic* is `nil`, the resulting duration is still that of *modulation*, so there may be some trailing silence. The behavior `amosc` should normally be called instead of `s-amosc`.

(`s-fmosc` *sound pitch srate freq modulation phase periodic*)
: Returns a sound which is *sound* oscillated for the duration of sound *modulation*, where *pitch* is the pitch or key number indicating the pitch of *sound*, *srate* is the desired sample rate (in hz), *freq* is the desired resultant center frequency, *modulation* frequency-modulates the oscillator (by adding the modulation signal to an offset determined by *freq*), *phase* is the initial phase in radians, and *periodic* should be `T` if the *sound* is a period of a waveform, or `nil` to prevent looping. In the case where *periodic* is `nil`, the resulting duration is still that of *modulation*, so there may be some trailing silence. 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

---

[2] Phases are always in radians, but this can be changed by redefining the constant ANGLEBASE and recompiling.

frequencies are well defined but not currently implemented; instead the modulation signal (again, the frequency deviation is the sum of *freq* and *modulation*) is simply clipped to avoid negative frequencies. The behavior `fmosc` should normally be called instead of `s-fmosc`.

`(s-env` *srate* t_1 t_2 t_3 t_4 l_1 l_2 l_3`)`
Returns a sound which is an envelope as specified via the time and level parameters. The sample-rate for the given samples is *srate* (in hz). The total time is $t_1+t_2+t_3+t_4$ (in seconds) and the level at end of each time interval t_N is l_N. The starting level, l_0 is not a parameter and is always zero, as is the ending level, l_4. Normally, the behavior `env` should be called instead of `s-env`.

`(s-pwl` *srate list*`)`
Returns a piece-wise linear signal described by breakpoints. *Srate* is the sample rate of the result, and *list* is a list of breakpoints in the form (t_1 a_1 t_2 a_2 t_3 a_3 ... t_n). The breakpoints are (0, 0), (t_1, a_1), (t_2, a_2), ... (t_n, 0). Note the implicit zeros at the beginning and end. Normally, the behavior `pwl` should be used instead of `s-pwl`.

`(snd-save` *sound string*`)`
Saves a copy of *sound* into the file named by *string*. The samples are multiplied by 32767, rounded to the nearest integer and written as 16-bit signed integers.

`(s-copy` *sound*`)`
Returns a duplicate of *sound*.

`(s-flatten` *sound*`)`
Returns *sound* which has been normalized; any trees created by lazy-evaluation are flattened. The sample's `scale` and `stretch` will be `1.0`, the sound's `start` will be `0.0`, `stop` will be the (number of `samples`) / `srate`. The `shift`, `srate`, and `logical-stop` may be any values.

`(s-white-noise` *duration srate*`)`
Computes white noise for the given *duration* at the given sample rate. Normally, the behavior `noise` should be used instead.

`(s-lp` *sound cutoff*`)`
A first-order Butterworth low-pass filter is applied to *sound* with the specified *cutoff* frequency (a float) in hertz. Normally, the behavior `snd-lp` should be used instead.

`(s-lp-var` *sound cutoff*`)`
A first-order Butterworth low-pass filter is applied to *sound* with the specified variable *cutoff* frequency. *Cutoff* is a sound (signal) whose sample rate determines the rate at which filter coefficients are recomputed. Normally, the behavior `snd-lp` should be used instead.

`(s-hp` *sound cutoff*`)`
A first-order Butterworth high-pass filter is applied to *sound* with the specified *cutoff* frequency (a float) in hertz. Normally, the behavior `snd-hp` should be used instead.

`(s-hp-var` *sound cutoff*`)`
A first-order Butterworth high-pass filter is applied to *sound* with the specified variable *cutoff* frequency. *Cutoff* is a sound (signal) whose sample rate determines the rate at which filter coefficients are recomputed. Normally, the behavior `snd-hp` should be used instead.

`(s-reson` *sound center bandwidth*`)`

A resonating filter is applied to *sound* with the specified *center* frequency and *bandwidth*, both floats expressing hertz. The gain is unity at the center frequency. Normally, the behavior `snd-reson` should be used instead.

