λ, A Sound Synthesizer

Reid Priedhorsky
Macalester College
Department of Mathematics and Computer Science
reid@reidster.net

Departmental Honors Committee:
Richard Molnar (advisor),
Dan O'Loughlin, Susan Fox

May 7, 2001

Abstract

λ is a program for creating and manipulating music. It provides a framework to facilitate communication of sound streams and control information between independent sound processing subprograms. The λ core is abstracted from any particular music system.

The goal is to make λ a professional quality sound processing system which is robust and portable. It is intended to be useful on standard midrange personal computers yet scalable up to powerful sound workstations with specialized hardware.

Contents

Chapter 1

Introduction

λ is a tool for sound processing, in the sense that it allows the user to manipulate streams of sound, whether they are generated from scratch within the program or taken from an external source. All the actual work is done by a collection of independent subprograms; one can think of λ as an operating system for programs that manipulate sound. λ provides facilities for these subprograms to send sound streams and control information between each other, so while independent they work as a coherent system.

Another way to think of λ is as a simulator. A modern musician using electronic tools has available to him or her a large set of music-making equipment: keyboards, synthesizers, sequencers, samplers, drum machines, effects boxes, mixers, and many others. He or she can connect these components into a complex music making system, then make adjustments to both individual parts and the structure of the system, or take everything apart and do something completely different. Some connections carry sound directly, and some carry control information. One can think of λ as a framework to simulate this collection of equipment in software.

λ is intended for processing sound in real time, if the available hardware is fast enough. It is not a "sample editor", a program which allows offline sample-level editing of sounds. λ's subprograms are allowed to do low level operations such as "add two frequencies" or "create a sine wave", but the λ paradigm calls for subprograms to mostly do medium to high-level operations: "mix several sound streams", "simulate a vibrating string".

Described below are λ's architecture, metaphors, and flavor. λ is written in C, but this paper is geared toward the general undergraduate computer science student. With the exception of Chapter 5 and Appendix C, it should be accessible to non-C programmers. Knowledge of digital sound and computer music is also helpful but not required. The list of references, Appendix B, is a good starting point for more about C programming and digital sound. This paper does not discuss implementation details; for those, peruse the source code, included as Appendix C.

The author can be contacted at `reid@reidster.net`. λ has a World Wide Web home page, `http://reidster.net/lambda/`. Source code is available for download here, and

electronic copies of this paper and any other documentation are posted as well. This page will remain current as future development progresses.

1.1 Goals, or Why λ

λ is intended to be a professional quality sound synthesis and processing system which is robust, portable, scalable, independent of any particular music system, inexpensive, and flexible.

Current systems are unsatisfactory. With hardware, the primary issue is cost. Computers and music/sound peripherals become very expensive as quality improves above the "hobbyist" level, so the hobbyist or novice user is stuck with "amateur" results unless they have the resources to spend large chunks of cash. With software, the problem is a bit more complex. Excellent software is available, but for a price—high-end commercial sound software is again prohibitively expensive. Low-cost sound software does exist, but it has either limited usefulness or a steep learning curve. In other words, one can get software which is either good or cheap but not both. (See Section A.1 for a discussion of some other available tools.)

However, novice or hobbyist-class users often become professional-class users, and with a bit of luck they may have more cash available for tools as well. Hence it is necessary that λ be able to scale, running effectively and efficiently on a wide range of hardware, from standard midrange PC systems to sound workstations with lots of powerful, specialized hardware. λ should permit arbitrarily complex manipulations if the underlying hardware is powerful enough, but still be useful with limited computing resources.

λ is a tool for making music. However, "music" has many and conflicting definitions; for example, not all music uses the Western 12-tone scale of notes. Because of this λ isolates all musical concepts into a small part of the system; the rest is only concerned with raw sound.

λ is intended to be useful both by itself and as part of a larger system, where it might be a master system which controls other subsystems (for example, in a concert situation λ could command the light controllers), or a slave system (for example, as a passive synthesizer translating external inputs to a sound stream which leaves the computer for further processing), or anything in between.

Chapter 2

Computer Music

This chapter is not comprehensive; it is intended to be an introduction to digital sound and synthesis thorough enough to give the reader a reasonable impression of the flavor of λ's problem domain. It attempts to do in a few pages what is properly done in a bookshelf; the reader is referred to Dodge [18] for a good comprehensive introduction to computer music. In particular, the module programmer should know more than is presented in this chapter.

2.1 Sound

As the reader may recall, sound is a pressure wave. A vibrating object compresses and rarifies the surrounding air, and these disturbances propagate outward in all directions, similar to the surface of a pond when a rock is tossed in. Figure 2.1 illustrates the most common view of a sound wave—pressure as a function of time. This particular wave, the sine wave, is the simplest and most important sound wave. It follows the curve of the function $\sin x$.

In the short term sound waves tend to be repeating, so one of the parameters of a sound wave is its *frequency*—how many times per second the cycle is repeated. The unit of frequency is the *hertz*, abbreviated Hz; a wave whose frequency is 100 Hz repeats one hundred times per second. Sound waves in the region approximately 20 Hz–20 kHz are audible to humans. The other important parameter of a wave is its *amplitude*—how much pressure variation is present. The frequency of a wave corresponds to its pitch, and its amplitude is related to its volume. In the long term one varies such parameters; for example, a person playing a musical instrument is varying the frequency of the resulting sound as he or she plays the various notes, and possibly varying the amplitude by e.g. blowing harder or softer.

One of the characteristics of waves such as sound waves is that they can be added together, just like any function. Figure 2.2 illustrates the addition of three simple sound waves to form a more complex wave. Further, and more importantly, with some technical limitations any wave can be expressed as the sum of one or more sine waves,

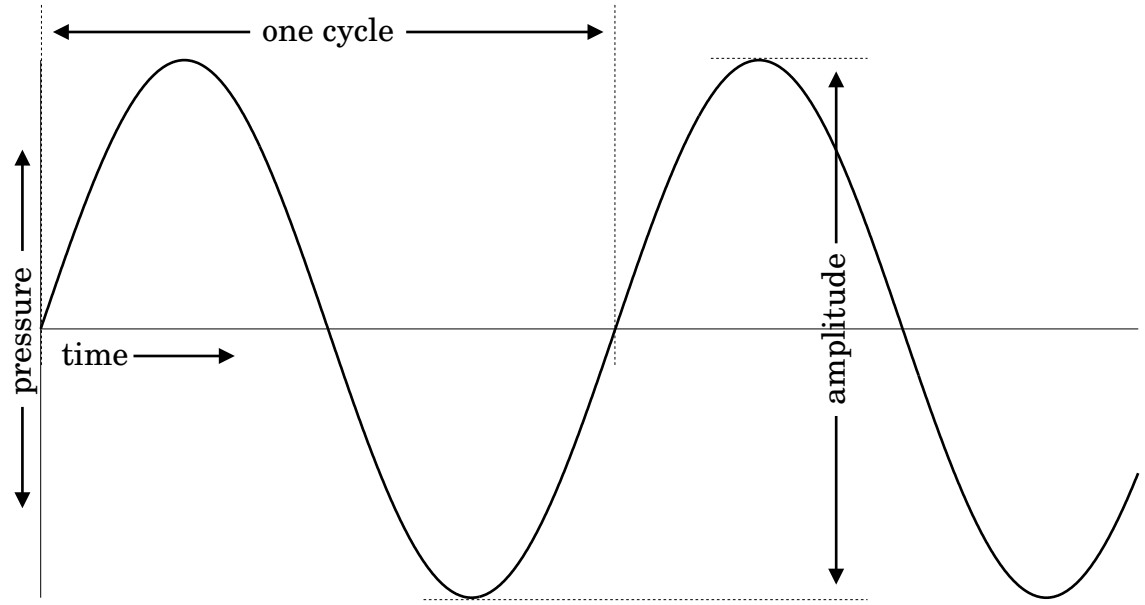

Figure 2.1: A sine wave, the simplest sound wave.

