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

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.

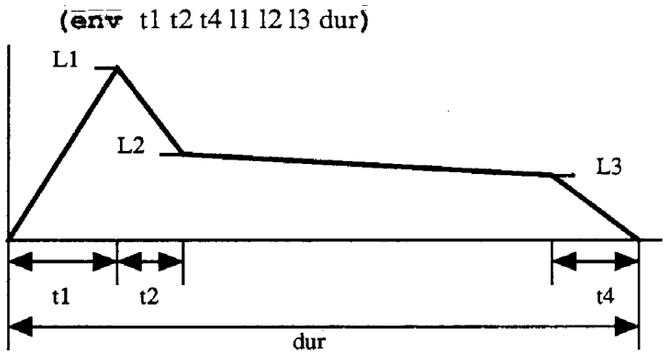

Figure 1: An envelope generated by the `env` function.

```
; 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*])

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).

`(rest [`*duration*`])`
Create silence for the given *duration*. Default duration is 1.0 sec, and duration is scaled by `*stretch*` and shifted by `*time*`.

`(noise [`*duration*`])`
Generate noise with the given *duration*. Duration (default is 1.0) is scaled by `*stretch*` and shifted by `*time*`. The sample rate is `*sound-srate*` and the amplitude is +/- `*volume*`.

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

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

`(snd-reson` *sound center bandwidth*`)`
Apply a resonating filter to *sound* with center frequency *center* (in hertz), which may be a float or a signal. *Bandwidth* is a float and therefore constant for the duration of *sound*. Filter coefficients (requiring trig functions) are recomputed at the sample rate of *cutoff*.

4.3. 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 with an "abs-" prefix) do not look at the current environment, but rather set an environment variable to a specific value. In this way, sections of code can be insulated from external transformations.

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

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

`(at-abs` *time beh*`)`
Evaluate *beh* with `*time*` set to *time*.

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

`(extract` *start stop beh*`)`
Returns a sound which is the portion of *beh* between *start* and *stop*. Note that this is done relative to the current `*time*`. The result is shifted to start at `*time*`, so normally the result will start at `*time*` rather than `*time*` + *start*.

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

`(loud` *volume beh*`)`
Evaluates *beh* with `*volume*` scaled by *volume*.

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

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

`(stretch` *factor beh*`)`
Evaluates *beh* with `*stretch*` scaled by *factor*.

`(stretch-abs` *factor beh*`)`
Evaluates *beh* with `*stretch*` set to *factor*.

`(trans` *amount beh*`)`
Evaluates *beh* with `*transpose*` shifted by *amount*.

`(trans-abs` *amount beh*`)`
Evaluates *beh* with `*transpose*` set to *amount*.

4.4. Combination and Time Structure

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

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

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

`(sim` beh_1 `[`beh_2 `...])`
Returns a sound which is the sum of the given behaviors evaluated with current value of `*time*`.

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

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

Appendix I
ICMC Conference Paper on Fugue

Fugue: Composition and Sound Synthesis With Lazy Evaluation and Behavioral Abstraction

Roger B. Dannenberg
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Christopher Lee Fraley
Microsoft Corporation
166011 NE 36 Way
Box 97017
Redmond, WA 98073-9717

Abstract

Fugue is an interactive language for music composition and synthesis. The goal of Fugue is to simplify the task of generating and manipulating sound samples while offering greater power and flexibility than other software synthesis languages. In contrast to other computer music systems, sounds in Fugue are abstract, immutable objects, and a set of functions are provided to create and manipulate these sound objects. Fugue directly supports behavioral abstraction whereby scores can be transformed using high-level abstract operations. Fugue is embedded in a Lisp environment, which provides great flexibility in manipulating scores and in performing other related symbolic processing. The semantics of Fugue are derived from Arctic and Canon which have been used for composition, research, and education at Carnegie Mellon University for several years.

Introduction

Fugue is a language for composition and sound synthesis. Features of Fugue include: (1) a full interactive environment based on Lisp, (2) a language which does not force a high-level distinction between the "score" and the "orchestra", (3) support for behavioral abstraction, (4) the ability to work both in terms of actual and perceptual start and stop times, and (5) a time- and memory-efficient implementation.

The Lisp environment provides an interactive interface, flexibility in manipulating sounds, and a base for performing other related symbolic processing. Sounds are first-class types in Fugue, hence they can be assigned to variables, passed as parameters, and stored in data structures. Storage for sounds is dynamically allocated as needed and reclaimed by automatic garbage collection. This allows "instruments" to be implemented as ordinary Lisp functions and eliminates the orchestra/score dichotomy.

Fugue semantics include behavioral abstraction as introduced by Arctic (Dannenberg,