`(s-reson-var` *sound center bandwidth*`)`
A resonating filter is applied to *sound* with the specified variable *center* frequency (a signal) and constant *bandwidth* (a float). The gain is unity at the center frequency. Normally, the behavior `snd-reson` should be used instead.

4.1.6. Miscellaneous Operations

These functions provide some useful utility and query functions:

`(normalize` *sound*`)`
Return a scaled version of *sound* such that the maximum amplitude (absolute value) is 1.0.

`(play` *sound*`)`
Play the sound through the DAC. The `play` function writes the file `temp.snd` in the current directory as a NeXT sound file and plays the sound. If the sound sample rate is less than 22050, the sound is resampled to 22050. If the sound sample rate is between 22050 and 44100, the sound is resampled to 44100.

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

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

`(get-logical-stop` *sound*`)`
Returns the logical end time of *sound*.

4.2. Behaviors

4.2.1. Using Previously Created Sounds

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

`(cue` *sound*`)`
Applies `*volume*`, `*time*`, `*start*`, and `*stop*` to *sound*.

`(cue-file` *filename*`)`
Same as `cue`, except the sound comes from the named file, which is assumed to have the current default `*sound-srate*` sample rate.

`(sound` *sound*`)`
Applies `*volume*`, `*time*`, `*start*`, `*stop*`, `*stretch*`, and `*sound-srate*` to *sound*.

`(control` *sound*`)`
Applies `*volume*`, `*time*`, `*start*`, `*stop*`, `*stretch*`, and `*cntrl-srate*` to *sound*.

4.2.2. Sound Synthesis

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

(env t_1 t_2 t_4 l_1 l_2 l_3 [*dur*])

Creates a 4-phase envelope. t_i is the duration of phase *i*, and l_i is the final level of phase *i*. t_3 is implied by the duration *dur*, and l_4 is `0.0`. If *dur* is not supplied, then `1.0` is assumed. The envelope duration is the product of *dur*, `*stretch*`, and `*duty*`. If t_1 + t_2 + 2ms + t_4 is greater than the envelope duration, then a two-phase envelope is substituted that has an attack/release time ratio of t_1/t_4. The sample rate of the returned sound is `*cntrl-srate*`. (See `pwl` for a more general piece-wise linear function generator.)

(lfo *freq* [*duration osc phase*])

Just like `osc` (below) except this computes at the `*cntrl-srate*` and frequency is specified in hz. The `*transpose*` is not applied.

(mult beh_1 beh_2 ...)

Returns the product of behaviors.

(osc *pitch* [*duration table phase*])

Returns a sound which is the *table* oscillated at *pitch* for the given *duration*, starting with the *phase*. Defaults are: *duration* `1.0` (second), *table* `*table*`, *phase* `0.0`. ***Note:*** *table* is a list of the form

(*sound pitch-number periodic*)

where the first element is a sound, the second is the pitch of the sound (this is not redundant, because the sound may represent any number of periods), and the third element is `T` if the sound is one period of a periodic signal, or `nil` if the sound is a sample that should not be looped.

(amosc *pitch modulation* [*table phase*])

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

(fmosc *pitch modulation* [*table phase*])

Returns a sound which is *table* oscillated at *pitch* plus *modulation* for the duration of the sound *modulation*. *osc-table* defaults to `*table*`, and *phase* is the starting phase (default 0.0 radians) within *osc-table*. The *modulation* is expressed in hz, e.g. a sinusoid modulation signal with an amplitude of 1.0 (2.0 peak to peak), will cause a +/– 1.0 hz frequency deviation in *sound*. Negative frequencies are well defined but not currently implemented; instead the *modulation* signal is simply clipped to avoid negative frequencies.

(pwl t_1 t_2 l_2 ... t_n)

Creates a piece-wise linear envelope with breakpoints at (0, 0), (t_1, l_1), (t_2, l_2), ... (t_n, 0). The envelope is stretched by `*stretch*` and `*duty*`, the sample rate is `*control-srate*`, and the envelope shifted by `*time*`. Note that the times are relative to 0; they are not durations of each envelope segment.

(osc-note *pitch* [*duration env volume table*])