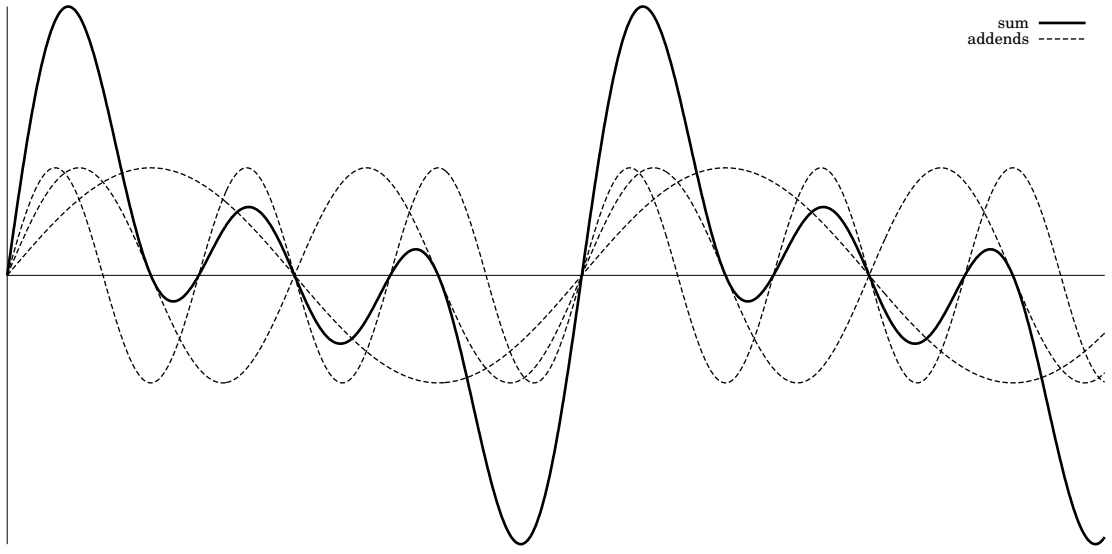

Figure 2.2: Addition of sound waves.

possibly an infinite number of them.[1]

Thus one can describe a sound wave in two ways: the *time domain*, i.e. the actual shape of the wave (pressure versus time), and the *frequency domain*, i.e. the relative amplitudes of each of the component sine waves. In the frequency domain one speaks of "frequency content" and a sound's "spectrum," which mean basically the same thing—what are the frequencies of the sine waves making up this sound, and what are their amplitudes. Most of what humans perceive as a sound's *timbre*, the "flavor" of a sound, is determined by the sound's spectrum.

Sounds and their spectra are categorized as *harmonic* if they have a perceptible pitch (frequency) or *inharmonic* if they do not. Most musical sounds are harmonic, i.e. almost all musical instruments produce sounds with harmonic spectra. The exception is percussion instruments such as drums; inharmonic spectra sound "dissonant" or "noisy". The classification can be fuzzy; one speaks of sounds that are "nearly harmonic" or "somewhat harmonic," etc. Mathematically, a spectrum is harmonic if its frequency components are at integral multiples of a given frequency f. f is known as the *fundamental* and the multiples $2f$, $3f$, $4f$, etc. are called *harmonic partials* or partials for short. The second partial has frequency $2f$, the third has frequency $3f$, etc. To hear the the maximally inharmonic spectrum known as "white noise," where all frequency components are present and of equal amplitude, turn on your TV set and tune it to a empty channel; a good approximation of white noise will emerge from the speaker. The waveforms and spectra of several important waves are shown in Figure 2.3.

2.2 Digital sound

A sound wave is continously changing; however, a computer must represent it with a finite amount of data. This is resolved by a technique called *sampling*; at regular intervals, the computer measures the value of the sound signal. Each measurement is one *sample*, and the rate at which samples are taken is known as the *sample rate*. By this method, the computer translates a continuous signal into a stream of numbers. Figure 2.4 illustrates the technique; each square indicates a sampling point. This particular waveform is represented by the stream of samples, "0.00, 0.26, 0.50, 0.71, 0.87, 0.97, 1.00, 0.97, 0.87," etc. If we assume a sampling rate of 1 kHz (1,000 samples per second), then the figure shows a 41.67 Hz sine wave, because there are 24 samples per cycle of the wave. A few common sample rates: the telephone system uses 8 kHz; compact discs use 44.1 kHz; digital audio tape (DAT) uses 48 kHz.

The resolution of sampling is limited; if a signal is changing too rapidly (i.e. if it has frequency components which are too high), important signal variations will occur between samples and be lost. Quantitatively, sampling is able to accurately represent all wave components whose frequencies are less than half the sampling rate.[2] The

[1] Some readers may notice that the author is ignoring the phase of a wave here. It turns out that phase has little effect on the human perception of sound, so this discussion purposely leaves it out.

[2] The limitation is due to technical reasons which are beyond the scope of this paper. The interested