McAvinney, and Rubine 1986) and Canon (Dannenberg 1989). The motivation for behavioral abstraction is the idea that one should be able to describe behaviors that respond appropriately to their environment. For example, stretching a sound may mean one thing in the context of granular synthesis and another in the context of sampling. It almost never means to compute a short sound and then resample it to make it longer. Fugue allows the programmer to describe abstract behaviors that "know" how to stretch, transpose, change loudness, and shift in time. Transformation operators are provided to operate on these abstractions.

Composition requires that sounds be placed simultaneously together, in sequence, and at arbitrary offsets. Because musical sounds often have attack and release portions, we make a distinction between the absolute first and last samples of a sound and the perceptual start and end to which other sounds should be aligned.

Fugue is designed with powerful workstations in mind. The current implementation relies upon virtual memory and a large disk memory to eliminate the need for explicit file access and buffer management. The low-level operations in Fugue are amenable to implementation on array processors or DSP chips when these are available. Fugue uses lazy evaluation to achieve reasonable performance without sacrificing its clean semantics.

Related Work

Many software synthesis and compositional systems have existed for years, each encountering and addressing a slightly different set of problems. To understand Fugue, it is beneficial to first review some other systems.

Music V takes a semi-functional approach to sound *generation* in that unit generators can be combined as functions applied to sample streams. The resulting instruments can be applied to parameter lists. Instruments cannot be applied to other instruments, nor can scores be constructed hierarchically. This division between sound manipulators (or generators) and parameter lists results in a corresponding separation between the orchestra and the score. Other consequences include a non-interactive environment. An interesting aspect of Music V is the idea that an *instance* of an instrument is created for each note specified in the score.

Kyma (Scaletti 1989) and the Sun/Mercury Workstation (Rodet and Eckel 1988) take a different approach, treating sound manipulators and sound generators as objects that can be "patched" together. This results in an intuitive system for synthesis, but there are problems. A level of indirection is required to manipulate graphs of unit generators which in turn manipulate sounds rather than to manipulate sounds directly. Various extensions have become necessary in order to handle graphs that change over time, but this also adds complexity to programs. Symbol processing and working with data structures are also difficult with these graph-oriented program representations. Both systems run all objects in synchrony, thereby assuming a global sample rate.

SRL (Kopec 1985) is a signal processing language that represents signals as parameterized computations. SRL signals are immutable objects that can be reused. SRL supports lazy evaluation and function caching by retaining a symbolic representation of all signals. SRL lacks in many musically useful concepts such as the starting time and duration of signals and

behavioral abstraction. Also the user must explicitly free buffers when they are no longer needed.

Formes (Cointe and Rodet 1984) takes an object-oriented approach to the computation of functions of time, but Formes was not designed to compute audio directly. Formes was originally designed to compute control information for the Chant synthesis system.

Behavioral Abstraction

Fugue provides an elegant and hierarchical way to express scores that combines qualities of both note lists and executable programs. Note lists of classical score languages are attractive because they can be generated, stored, and manipulated as data. For example, making all the notes in a section louder is easy to do if the notes are represented as data. On the other hand, note lists suffer from the fact that they are not programs. In particular, there comes a time when "loudness" (and every other note-list parameter) must be interpreted to produce or control sound. The point at which interpretation starts defines the boundary between the "score" and the "orchestra".

Fugue avoids the boundary through the use of declarative-style programs that "feel" like note lists and by using the same language to define both scores and synthesis procedures. It is possible to alter Fugue scores by applying various transformations along the dimensions of time, loudness, pitch, articulation, and even sample rate.

A potential liability of these transformations is that they may transform the wrong thing. For example, in stretching a section of music that contains a trill, we do not necessarily want the trill to slow down, and we almost certainly do not want the pitch to drop! Fugue provides defaults for transformations, but allows the programmer/composer to override the defaults with more appropriate behaviors.

Thus, the programmer/composer defines behaviors that "do the right thing" in the context of a specified set of transformations. The definition of a class of behaviors that are realized according to a context is called *behavioral abstraction.* A few examples should clarify how Fugue works. The first example is a sequence of three sounds:

```
(seq (cue wind) (cue water) (osc Bf3))
```

where `cue` is a behavior that simply plays a sound at a given time, and `osc` is a behavior that plays a given pitch. `wind` and `water` are two sounds, perhaps loaded from sound files. If we wanted to hear the same sequence at a lower amplitude and with the `water` sound delayed by 2 seconds, we could write:

```
(loud 0.2
      (seq (cue wind) (at 2.0 (cue water)) (osc Bf3)))
```

Now suppose we wish to change the pitch. We could write

```
(transpose 3 (seq (cue wind) (cue water) (osc Bf3)))
```

This would have the effect of transposing the sequence up by 3 semitones. However, since the `cue` abstraction overrides and prevents transposition, only the `osc` behavior will be affected.

Signal Processing

Fugue is intended as a versatile system for the analysis, synthesis, and processing of sound. Thus far, our efforts have focussed on building an extensible kernel for Fugue and implementing some simple synthesis primitives. In the current implementation, sounds may be obtained using a generalized oscillation function or by loading sounds from files.

Primitives are also supplied to manipulate the environment in which sounds are generated and composed. These operations are used to perform cutting and splicing, stretching, controlling the amount of *legato* (sound overlap), loudness, and pitch. These operations manipulate the environment in which sounds are computed and may be applied from the score level all the way down to sound generation primitives.

System Organization

Fugue is implemented in a combination of C (Kernighan and Richie 1978) and XLisp (Betz 1986). We use XLisp because it is fairly easy to extend with a new type. (XLisp is itself written in C.) The use of two languages reflects our goal to provide an interactive and efficient environment. New synthesis techniques can be introduced by combining existing Lisp functions on sounds or by writing new sound synthesis algorithms in C and making them callable from Lisp.

Multiple sample rates allow "control" signals to exist at a low sample rate as in Music-11 (Vercoe 1981) and Csound (Vercoe 1986), reducing time and memory requirements. Linear interpolation is used (by default) when it becomes necessary to manipulate two sounds with different sample rates. There is no distinction between control signals and audio signals. Filters can be used to modify spectra or to smooth envelopes, and multiplication can be used uniformly for gain control, amplitude envelopes, or audio-rate amplitude modulation.

The fact that sounds in Fugue are immutable values implies that the implementation cannot add several sounds directly into a buffer. If sounds are immutable, then each addition of two sounds produces a new sound and requires storage allocation. One might expect an implementation with immutable values to be very inefficient, but we avoid this problem through lazy evaluation. When additions (and many other operations) are performed, our implementation merely builds a small data-structure describing the desired operation without actually computing any samples. This technique avoids many redundant copy operations but is completely hidden from the user.

Also hidden from the user is the use of reference counting (Pratt 1975) to reclaim storage from sounds that are no longer referenced. This reference counting scheme is integrated with the XLisp mark-and-sweep garbage collector (Schorr and Waite 1967).

Conclusion

Fugue is a new language that provides high-level operations on sounds. Fugue is unique in that is spans a range of computational tasks from score manipulation to synthesis within a single integrated language. Fugue already has an efficient implementation running on Unix workstations. We intend to improve this further by taking advantage of virtual copy and mapped file capabilities of the Mach (Accetta *et. al.* 1986) operating system and a DSP chip for signal processing. We also plan to extend Fugue with more sound functions from other systems such as Moore's Cmusic (Moore 1982), Vercoe's Csound (Vercoe 1986), NeXT's Sound Kit (Jaffe and Boynton 1989), and Lansky's Cmix (Lansky 1987).

References

Accetta, M., Baron, R. Bolosky, W., Golub, D., Rashid, R., Tevanian, A., and Young, M. "Mach: A New Kernel Foundation for UNIX Development." *Proc. of Summer Usenix*, July 1986.

Betz, D. 1986. XLISP: An Experimental Object-oriented Language, Version 1.7. (program documentation).

Cointe, P. and Rodet, X. 1984. "Formes: an Object & Time Oriented System for Music Composition and Synthesis." In *1984 Symposium on LISP and Functional Programming*. ACM Press, pp. 85-95.

Dannenberg, R. B., McAvinney, P., Rubine, D. 1986. "Arctic: A Functional Language for Real- Time Systems." *Computer Music Journal* 10(4):67-78.

Dannenberg, R. B. 1989. "The Canon Score Language." *Computer Music Journal* 13(1):47-56.

Jaffe, D. and Boynton, L. 1989. "An Overview of the Sound and Music Kits for the NeXT Computer." *Computer Music Journal* 13(2):48-55.

Kernighan, B. M. and Richie, D. M. 1978, *The C Programming Language*. Englewood Cliffs: Prentice-Hall.

Kopec, G. E. 1985. "The Signal Representation Language SRL." *IEEE Transactions Acoustics, Speech and Signal Processing*. 33(4):921-932.

Lansky, P. 1987. "CMIX". Princeton Univ. (Software and documentation).

Moore, F. R. 1982. "The Computer Audio Research Laboratory at UCSD." *Computer Music Journal* (6)1:18-29.

Pratt, T. 1975. *Programming Languages: design and implementation*. Englewood Cliffs: Prentice Hall.

Rodet, X. and Eckel, G. 1988. "Dynamic Patches: Implementation and Control in the Sun-Mercury Workstation." In *Proceedings of the 1988 International Computer Music Conference*. Computer Music Association. pp 82-89.

Scaletti, C. 1989. "The Kyma/Platypus Computer Music Workstation." *Computer Music Journal* 13(2):23-38.

Schorr, H. and Waite, W. 1967. "An Efficient and Machine Independent Procedure for Garbage Collection in Various List Structures." *Comm. ACM* 10(8):501-506.

Vercoe, B. 1981. *Reference Manual for the MUSIC 11 Sound Synthesis Language*. MIT Experimental Music Studio.

Vercoe, B. 1986. *CSOUND: A Manual for the Audio Processing System and Supporting Programs*. MIT Media Lab.

Appendix II
Lazy Evaluation

The Fugue implementation uses *lazy evaluation* of sounds to implement nice language semantics without losing efficiency. I will describe the rationale behind the semantics, show where a potential problem arises, and then show how lazy evaluation solves the problem.

Fugue semantics say that sounds are *values* just like the number 3. Values (including sounds) cannot change because allowing changes to sounds would admit the possibility of unwanted side-effects. An example of a side-effect is passing a sound as a parameter to a function that modifies the sound. Such modifications are sometimes avoided by making copies of parameters. However, since Fugue guarantees the absence of side-effects, there is no need to make copies of sounds (why make a copy if you know it will not change?). In fact there is no `s-copy` function in Fugue. This all makes it very easy and efficient to pass sounds around as parameters, store them in variables, etc. In the implementation, assignments simply generate new references to the sole representation of a sound. This is semantically equivalent to a copy because the value of the sound can never change.

To maintain this semantic requirement, each operation, such as adding two sounds together, requires that new storage has to be allocated and data must be transferred from the two source sounds to the destination (the sum). This is especially costly when the goal is to add a series of short sounds (e.g. notes) to a large composition. In this case, the entire composition must be copied for each note. Without the copy, however, the value semantics would not be preserved.

One way to avoid this problem is to simply abandon the value semantics and allow sounds to be modified. A better approach, chosen for Fugue, maintains the nice value semantics and is still quite efficient. The solution, called *lazy evaluation*, defers storage allocation and computation until it is really needed. Usually, optimizations can be made on the deferred computation to avoid what would otherwise require many copies.

When sounds are created and manipulated via the supplied primitives, sound samples are generally not computed. Instead of allocating new memory for completely new samples whenever a function is performed on a sound, an evaluation tree is built instead. This tree can represent the sum of two sounds, scaling sounds by a constant, extracting a portion of a sound, shifting sounds in time, stretching sound in time, and converting sounds to a new sample-rate.

The Fugue function `snd-show` can be used to examine the evaluation tree. For example, if the following code is executed:

```
(setf mysnd (osc c4))
(setf s (loud 2.0
              (seq (loud 0.02 (cue mysnd))
                   (loud 0.1 (cue mysnd))
                   (cue mysnd))))
(snd-show s)
```

then the structure of `s` will be printed as follows:

```
{85c3c} tag:SUMNODES
   from:0  to:3  shift:0  stretch:1
   logTo:3  scale:1  srate:22050  refCount:1
   {855a8}->
   {85c00} tag:SUMNODES
      from:0  to:2  shift:0  stretch:1
      logTo:2  scale:1  srate:22050  refCount:2
      {85ac0}->
      {85b88} tag:SAMPLES
         from:0  to:1  shift:0  stretch:1
         logTo:1  scale:0.04  srate:22050  refCount:3
         sound: {85558}
            refCount: 5  len:22050  srate:22050  data:{99fd0}
      {856cc}->
      {85bc4} tag:SAMPLES
         from:0  to:1  shift:1  stretch:1
         logTo:2  scale:0.2  srate:22050  refCount:2
         sound: {85558}
            refCount: 5  len:22050  srate:22050  data:{99fd0}

   {855b4}->
   {8fac0} tag:SAMPLES
      from:0  to:1  shift:2  stretch:1
      logTo:3  scale:2  srate:22050  refCount:2
      sound: {85558}
         refCount: 5  len:22050  srate:22050  data:{99fd0}

{SOUND: {85c3c}  node[0:3|3]@22050}
```

The `tag:SUMNODES` indicates the structure represents the sum of sounds. This is followed by parameters that apply to the whole sum, e.g. the sound runs from time 0 to time 3. The `SUMNODES` structure has pointers (shown by `->`) to two sounds, the first of which is another `SUMNODES` structure. This one points to two `SAMPLES` structures which indicate transformations on actual samples. The samples can be shared by many `SAMPLES` nodes. In this example, all three `SAMPLES` nodes point to the `sound` structure at address `85558`. This sound has a reference count of 5 (3 references are shown here, the variable `mysnd` also points to the sound, and the 5th reference is probably from a temporary variable that will be garbage collected in the future).

Functions which operate on actual samples as opposed to a lazy-evaluation tree must first "flatten", or normalize, their signal parameters so that the resulting sound is an array of samples with no scale, from, to, stretch, and sample rate change to be applied. Note that shift is not in this list — it is allowed to remain non-zero to avoid an initial silence (zeros), saving both evaluation time and memory.

The `s-flatten` function converts any sound into a `SAMPLES` node by performing any computation implied by transformation parameters or by a `SUMNODES` tree. The following transcript shows what happens to the sound *s* computed above:

```
> (setf scopy s)
{SOUND: {85c3c}  sample[0:3|3]@22050}
> (s-flatten s)
{SOUND: {85c3c}  sample[0:3|3]@22050}
> (snd-show s)
{85c3c} tag:SAMPLES
   from:0  to:3  shift:0  stretch:1
   logTo:3  scale:1  srate:22050  refCount:3
   sound: {9074c}
      refCount: 1  len:66150  srate:22050  data:{af864}
{SOUND: {85c3c}  sample[0:3|3]@22050}
>
```

This shows that a new sound was allocated and the `SUMNODES` tree was collapsed. Notice that

the address of the sound structure, $85c3c_{hex}$, remains the same, indicating that the structure is modified in place, and any copy of `s`, such as `scopy`, will also see the change. (This is a desirable side-effect because all copies of `s` will continue to share memory and another flatten operation is not needed for the copies.)

Normally, an internal version of `s-flatten` is called automatically by all sound processing functions, so there is no need write calls to `s-flatten`.

There are three basic data structures which comprise this lazy-evaluation tree: the sample structure, the node structure, and the sound structure. Below, each is described in detail.

II.1. Data Types

There are two data types which specifically deal with samples:

```
typedef short SFDataType, *SFDataPtr;
typedef float SDataType,  *SDataPtr;
```

`SFDataType` defines the type of data in sound files (on disk), while `SDataType` defines the type of sample data used internally. Note that if `SFDataType` is changed, some routines dealing with loading/storing sound files may need to be modified. `SDataType` may be changed to double without any problems, but changing to an integer type will require extensive changes to code to deal with overflows, resolution problems, etc.

II.2. The Sample Structure

Samples are managed by the following structure:

```
typedef struct
   {
   int      refCount;
   int      length;
   float    srate;
   SDataPtr data;
   } SampleType, *SamplePtr;
```

`RefCount` is used to keep track of how many times this structure is referenced by others. This count must be incremented when something references this structure, and must be decremented when something stops referencing this structure. If the count goes to zero, this structure must be freed.

`Length` indicates the length (in samples) of this sample. `Srate` is the sampling rate of this sample. `Data` is a pointer to the actual samples.

Note that this structure can only be referenced by something of `SoundType`.

II.3. The Node Structure

This structure is used to indicate the sum of one or more sounds. Note that this is an arithmetic sum, and takes into account any time shifts, etc.

```
typedef struct
   {
   SoundPtr sound;
   NodePtr  node;
   } NodeType, *NodePtr;
```

This structure forms one element in a linked list of nodes. Each element in this list points to a sound which is to be simultaneously summed with the other sounds in this list.

`Sound` is a pointer to one of the sounds in this sum, `node` is a pointer to the next node. If `node` is `NULL`, then this is the last node in the linked list.

II.4. The Sound Structure

The sound structure contains several elements which facilitate lazy evaluation when sounds are scaled, clipped, shifted in time, stretched in time, and added. The structure is defined as follows:

```
typedef struct
   {
   float  scale;
   double from, to, shift;
   float  stretch, srate;
   double logstop;
   int    refCount;
   short  tag;
   union
      {
      SamplePtr sample;
      NodePtr   node;
      }   ptr;
   } SoundType, *SoundPtr;
```

`Ptr` points to either a node or a sample. If it points to a node, this represents a lazy add: this sound is really the sum of several other sounds in the linked list pointed to by `ptr.node`. Alternatively, `ptr` may point directly to a sample. `Tag` indicates what type `ptr` is: `tag` may be `SAMPLES` or `NODES` or `SILENCE`. `SILENCE` is a special case, and indicates that this sound contains no samples.

The `scale`, `from`, `to`, `shift`, `stretch`, `srate`, and `logstop` fields indicate how the sample (when `tag` is `SAMPLES`) or the sum of other sounds (when `tag` is `NODES`) is to be lazy-evaluated. For clarity, these fields will be described when applied to samples. When applied to a sum of sounds, the result is the same as if these sounds were added together into a single sample, and these fields were applied to this resulting sample.

`scale` is a constant by which each sample is to be multiplied. No scaling occurs when `scale` is `1.0`.

`from` and `to` indicate what portion of a sample to extract. The units of `from` and `to` are seconds. Note that the time origin is moved to the beginning of the extracted sound. For example, if `from` is `10.0`, then the signal is shifted back by `10.0` seconds so that what was at time `10.0` is now at the origin. No extraction occurs if `from` is `0.0` and `to` is beyond the extent of the sound.

`shift` says to shift in time by `shift` seconds. If `shift` is positive, this is

equivalent to splicing silence onto the beginning of the sample. If negative, this can have the effect of setting a logical start time for this sound: In order to make the logical start time of a sound correspond to a given starting `*time*`, it is convenient to shift the physical starting time back. Note that the shift is applied after the extraction of a portion of a sound. A shift of 0.0 indicates the sample is not shifted at all.

`stretch` takes a sound and stretches it in time. This usually entails resampling the sound by linear interpolation. Since this is applied after the extraction and shift, the sound may be shifted further in time by the amount of `stretch`.

`srate` indicates the sample rate that this sound will have when lazy-evaluated.

`logstop` indicates the logical or perceptual end of the sound as opposed to `to` which indicates the absolute end of (the last sample in) the sound.

Also note that some of these transformations are interactive (non-associative), such as from, to, shift, stretch, and logical-stop. Hence, defining the order in which they are to be applied is important. By definition, the order is from, to, shift, stretch, then logical-stop. This means that the flatten operation does the following:

1. Scale the signal by scale;
2. Clip the signal on the left at from;
3. Clip the signal on the right at to;
4. Shift the signal by shift;
5. Stretch the signal by stretch;
6. Assert the logical-stop.

Hence, the normalized form for a sound is as follows:

```
    scale = 1.0
     from = 0.0
       to = (length of sample) / (srate of sample)
    shift = ???
  stretch = 1.0
    srate = (srate of sample)
  logstop = ???
      tag = SAMPLES
```

By definition logical-stop is the logical end of a sound, after from, to, shift, and stretch have been applied to the sample. Hence, it is not affected by flattening.

Once you are done using a sound that you have created, you must call this routine. This function `free()`'s all the structures referenced by the given `sound` if they are not still in use by someone else.

`void nodes_free(NodePtr node)`
Does the same for nodes as `sound_free` does for sounds.

`void sample_free(SamplePtr sample)`
Does the same for samples as `sound_free` does for sounds. Note this function will `free()` both the sample structure *and* the sound samples themselves.

III.3. Manipulators

`int soundp(LVAL lval)`
Takes an XLisp `LVAL` and returns `TRUE` if this is a `sound`, and `FALSE` otherwise. See XLisp documentation for info in `LVAL`'s.

`SoundPtr s_flatten(SoundPtr sound)`
"flattens" a sound by forcing evaluation. See Appendix II for more on lazy evaluation and the flattening process. One must `s_flatten()` before accessing any of a sound's sample data. After `s_flatten()` is called, one must still make sure `sound->tag` is `SAMPLES`, and not `SILENCE`.

III.4. fugue.c

Any of the functions declared in the header file `fugue.h` may also be called from C, as can any new functions you create. When using the functions found in `fugue.c`, one must be sure to `#include fugue.h` *after* `sound.h` is `#include`'d. These functions behave the same way when called from C as their corresponding XLisp functions behave when called from XLisp. Here is the list of functions from `fugue.h`:

```
float     s_maxSample();      /* LISP: (S-MAXSAMP SOUND)                          */
float     s_from();             /* LISP: (S-FROM SOUND)                           */
float     s_dur();              /* LISP: (S-DUR SOUND)                            */
float     s_to();               /* LISP: (S-TO SOUND)                             */
LVAL      s_samples();          /* LISP: (S-SAMPLES SOUND FIXNUM)                 */
SoundPtr  s_show();             /* LISP: (SND-SHOW SOUND)                         */
float     s_srate();            /* LISP: (SND-SRATE SOUND)                        */
SoundPtr  s_stats();             /* LISP: (S-STATS SOUND)                         */
float     s_access();           /* LISP: (SND-ACCESS SOUND FLONUM)                */
SoundPtr  s_apply();
      /* LISP: (S-APPLY SOUND FLONUM FLONUM FLONUM FLONUM FLONUM FLONUM)         */
SoundPtr  s_scale();             /* LISP: (S-SCALE SOUND FLONUM)                  */
SoundPtr  s_clip();               /* LISP: (S-CLIP SOUND FLONUM FLONUM)           */
SoundPtr  s_lclip();             /* LISP: (S-LCLIP SOUND FLONUM)                  */
SoundPtr  s_rclip();             /* LISP: (S-RCLIP SOUND FLONUM)                  */
SoundPtr  s_shift();             /* LISP: (S-SHIFT SOUND FLONUM)                  */
SoundPtr  s_stretch();          /* LISP: (S-STRETCH SOUND FLONUM)                 */
SoundPtr  s_add();           /* LISP: (S-ADD SOUND SOUND)                         */
SoundPtr  s_mult();              /* LISP: (S-MULT SOUND SOUND)                    */
SoundPtr  s_osc();
     /* LISP: (S-OSC SOUND FLONUM FLONUM FLONUM FLONUM FLONUM FIXNUM)           */
SoundPtr  s_env();
 /* LISP: (S-ENV FLONUM FLONUM FLONUM FLONUM FLONUM FLONUM FLONUM FLONUM)       */
SoundPtr  s_pwl();               /* LISP: (S-PWL FLONUM ANY)                     */
SoundPtr  sf_load();              /* LISP: (SF-LOAD STRING FLONUM)               */
SoundPtr  sf_save();              /* LISP: (SF-SAVE SOUND STRING)                */
double    step_to_hz();              /* LISP: (STEP-TO-HZ FLONUM)                */
double    hz_to_step();              /* LISP: (HZ-TO-STEP FLONUM)                */
double    s_logicalTo();         /* LISP: (S-LOGICALTO SOUND)                    */
SoundPtr  s_setLogicalTo();      /* LISP: (S-SETLOGICALTO FLONUM SOUND)          */
double    log();                 /* LISP: (LOG FLONUM) */
```

```
/*
 * FILE: example2.c
 *   BY: Christopher Lee Fraley
 *
 * 1.0 ( 3-JUN-89) - created. (clf)
 */

#include <math.h>
#include "xlisp.h"
#include "sound.h"


SoundPtr round(sound, k)
SoundPtr sound;
int k;
{
   SoundPtr val;
   SDataPtr data;
   int len, newLen;
   double newSrate;
   int i;

   /* Must flatten sound before I can access its samples: */
   (void)s_flatten(sound);
   len = sound->ptr.sample->length;

   /* Let's create the return value: */
   val = s_create();
   newLen = len;          /* Make return sound the same length as input sound */
   newSrate = sound->srate;      /* Make val have same srate as input sound */
   data = sdata_create(newLen);
   val->ptr.sample = spl_create(data, newLen, newSrate);
   val->tag = SAMPLES;
   val->to = val->logicalTo = newLen / newSrate;
   val->srate = newSrate;
   val->shift = sound->shift;

   for (i=0; i<len && i<newLen; i++)
     {
     /* Put your own calculations here, for example: */
     val->ptr.sample->data[i] = floor(sound->ptr.sample->data[i] / k) * k;
     }
   return (val);
}
```

Again, we need to provide a `.h` file to interface the new C code to Lisp via Intgen:

```
/*
 * FILE: example2.h
 *   BY: Christopher Lee Fraley (cf0v@spice.cs.cmu.edu)
 * DESC: example of extending the Fugue/XLisp system.
 *
 * 1.0 (3-JUN-89) - created. (cf0v)
 */

#include "sound.h"

SoundPtr round();               /* LISP: (ROUND SOUND FIXNUM) */
```

Appendix IV
Intgen

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

The interface generator will take as command-line input:

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

The output is:

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

For example, the command line

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

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

IV.0.1. Extending Xlisp

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

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

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

and edit `localptrs.h` to contain:

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

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

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

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

IV.1. Header file format

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

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

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

Legal type_names are:

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

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

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

Parameter types may be any of the following:

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

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

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

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

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

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

IV.2. Using #define'd macros

If a comment of the form:

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

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

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

will implement the LeLisp function LOGSHIFT.

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

IV.3. Lisp Include Files

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

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

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