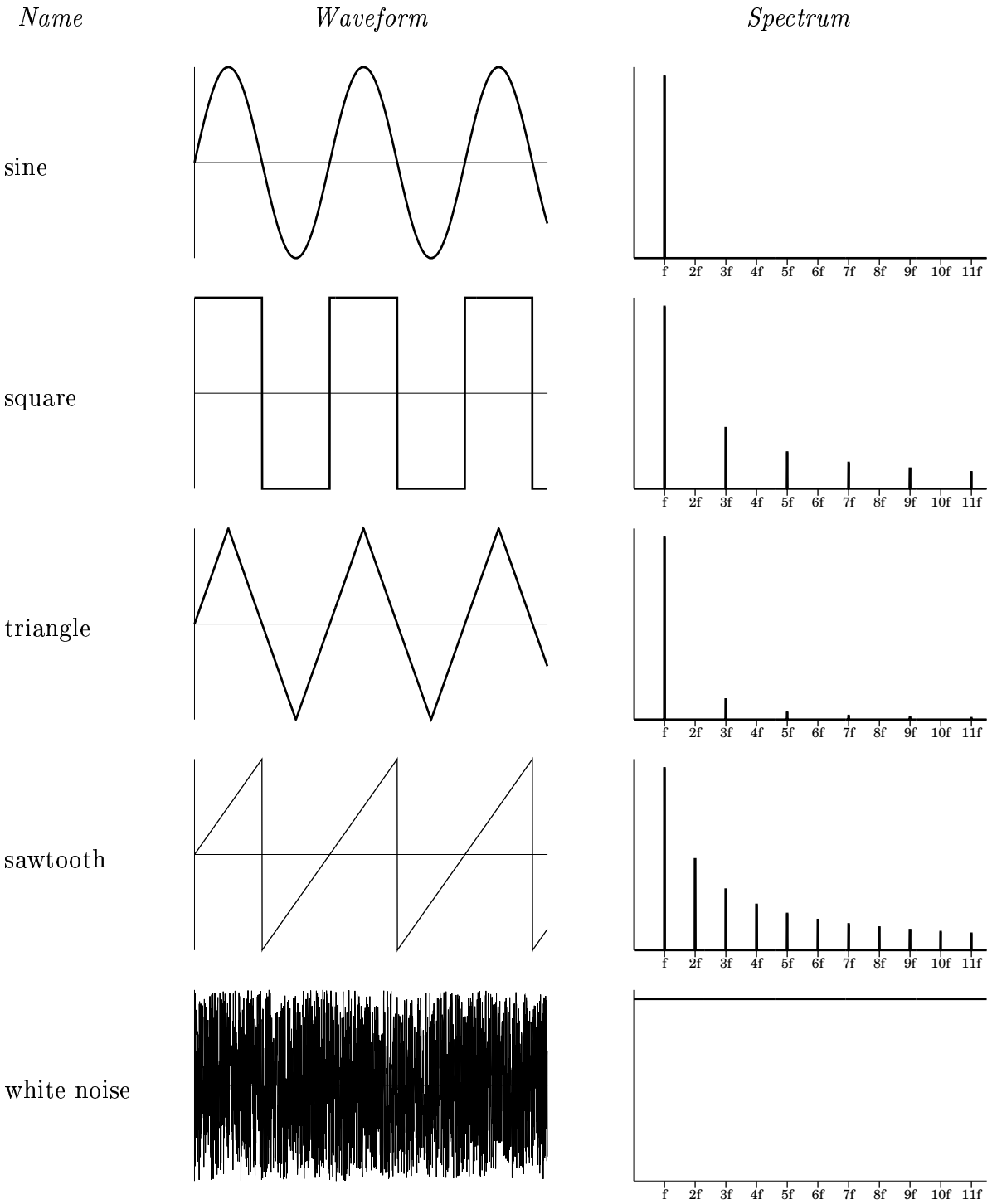

Figure 2.3: Some important waveforms.

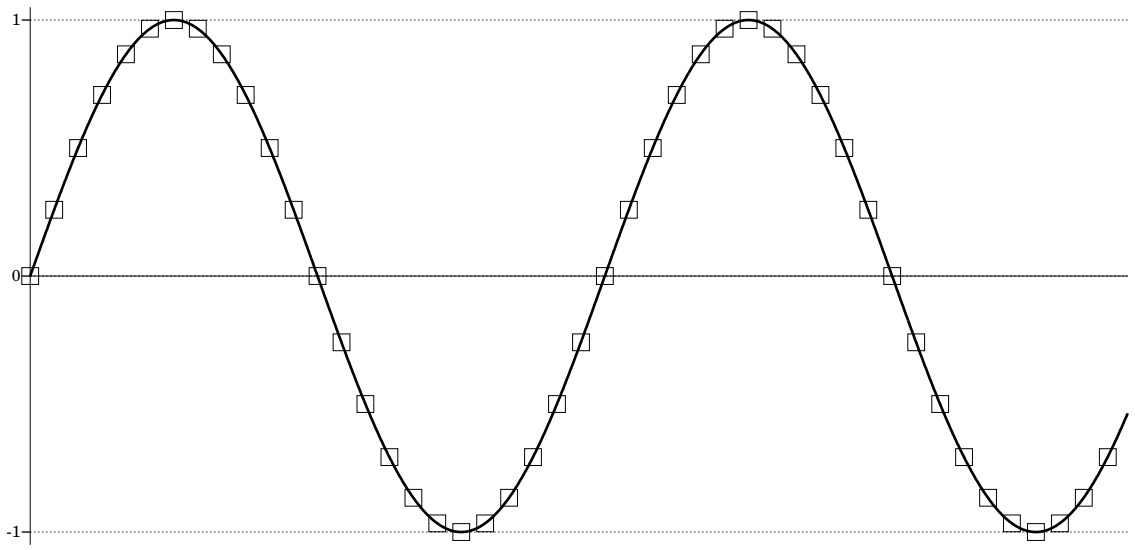

Figure 2.4: Digital sampling of sound.

consequence of this is that a digital system cannot be presented with a sound signal containing frequencies above half the sampling frequency or distortion will result.

Another consideration of digital sound is the size of samples, which determines how many discrete signal levels can be represented. If the system cannot quantify the signal level precisely enough, the signal again cannot be faithfully represented. Compact discs use 16-bit integers to represent samples, meaning that any sample can have one of 65,536 values. 16 bits is generally considered sufficient for reproduction of sound intended for a human listener, but inside a system it is useful to allow some "headroom" for calculation. Many systems use 24 or 32 bit samples internally, allowing 16 million or 4 billion discrete signal levels, respectively. However, the result of a too-small sample size is only increased background noise, not the severe distortion resulting from a too-slow sampling rate.

The tradeoff of using faster sampling rates and bigger samples is of course greater memory consumption. One minute of sound at telephone quality, 8 bit samples by 8 kHz, requires 470 kilobytes of memory; the same one minute of sound at 32 bits by 48 kHz requires 11 megabytes.

λ uses for its samples a floating point type which has a width of 64 bits on most machines. This is not out of any aspiration to ultra-precision but rather as a convenience for the programmer; this type is easy to work with, has plenty of headroom, and allows fast calculation on modern machines. Since λ does not keep lots of samples in memory, conservation of same is not a great concern, especially given the large memory capacities of current machines.

reader should consult the references listed in Appendix B.

2.3 Synthesis methods

This section presents brief discussion of a few of the more well-known sound synthesis methods. There are of course many others; the interested reader should consult the list of references, Appendix B.

Perhaps the most straightforward synthesis algorithm is *additive synthesis*. The output sound is produced by summing the appropriate component sine waves. The primary disadvantage is computational cost—recall from Section 2.1 that some waves have an infinite number of frequency components. A sine wave must be generated for each frequency component; producing some sounds at an acceptable level of accuracy may require more sine waves than the computer is capable of producing in a reasonable amount of time. Thus additive synthesis, while capable in principle of reproducing any sound, is in practice only useful for sounds with harmonic or nearly harmonic spectra.

A second major class of synthesis algorithms is known as *subtractive synthesis*. Generally, subtractive synthesis algorithms change the amplitude of frequency components of a sound but do not add components. (They can remove components by changing their amplitude to zero.) The typical use of subtractive algorithms is to take a source wave with a complex, busy spectrum and shave away components to make a simpler sound. Synthesizer tones in "80's music" are often produced by subtractive synthesis.