```
(any lisp expression)
```

to *name*.lsp, where *name* is the interface name passed on the command line. If none of the include files examined have comments of this form, then no *name*.lsp file is generated.

IV.4. Example

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

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

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

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

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

IV.5. More Details

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

Appendix V
XLISP: An Object-oriented Lisp

Version 2.0

February 6, 1988

by
David Michael Betz
127 Taylor Road
Peterborough, NH 03458

(603) 924-6936 (home)

V.1. Introduction

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

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

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

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

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

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

V.2. A Note From The Author

If you have any problems with XLISP, feel free to contact me for help or advice. Please remember that since XLISP is available in source form in a high level language, many users 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.

V.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.

V.4. Break Command Loop

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

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

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

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

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

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

V.5. Data Types

There are several different data types available to XLISP programmers.

- lists
- symbols

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

V.6. The Evaluator

The process of evaluation in XLISP:

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

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

V.7. Lexical Conventions

The following conventions must be followed when entering XLISP programs:

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

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

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

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

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

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

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

- `\\` means the character "\"
- `\n` means newline
- `\t` means tab
- `\r` means return
- `\f` means form feed
- `\nnn` means the character whose octal code is nnn

V.8. Readtables

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

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

In the case of `:TMACRO` and `:NMACRO`, the *fun* component is a function. This can either be a built-in readmacro function or a lambda expression. The function should take two parameters. The first is the input stream and the second is the character that caused the invocation of the

readmacro. The readmacro function should return NIL to indicate that the character should be treated as white space or a value consed with NIL to indicate that the readmacro should be treated as an occurence of the specified value. Of course, the readmacro code is free to read additional characters from the input stream.

XLISP defines several useful read macros:

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

V.9. Lambda Lists

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

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

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

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

The &rest argument is followed by the &key arguments. When a keyword argument is passed to a function, a pair of values appears in the argument list. The first expression in the pair should evaluate to a keyword symbol (a symbol that begins with a ":"). The value of the second

expression is the value of the keyword argument. Like &optional arguments, &key arguments can have initialization expressions and supplied-p variables. In addition, it is possible to specify the keyword to be used in a function call. If no keyword is specified, the keyword obtained by adding a ":" to the beginning of the keyword argument symbol is used. In other words, if the keyword argument symbol is `foo`, the keyword will be `':foo`.

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

Here is the complete syntax for lambda lists:

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

where:

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

V.10. Objects

Definitions:

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

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

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

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

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

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

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

V.11. The "Object" Class

`Object` — the top of the class hierarchy.

Messages:

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

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

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

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

V.12. The "Class" Class

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

Messages:

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

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

- `*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.

V.14. Evaluation Functions

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

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

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

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

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

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

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

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

(macroexpand *form*) — recursively expand macro calls

form — the form to expand
returns — the macro expansion

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

V.15. Symbol Functions

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

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

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

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

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

(gensym [*tag*]) — generate a symbol

tag — string or number
returns — the new symbol

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

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

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

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

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

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

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

V.16. Property List Functions

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

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

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

V.17. Array Functions

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

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

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

V.18. List Functions

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

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

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

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

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

(first *expr*) — a synonym for car

(second *expr*) — a synonym for cadr

(third *expr*) — a synonym for caddr

(fourth *expr*) — a synonym for cadddr

(rest *expr*) — a synonym for cdr

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

(list *expr...*) — create a list of values
expr — expressions to be combined into a list
returns — the new list

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

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

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

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

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

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

(remove-if *test list*) — remove elements that pass test
test — the test predicate

list — the list
returns — copy of list with matching elements removed

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

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

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

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

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

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

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

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

(subst *to from expr* &key :test :test-not) — substitute expressions
to — the new expression
from — the old expression
expr — the expression in which to do the substitutions

:test — the test function (defaults to eql)
:test-not — the test function (sense inverted)
returns — the expression with substitutions

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

V.19. Destructive List Functions

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

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

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

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

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

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

(sort *list test*) — sort a list

list — the list to sort
test — the comparison function
returns — the sorted list

V.20. Predicate Functions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

(eql *expr1 expr2*) — are the expressions identical? (works with all numbers)
expr1 — the first expression
expr2 — the second expression
returns — `t` if they are equal, `nil` otherwise

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

V.21. Control Constructs

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

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

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

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

expr2 — the expression to be evaluated if texpr is `nil`
returns — the value of the selected expression

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

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

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

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

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

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

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

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

(catch *sym expr...*) — evaluate expressions and catch throws

sym — the catch tag
expr — expressions to evaluate
returns — the value of the last expression the throw expression

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

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

V.22. Looping Constructs

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

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

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

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

expr — the body of the loop (treated like an implicit prog)

V.23. The Program Feature

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