Distortion synthesis algorithms work by changing the shape of the waveform in some way. These algorithms often add many frequency components, so production of waves complex enough to be interesting requires only simple operands (perhaps as simple as plain sine waves). Many people are familiar with *frequency modulation*, abbreviated FM; the general idea here is that one alters the frequency of a "carrier wave" with a "modulating wave." When the modulating signal is high, the carrier wave's frequency is increased, and when the modulating signal is low, the carrier wave's frequency is decreased. (FM radio uses this technique on electromagnetic waves.) Distortion algorithms in general are quite efficient, meaning that interesting waveforms can be produced with comparatively little calculation.

Finally, the class of algorithms called *physical modeling* is benefitting much from the current rapid advance of computing speed. With physical models, the programmer models a physical system rather than the mathematics of the sound it produces. For example, to create a violin sound the programmer's algorithm would simulate the vibration of the strings and wooden parts of a violin rather than concerning itself with the shape or spectrum of the resulting wave. Physical models enable extremely realistic sounds to be produced because they produce sound in nearly the same way that real instruments do—instead of real objects vibrating, virtual objects do. However, the use of physical models extends beyond the real world; the programmer can model instruments which would be impractical or impossible to build in real life. For example, what does a Klein bottle sound like, or a 12-dimensional block of gold?

Chapter 3

Architecture

λ provides a communication framework for a collection of subprograms. Communication channels are typed, meaning that associated with each channel is type information such as "this channel carries a sample stream" or "this channel carries a frequency." Traditional hardware-based sound processing systems are typed, too, though they are not usually though of as such. For example, one cannot connect the sound signal generated by a synthesizer to an input which expects note information, because the type of the output (a sound signal) is different from the type of the input (notes). In hardware systems, correct type matching is often enforced by using different kinds of connector plugs.

In λ terminology, subprograms are called *modules*; multiple instances of each module can be created, and each instance is called a *widget*. The collection of widgets is called the *arena*. Each widget has typed input and output *ports* where data can be moved in or out of the widget; each output port can be connected to zero or more input ports of the same type (i.e., the types of connected ports must match). The configuration of a widget's ports is malleable and specified with a list of dependency rules. Each widget has a unique name.

The arena and the widgets in it have one of three states: go, pause, or stop. Stop means that nothing is happening and no modules have local storage or resources allocated (e.g. the sound card is not open and no internal buffers are allocated); it is the initial state. Go means that the system is in operation, all local storage and resources are allocated, and samples are being calculated. Pause means that local storage and resources remain allocated but no computation is taking place. The only change of state which is not allowed is stop to pause. The reader might compare λ's go, pause, and stop to the play, pause, and stop modes of a compact disc player. If the CD player is in stop mode, the device is inactive, and the user can change discs, etc. In play mode, the disc spins and sound is played, while in pause mode the disc spins and the device remembers its place in the music, but no sound is played.

λ is a single-threaded program. The program maintains a virtual time which corresponds to the duration of the samples calculated. Virtual time advances in discrete

chunks called *ticks*; all operations occur on blocks of samples one tick long. In the go state λ does what operating systems people call cooperative multitasking: each widget's tick function is called in turn. If the hardware is sufficiently powerful, calculating one tick will take less time than the duration of the sample block; in this case the system can run in real time, meaning that it can calculate samples faster than they are played. The system supports a "real-time mode" where if a tick's calculation is completed in a shorter wall-clock time than the duration of one block of samples, the system will sleep for the difference. For example, if the sample rate is 44.1 kHz and ticks are 256 samples long, the nominal duration of a tick is 5.8 ms. If λ completes the tick's calculation in 3.4 ms, then it will sleep for 2.4 ms—the program outputs samples exactly as fast as they are played. If the system is not running in real-time mode, calculation will proceed as fast as possible.

3.1 Types

The data communicated between widgets is typed; hence, λ implements a fairly elaborate type system. It is described here in general terms, and its exact programmer's interface appears in Section 5.2.

3.1.1 Plain types

λ defines several types to assist in module programming. Some of these are general types to augment the standard C types: a boolean type, a byte type, and several integers of specific sizes. The others are higher level types for sound related numbers. Each of these are floating point types; in particular it is important to know that λ uses floating point samples. They are the following:

- `sample`. A single sound sample. A sound signal ranging between –1.0 and +1.0 is considered to be full amplitude. Sample values outside this range risk being clipped, but such behavior is documented.[1]
- `frequency`. A frequency in hertz. Since the λ core is abstracted from any particular music system, it has no concept of notes—all frequencies are represented numerically.
- `duration`. A duration or time interval, in seconds. Again, the λ core has no concept of tempo.
- `level`. A value used as scaling factor or intensity selector. For example, a mixer's volume controls (for both input and output streams) would have type `level`.

[1] Currently, there is no clipping anywhere.

3.1.2 Complex types

Since C does not provide a useful built-in type descriptor type, λ supplies its own, creating a `ltyp` data type which describes other types.

An `ltyp` object can describe three "meta-types" of objects: scalars, arrays, and *relarrays*. Scalars and arrays are as usually considered. A scalar needs one parameter to describe it, its type (e.g. `int`), while an array needs two, its type and the number of elements (e.g. `int[10]`). A relarray (short for "relatively sized array") works like a normal array, except that the number of elements is implicitly multiplied by the number of samples per tick. For example, we might have an algorithm which requires four `int`s per sample; in this case, we would create a relarray of size 4. If there are 64 samples per tick, then the relarray would have 256 elements; if 23 samples per tick, then it would have 92 elements. λ writes relarrays like so: `int[4r]`. An `ltyp` object to describe such a relarray would specify "relarray, `int`, 4".

In λ, sample streams are relarrays of samples of size one—widgets communicate sample streams in blocks whose length is that of one tick.

3.2 Ports

Widgets move data in and out through their ports. As noted earlier, an input port and an output port which have the same type (each port has associated with it an `ltyp` type descriptor) can be connected together, forming a channel along which data can be moved. An output port can be connected to more than one input port, but the reverse is not allowed. Each port has a name. Input ports can also be set to a constant value.

Widgets do not have a fixed number of ports. When writing the module the programmer specifies a list of all the potential ports and dependencies among them. A port depending on another port means that it is not legal to connect or set the first port unless the second is connected or set also. For example, a mixer module might have several input sound streams, and associated with each sound stream is a volume control. Each volume control would depend on its associated sound stream, because it is nonsensical to adjust the volume of a sound stream which does not exist. Further, ports can be tagged *strict*, in which case they must be connected or set if their dependencies are satisfied. The volume controls in the preceding example would be strict because if a sound stream is present, its volume must have a definite value.

The connection between ports is not a FIFO; the system buffers only a single object within a connection. The intended correspondence is to signal cables in hardware systems—only one electrical value can be asserted at once. Each time an output port is written to, the previous object in the connection is overwritten, and objects in connections persist until they are overwritten. Widgets can read or write only whole objects (e.g. a widget writing an array must write it all at once).