(block *name expr...*) — named block
name — the block name (symbol)
expr — the block body
returns — the value of the last expression

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

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

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

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

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

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

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

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

V.24. Debugging and Error Handling

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

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

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

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

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

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

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

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

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

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

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

V.25. Arithmetic Functions

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

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

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

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

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

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

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

returns — the number plus one

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

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

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

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

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

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

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

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

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

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

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

y-expr — the floating point exponent
returns — x to the y power

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

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

(< *n1 n2...*) — test for less than
(<= *n1 n2...*) — test for less than or equal to
(= *n1 n2...*) — test for equal to
(/= *n1 n2...*) — test for not equal to
(>= *n1 n2...*) — test for greater than or equal to
(> *n1 n2...*) — test for greater than
n1 — the first number to compare
n2 — the second number to compare
returns — t if the results of comparing *n1* with *n2*, *n2* with *n3*, etc., are all true.

V.26. Bitwise Logical Functions

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

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

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

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

V.27. String Functions

(string *expr*) — make a string from an integer ascii value
expr — the integer
returns — a one character string

start — the starting position (zero origin)
end — the ending position + 1 (defaults to end)
returns — substring between *start* and *end*

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

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

V.28. Character Functions

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

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

(lower-case-p *chr*) — is this a lower case character?

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

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

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

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

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

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

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

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

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

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

(char< *chr1 chr2...*)
(char<= *chr1 chr2...*)
(char= *chr1 chr2...*)
(char/= *chr1 chr2...*)
(char>= *chr1 chr2...*)
(char> *chr1 chr2...*)
chr1 — the first character to compare

chr2 — the second character(s) to compare
returns — `t` if predicate is true, `nil` otherwise
Note: case is significant with these comparison functions.

(char-lessp *chr1 chr2...*)
(char-not-greaterp *chr1 chr2...*)
(char-equalp *chr1 chr2...*)
(char-not-equalp *chr1 chr2...*)
(char-not-lessp *chr1 chr2...*)
(char-greaterp *chr1 chr2...*)
chr1 — the first string to compare
chr2 — the second string(s) to compare
returns — `t` if predicate is true, `nil` otherwise

Note: case is not significant with these comparison functions.

V.29. Input/Output Functions

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

(print *expr* [*stream*]) — print an expression on a new line
expr — the expression to be printed
stream — the output stream (default is standard output)
returns — the expression

(prin1 *expr* [*stream*]) — print an expression
expr — the expression to be printed
stream — the output stream (default is standard output)
returns — the expression

(princ *expr* [*stream*]) — print an expression without quoting
expr — the expressions to be printed
stream — the output stream (default is standard output)
returns — the expression

(pprint *expr* [*stream*]) — pretty print an expression
expr — the expressions to be printed
stream — the output stream (default is standard output)
returns — the expression

(terpri [*stream*]) — terminate the current print line
stream — the output stream (default is standard output)

returns — `nil`

(flatsize *expr*) — length of printed representation using prin1
expr — the expression
returns — the length

(flatc *expr*) — length of printed representation using princ
expr — the expression
returns — the length

V.30. The Format Function

(format *stream fmt arg...*) — do formated output
stream — the output stream
fmt — the format string
arg — the format arguments
returns — output string if *stream* is `nil`, `nil` otherwise

The format string can contain characters that should be copied directly to the output and formatting directives. The formatting directives are:

~A — print next argument using princ
~S — print next argument using prin1
~% — start a new line
~~ — print a tilde character

V.31. File I/O Functions

(open *fname* &key :direction) — open a file stream
fname — the file name string or symbol
:direction — :input or :output (default is :input)
returns — a stream

(close *stream*) — close a file stream
stream — the stream
returns — `nil`

(read-char [*stream*]) — read a character from a stream
stream — the input stream (default is standard input)
returns — the character

(peek-char [*flag* [*stream*]]) — peek at the next character
flag — flag for skipping white space (default is `nil`)
stream — the input stream (default is standard input)
returns — the character (integer)

(write-char *ch* [*stream*]) — write a character to a stream
ch — the character to write
stream — the output stream (default is standard output)
returns — the character

(read-line [*stream*]) — read a line from a stream
stream — the input stream (default is standard input)
returns — the string

(read-byte [*stream*]) — read a byte from a stream
stream — the input stream (default is standard input)
returns — the byte (integer)

(write-byte *byte* [*stream*]) — write a byte to a stream
byte — the byte to write (integer)
stream — the output stream (default is standard output)
returns — the byte (integer)

V.32. String Stream Functions

These functions operate on unnamed streams. An unnamed output stream collects characters sent to it when it is used as the destination of any output function. The functions `get-output-stream-string` and string or a list of characters.

An unnamed input stream is setup with the `make-string-input-stream` function and returns each character of the string when it is used as the source of any input function.

(make-string-input-stream *str* [*start* [*end*]])
str — the string
start — the starting offset
end — the ending offset + 1
returns — an unnamed stream that reads from the string

(make-string-output-stream)
returns — an unnamed output stream

(get-output-stream-string *stream*)
stream — the output stream
returns — the output so far as a string
Note: the output stream is emptied by this function

(get-output-stream-list *stream*)
stream — the output stream
returns — the output so far as a list

Note: the output stream is emptied by this function

V.33. System Functions

(load *fname* &key :verbose :print) — load a source file
fname — the filename string or symbol
:verbose — the verbose flag (default is t)
:print — the print flag (default is `nil`)
returns — the filename

(save *fname*) — save workspace to a file
fname — the filename string or symbol
returns — `t` if workspace was written, `nil` otherwise

(restore *fname*) — restore workspace from a file
fname — the filename string or symbol
returns — `nil` on failure, otherwise never returns

(dribble [*fname*]) — create a file with a transcript of a session
fname — file name string or symbol (if missing, close current transcript)
returns — `t` if the transcript is opened, `nil` if it is closed

(gc) — force garbage collection
returns — `nil`

(expand *num*) — expand memory by adding segments
num — the number of segments to add
returns — the number of segments added

(alloc *num*) — change number of nodes to allocate in each segment
num — the number of nodes to allocate
returns — the old number of nodes to allocate

(room) — show memory allocation statistics
returns — `nil`

(type-of *expr*) — returns the type of the expression
expr — the expression to return the type of
returns — `nil` if the value is `nil` otherwise one of the symbols:
SYMBOL — for symbols
OBJECT — for objects
CONS — for conses
SUBR — for built-in functions
FSUBR — for special forms
CLOSURE — for defined functions

STRING — for strings
FIXNUM — for integers
FLONUM — for floating point numbers
CHARACTER — for characters
FILE-STREAM — for file pointers
UNNAMED-STREAM — for unnamed streams
ARRAY — for arrays

(peek *addrs*) — peek at a location in memory
addrs — the address to peek at (integer)
returns — the value at the specified address (integer)

(poke *addrs value*) — poke a value into memory
addrs — the address to poke (integer)
value — the value to poke into the address (integer)
returns — the value

(address-of *expr*) — get the address of an xlisp node
expr — the node
returns — the address of the node (integer)

(exit) — exit xlisp
returns — never returns

V.34. File I/O Functions

V.34.1. 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.

V.34.2. 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.

V.34.3. 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))
```

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.

V.34.2. 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.

V.34.3. A Slightly More Complicated File Example

This example shows how to open a file, read each Lisp expression from the file and print it. It demonstrates the use of files and the use of the optional *stream* argument to the `read` function.

```
(do* ((fp (open "test.dat" :direction :input))
      (ex (read fp) (read fp)))
     ((null ex) nil)
  (print ex))
```

References

[Touretzky 84] Touretzky, David S. *LISP: a gentle introduction to symbolic computation.* Harper & Row, New York, 1984.

Index