Data is available at the other end of the connection as soon as it is written to the

output port; this means that signals propagate through the arena in zero virtual time if each widget's tick function is called by the system after all widgets writing to connections to its input ports and before all widgets reading from connections to its output ports. A connection which does not follow this will have an *implicit delay* of the duration of one tick's worth of samples. Widgets are called in reverse order of when they were created.[2]

λ is responsible for ensuring the satisfaction of widgets' port dependencies. If all dependencies are known to be satisfied, then the arena is *clean*; otherwise, the arena is *dirty*. A dirty arena becomes clean when the user explicitly asks λ to check dependencies and all dependencies turn out to be satisfied; a clean arena becomes dirty when the configuration of its widgets is modified in any way. The arena is not allowed to enter the go state unless it is clean. When the arena enters the go state, each widget receives a list of all of its potential ports and their connection status. Widgets need not be concerned about being given a set of ports with unsatisfied dependencies, but they might verify dependencies anyway using assertions, as a sanity check.

3.3 Modules

Modules are dynamically loaded executable binary object files exporting specified functions and variables. Modules provide a function to be called on entry of each state and a function called during calculation which advances the widget's local virtual time. All functions must be reentrant because a single module might be used for several widgets.

3.3.1 What a module must implement

A module must identify itself. It defines a variable containing a certain constant (a "magic number"), so λ can identify it as a module conforming to the proper interface. It also defines strings containing its name, version, and a short description. Also, it defines a list of all of its potential ports, their types, and their dependencies.

A module provides several functions which are called by λ at certain times. Each of these functions accepts as one of its parameters a context; this allows for reentrancy. It is module-defined and contains whatever information should be kept during calculation for that kind of widget. The module implements one function for each state, called on state entry. The go entry function initializes the context and performs all other necessary initializations and resource allocations (e.g. opening the sound card, allocating buffers). It is only called on entry of the go state from stop—no function is called on entry of go from pause. The pause entry function should do nothing unless some external constraint requires it. For example, some sound card libraries require notification if no more samples are forthcoming for a while but the sound card should not be closed. The

[2]This is a technical limitation of the current implementation. Future versions will allow a user-specified calling order.

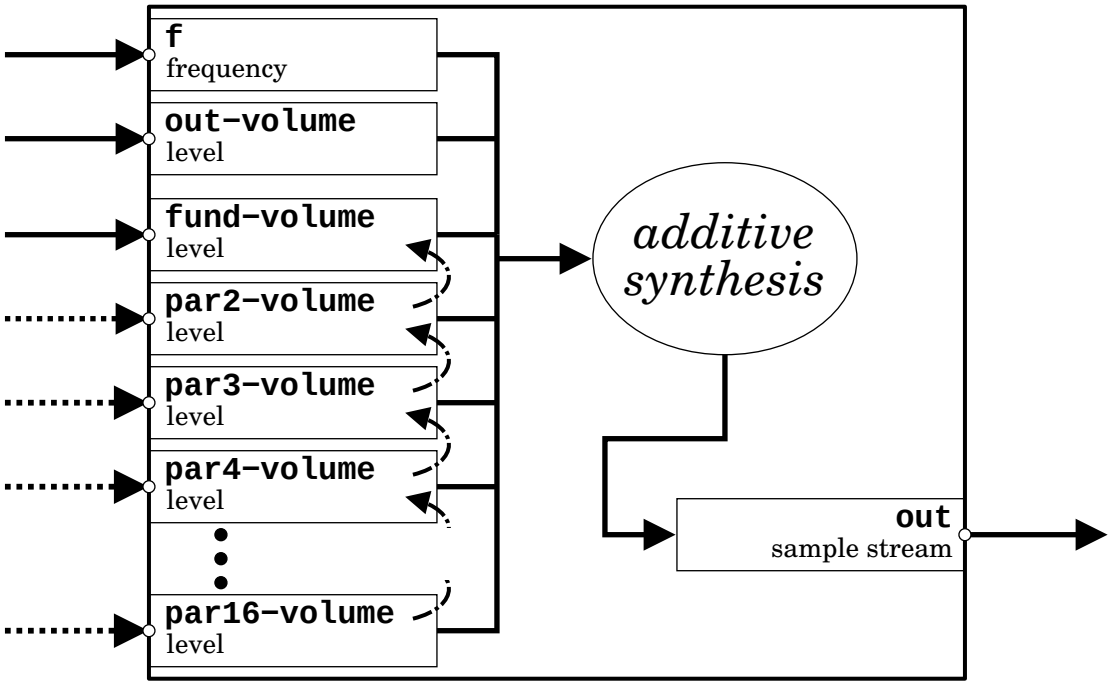

Figure 3.1: `synth-additive`: Simple additive synthesizer with 16 harmonics.

stop entry function deallocates all resources (e.g. closes the sound card and deallocates buffers).

The last function, the "tick function," is called once per tick. It must do whatever is appropriate for that module to advance its local virtual time by one tick's worth of samples—reading and writing from ports, generating samples, dealing with external devices, etc.

3.3.2 Example module

The following is a detailed description of a synthesizer module named `synth-additive` which implements simple additive synthesis. No source code is presented in this section, only algorithms; the interested reader should consult the source file `synth-additive.c` in Appendix C. Figure 3.1 is a block diagram of the module. The principal inputs are a fundamental frequency and the strengths of up to 15 harmonic partials. One possible use of this module could be to gradually (by dynamically changing the partial strengths) sweep the timbre of the output sound stream from, for example, trumpet-like to harpsichord-like.

In the figure, strict ports have a solid arrow while nonstrict ports have a dotted arrow. `synth-additive` has 19 ports, 18 input and one output:

- `out`. The result of the module's calculations is this outgoing sample stream. It has no dependencies and is strict, i.e. this port must always be connected.

- `out-volume`. This port controls the volume of the output sample stream (port `out`); it is of type `level` and also has no dependencies and is strict.

- `f`. This port sets the fundamental frequency. Its type is of course `frequency`, and it again has no dependencies and is strict.

- `fund-volume` and `par2-volume` through `par16-volume`. These `level` ports control the relative volumes of the fundamental and each partial. `fund-volume` is strict and has no dependencies; the rest are nonstrict and each depends on the previous one, i.e. `par2-volume` depends on `fund-volume`, `par3-volume` on `par2-volume`, etc. For example, if `par8-volume` is to be connected, `par2-volume` through `par7-volume` must be connected or set as well.[3]

`synth-additive`'s go function does some memory allocation, the details of which are not important here. The stop function does corresponding deallocations, and the pause function is empty. The tick function's algorithm is given below; recall that its job is to advance the widget's virtual time by the proper number of samples.[4] Two variables are defined *a priori*: *samples_per_tick*, which is (obviously) the number of samples per tick, and *partials*, the number of partial volume ports which are connected or set (`fund-volume` and `par`*i*`-volume` are such ports). (Recall that because of the module's dependencies, `fund-volume` is always active, and *harmonics* will be the number of the highest numbered active `par`*i*`-volume` port.)

1. [Read data waiting on input ports to local variables.]

   (a) Store the frequency waiting on port `f` in f.

   (b) Store the level waiting on port `out-volume` in *out_vol*.

   (c) Store the level waiting on port `fund-volume` in par_vols_1

   (d) For i from 2 to *harmonics*, store the level waiting on port `par`*i*`-volume` in par_vols_i.

2. [Initialize an array of samples to be used as an accumulator.] Set each element in *buf_a*, an array of samples of length *samples_per_tick*, to zero.

[3] Another, more flexible way to structure the dependencies would be to eliminate them, with each unconnected port assumed to be zero. The present arrangment was chosen to illustrate the concepts of port dependencies and strictness.

[4] There exist more efficient algorithms with equivalent results, but this one is presented for simplicity.

3. [Generate the waveform for each harmonic and add it to the accumulator.] For i from 1 to *harmonics*:

   (a) Generate *samples_per_tick* samples of a sine wave at frequency $f \times i$ and store them in array *buf*.

   (b) For each element in *buf*, multiply it by par_vols_i and increase the corresponding element of *buf_a* by the result.

4. [Scale the generated samples according to *out_vol*.] Set each element of *buf_a* to itself multiplied by *out_vol*.

5. Write *buf_a* to output port `out`.

3.4 Example arena

This section discusses an example configuration of λ. The tools used in this example are a MIDI[5] keyboard, a foot pedal, a disk file containing a sequence of notes, a stereo sound card with speakers, and a computer running λ. The musician wants to play a simulated tuba via the keyboard and pipe the tuba's output through a distortion effect whose level of distortion is controlled by a foot pedal; the resulting distorted sound stream is to be played through the right speaker. Playing through the left speaker is a bass guitar; it is to follow a preprogrammed sequence of notes taken from the disk file.

This arrangement is shown in Figure 3.2. The arena is populated with seven widgets. In this example, each widget is an instance of a different module, but that is not necessarily the case.

One can informally classify the widgets in this arena into three classes. The first class is the "device drivers;" they communicate with with parts of the system external to λ. `io-midi` translates the input from the keyboard into a frequency; hence it has a single port, an output port called `out-f` of type freqency. For example, if the musician pressed middle C on the keyboard, the frequency 261.6 Hz would appear on `out-f`. `io-pedal` translates the input from the foot pedal into a form useful to λ; it has a single output port, `depress`, of type `level`. When the pedal is resting, 0.0 appears on `depress`; when the pedal is fully depressed, 1.0 appears; in some intermediate level of depression, some intermediate value appears. `sequencer` reads the list of notes in the disk file, and the proper frequencies appear in order on its output port `out-f`. (Note that different widgets can have ports of the same name.) Finally, the widget accepts two sound streams on its input ports `left` and `right`, converts them internally to the proper format for the sound card, and arranges for them to be played on the proper speakers.

[5]All the reader unfamiliar with MIDI needs to know here is that MIDI is a communications protocol used in computer music; in this case a music keyboard is using it to send information to λ.

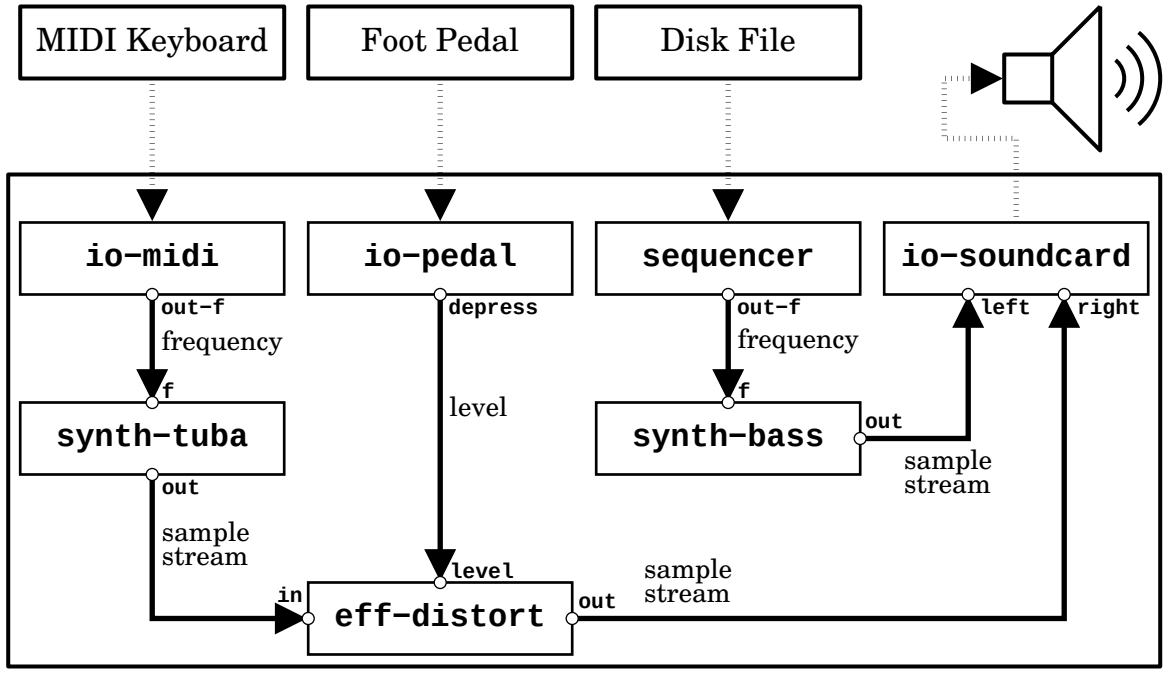

Figure 3.2: Example λ setup with seven widgets in the arena.

Another class of widgets has only one member in this example. `eff-distort` is an effect, changing one sample stream into another according to some other inputs. This widgets accepts an incoming sample stream on its input port `in`; the resulting sample stream emerges from the output port `out`. The sample stream is distorted according to the value at the other input port, `level`; if zero is present the signal is passed unchanged, or if one is present the signal is mangled to unrecognizability, or some intermediate amount.

The last class, synthesizers, produce a sound stream according to various parameters. In this case, `synth-tuba` and `synth-bass` synthesize a tuba and a bass tone, respectively, at the frequencies present on their input ports `f`. For example, if `synth-tuba` reads 261.6 from its `f` port, it will synthesize a tuba playing the note "middle C." The resulting sound stream emerges from `out` on both widgets.

3.5 Thoughts

The next question on the reader's mind might be, "What is it good for? How do I go about using this system?" The person who is interested in using what is already available in the system, i.e. the λ core with already-programmed modules, might follow these steps:

1. Review the available modules and decide which ones to use.
2. Build an arena containing widgets and connections between them which are "interesting" in some way.
3. Make some music.
4. Repeat Steps 2 and 3 as desired.

A programmer who is interested in extending λ might do the following to create a new module and make it accessible to the system:

1. Choose an "interesting" sound synthesis or manipulation algorithm.
2. Adapt the algorithm to proceed in chunks, so λ can advance it one tick at a time.
3. Write a program conforming to the λ programmer's interface which implements the adapted algorithm.
4. Compile the program into a form accessible to λ, creating a module.
5. Put the module where modules are stored (documentation included with the λ distribution package explains the directory structure).

On that note,[6] the main conceptual discussion of the system is concluded. The following two chapters cover the implementation; Chapter 4 discusses the user's interface and Chapter 5 gives the programmer's interface.

[6] Pun intended.

Chapter 4

Interface

λ is a console-mode application; the user provides instructions to the system via invocation arguments and an obtuse, hard-to-use command-line interface. It is designed to permit an external add-on graphical interface, which will be included in future versions of the system.

4.1 Invocation

The user starts lambda by typing `lcore` followed by zero or more command-line options at the shell prompt. λ recognizes both traditional short options and equivalent GNU-style long options; the available options are listed in Table 4.1. Some options take an integer argument *x*.

Options		*Meaning*
`-d`	`--debug`	Print debugging output.
`-l`	`--no-speed-limit`	Disable real-time mode; calculate as fast as possible.
`-s`*x*	`--seed=`*x*	Seed the random number generator with the value *x*.
`-r`*x*	`--sample-rate=`*x*	Set the sample rate to *x* samples per second.
`-t`*x*	`--samples-per-tick=`*x*	Set the tick length to *x* samples.
`-i`*x*	`--input-microseconds=`*x*	Check for console input about every *x* microseconds.

Table 4.1: Command line options.

For example, one might say

```
lcore -d -s42 --samples-per-tick=64
```

causing λ to enable debugging output, seed the random number generator with the value 42, and operate with a tick length of 64 samples.

4.2 The λ command line

λ asks for input by printing `lambda:`*`state`*`>`, where *`state`* is one of `go`, `pause`, or `stop`. At this point the user can type a command and press enter. In the pause and stop states, the command is executed immediately; in the go state there is a short (but hopefully imperceptible) delay because the system only stops calculation occasionally to check for input. (The frequency of checking for input can be adjusted with a command-line option; see the previous section.) Many commands require the specification of a particular port on a particular widget; λ uses a slash character to separate the two, e.g. `foo-widget/bar-port`. If a command cannot be completed for some reason a diagnostic message is printed. The available commands are the following.

`load` *`module widget`*

> Load the module named *`module`* (if it is not already loaded) and create a widget named *`widget`* from it in the arena. This command is only allowed in the stop state.

`delete` *`widget`*

> Remove the widget named *`widget`* from the arena, breaking all connections involving it. This command is only allowed in the stop state.

`connect` *`widget-1/port-A widget-2/port-B`*

> Connect port *`port-A`* of widget *`widget-1`* to *`port-B`* of widget *`widget-2`*. Of course, *`port-A`* must be an output port, *`port-B`* must be an input port, and their types must match. This command is only allowed in the stop state.

`disconnect` *`widget-1/port-A widget-2/port-B`*

> Remove the connection from port *`port-A`* of widget *`widget-1`* to *`port-B`* of widget *`widget-2`*. Such a connection must exist. This command is only allowed in the stop state.

`set` *`widget/port value`*

> Set the input port *`port`* on widget *`widget`* to *`value`*. This command is allowed in all states. For example,
>
> ```
> set synth-additive/f 150.0
> ```

`unset` *`widget/port`*

> Remove any set value associated with port *`port`* of widget *`widget`*. This command is only allowed in the stop state.

`print`

Print the contents of the arena and the configuration of all the widgets in it. The next section describes how to interpret the output. This command is allowed in all states, but in the go state it might cause a brief interruption in sound output if the system is running in real-time mode.

`verify`

Check that all widgets' port dependencies are satisified. If so, mark the arena clean. This command is only allowed in the stop state.

`clear`

Remove all widgets from the arena. This command is only allowed in the stop state.

`halt` *np*

This command causes the system to solve the halting problem. The argument *np* is optional; if specified, the system also proves that $NP = P$. This command is allowed in all states.

`read` *filename*

Read the contents of the file *filename* and interpret them as if they had been typed at the command prompt. This command is only allowed in the stop state.

`go`

Enter the go state. This command is not allowed if the system is already in the go state or if the arena is dirty.

`pause`

Enter the pause state and print some information about resource utilization. This command is only allowed in the go state.

`stop`

Enter the stop state and print some information about resource utilization. This command is not allowed in the stop state.

`quit`

Perhaps the most important command, `quit` causes λ to switch to the stop state if it is not already stopped, clean up, and exit.

4.3 Example interaction

In this example, the user starts λ with debugging output enabled and ticks of 64 samples length. He loads an FM synthesizer and a sound card device driver module, connects the synthesizer to the sound card widget, and sets some parameters on the synthesizer. He verifies the arena (to make it clean). Then he outputs 11 seconds of sound, stops (note the statistics given), and quits.

```
$ ./lcore -d -t64
lambda version 1.0d1 starting, please mind the rotor wash.
debugging output: enabled
assertions:       enabled

random seed: 988070259
sample rate: 44100.000000
samples per tick: 64
input microseconds: 100000
speed limiting: yes
lambda: stop> load synth-fm fm
lambda: stop> load io-oss sc
lambda: stop> connect fm/out sc/play-right
lambda: stop> set fm/f-car 4
lambda: stop> set fm/f-mod 30
lambda: stop> set fm/index 40
lambda: stop> verify
lambda: stop> go
io-oss: configuring /dev/dsp for 16 bits, 1 channels, 44100 Hz
lambda:   go> stop
Since entry of go state from stop:
  Samples calculated:        484544
Since entry of go state:
  Samples calculated:        484544
  Seconds spent calculating: 0.914
  Total seconds elapsed:     11.007
  Calculation load:          0.08
lambda: stop> quit
Goodbye.
```

Chapter 5

Module API

This chapter describes the λ modules application programming interface (API). The first section describes what the module programmer must write; the rest describes the λ API functions available to modules. This chapter is intended for C programmers, but the knowledge of C required for basic understanding of it is not large. All the reader needs to know to get a general idea of what is going on is that C is a weakly typed procedural language, and the "null" type used to indicate no return value or no arguments is `void`. C's syntax should be familiar to C++ and Java programmers because C is the ancestor of these languages. The canonical C book is Kernighan and Ritchie [26], and C tutorials of varying quality abound, both online and printed.

Module source files must include the λ header `lambda.h`, which contains definitions of all the types and functions described in this chapter. The format of a ready-to-use module is a single compiled object file which can be dynamically loaded by the main λ program; most compilers need to be told specifically to produce this kind of object code. With GNU `gcc`, modules must be compiled with the flags `-shared` and `-fPIC` (correspondingly, the main program is compiled with the flag `-rdynamic`). The organization of a module's source code is up to the programmer; the only requirement imposed by λ is that the final product be contained in a single object file which exports the variables and functions described in the next section. It is not required that modules be written in C, but the module programmer must ensure that the objects in this section follow C conventions so they are accessible to the λ core.[1]

Functions in this chapter are documented like so:

`foo f (bar a)`

> This is a function, `f`, taking one parameter, *a*, of type `bar`. It returns an object of type `foo`.

[1] For example, C++ programmers would declare them as `extern "C"`.

5.1 What a module must implement

This chapter describes functions and variables which must be exported by a module; that is, when λ loads a module it expects these objects to be accessible. Each module has a separate namespace, so the programmer need not be concerned about clashes with other modules' object names. In addition to those given here, λ reserves all other names beginning with `lm_` for future use, and certain names beginning with _ (an underscore) are used by the operating system's dynamic loader. The rest are free for use by the module programmer. Source code for several example modules is included in Appendix C.

The following four variables must be exported by a module as identification.

`unsigned magic`

A module proves that it is a λ module conforming to the proper API version by setting this variable to the macro `MOD_MAGIC_1_0_0`.

`char name[]`

The module's (short) name.

`char version[]`

The module's (short) version string.

`char description[]`

A one-line description of the module.

A module must list all of its potential ports by exporting an array of port declarations:

`port_decl port_decls[]`

A list of port declarations terminated with the macro `END_PORT_DECLARATIONS`. The `port_decl` structure is defined as:

```
typedef struct {
    char * name;
    enum port_direction direction;
    meta_ltyp meta_type;
    raw_ltyp raw_type;
    unsigned n_elements;
    enum port_strictness strict;
    char * depends[MAX_PORT_DEPS + 1];
} port_decl;
```

The `name` field holds the name of the port, and the direction, either `IN` or `OUT`, is in `direction`. The contents of `meta_type`, `raw_type`, and `n_elements` are passed directly to `ltyp_new()`; see the documentation for that function in the next section. The field `strict` defines the port's strictness, either `STRICT` or `NONSTRICT`. Finally, `depends` contains a `NULL`-terminated[2] list of names of ports the port depends on; up to *`MAX_PORT_DEPS`* (currently 15) dependencies are allowed.

For example, the additive synthesizer module described in Section 3.3.2 defines its `port_decls` as follows:

```
port_decl port_decls[] = {
    { "out",          OUT, ARRAY_REL, LTYP_SAMPLE, 1, STRICT, {} },
    { "out-volume",   IN,  SCALAR,    LTYP_LEVEL,  0, STRICT, {} },
    { "f",            IN,  SCALAR,    LTYP_FREQ,   0, STRICT, {} },
    { "fund-volume",  IN,  SCALAR,    LTYP_LEVEL,  0, STRICT, {} },
    { "par2-volume",  IN,  SCALAR,    LTYP_LEVEL,  0, NONSTRICT,
      { "fund-volume" } },
    { "par3-volume",  IN,  SCALAR,    LTYP_LEVEL,  0, NONSTRICT,
      { "par2-volume" } },
    { "par4-volume",  IN,  SCALAR,    LTYP_LEVEL,  0, NONSTRICT,
      { "par3-volume" } },
    /* ... etc. ... */
    { "par16-volume", IN,  SCALAR,    LTYP_LEVEL,  0, NONSTRICT,
      { "par15-volume" } },
    END_PORT_DECLARATIONS
};
```

Finally, the module must provide four call-back functions; λ calls them at entry of states and during calculation. Their return value is according to the error handling mechanism in Section 5.5.

`lerr lm_go(void ** `*`context`*`, unsigned `*`n_ports`*`, port * `*`ports`*`[])`

Called at entry of the go state. Provided is a list of the module's ports, both connected/set and unconnected, in the array *`ports`*; it has *`n_ports`* members. This function's primary purpose is to do any allocation or initialization required for the module and store a pointer to whatever context object is required in `*`*`context`*.

`lerr lm_pause(void * `*`context`*`)`

Called at entry of the pause state. *`context`* is a pointer to the context object initialized by `lm_go()`. For most modules this function is empty, returning `OK` immediately.

[2]The compiler will supply the terminator automatically if the dependency list is written explicitly in the source file (technically, as an "initializer"). See Kernighan and Ritchie [26], page 219.

`lerr lm_stop(void * context)`

Called at entry of the stop state. *context* is a pointer to the context object initialized by `lm_go()`. This function's job is to deallocate and release all resources allocated by `lm_go()`. λ does no cleanup on behalf of modules; after this function returns λ no longer keeps a copy of *context*.

`lerr lm_tick(void * context)`

Called during each tick; widgets are guaranteed to be called in the same order unless the arena is changed. *context* is a pointer to the context object initialized by `lm_go()`. This function is really the core of the module; its job is to advance the widget's virtual time by the duration of one tick, doing all necessary calculation and port input/output.

5.2 Types

This section gives the specific API of the λ type scheme. For a semantic discussion of the type scheme, see Section 3.1. A word of caution: the compiler can do little to ensure the correct use of the λ type system. If λ expects a pointer to point to an object of a certain type because a type descriptor said so, and the object is not the expected type, unexpected and bad things will happen, and there is nothing the compiler can do to help. The effects might be subtle. Watch out!

5.2.1 C types

The type descriptor type is `ltyp`. It is an abstract data type with no public members. Also, λ defines the following plain types.

- `bool`: A boolean type; permissible values are `0` (false), `1` (true), and a result of the standard boolean C operators (`&&`, etc.). The compiler might permit it, but storage of other values in a `bool` variable is undefined.
- `byte`: A single byte which can be treated as an 8-bit unsigned integer.
- `int16`, `uint16`, `int32`, `uint32`, `int64`, and `uint64`: Signed and unsigned integers of specific widths.
- `sample`, `freq`, `dur`, `level`: Floating point types representing a sample, frequency, duration, and level.

5.2.2 Type operations

The following are the type manipulation functions available to modules.

`ltyp * ltyp_new (meta_ltyp` *meta_t*`, raw_ltyp` *raw_t*`, unsigned` *n_elems*`)`

Allocate and build a new `ltyp` type descriptor object and return a pointer to it. If *meta_t* is `SCALAR`, *n_elems* is ignored; if *meta_t* is `ARRAY` or `ARRAY_REL`, *n_elems* defines the size of the array or relarray. (Recall the definition of relarray in Section 3.1.2.) No other values of *meta_t* are valid. *raw_t* gives the "base type" for the new descripton and is one of these constants:

```
LYTP_BOOL  LTYP_INT16   LTYP_FLOAT   LTYP_SAMPLE
LTYP_BYTE  LTYP_UINT16  LTYP_DOUBLE  LTYP_FREQ
LTYP_INT   LTYP_INT32                LTYP_DUR
LTYP_LONG  LTYP_UINT32               LTYP_LEVEL
           LTYP_INT64
           LTYP_UINT64
```

Arrays or relarrays of size zero are undefined. This function cannot fail.

For example, after

```
ltyp * t;
t = ltyp_new(ARRAY, LTYP_INT, 10);
```

`t` describes an array of ten integers.

`void ltyp_free (ltyp *` *t*`)`

Deallocate *t*.

`size_t l_sizeof (ltyp *` *t*`)`

Return the number of bytes required to store an object described by *t*.

`bool ltyp_eq (ltyp *` *t1*`, ltyp *` *t2*`)`

Return true if *t1* and *t2* are equivalent descriptions, false otherwise. Arrays and relarrays are never equivalent, even if they have the same absolute number of elements.

`char * ltyp_str (ltyp *` *t*`)`

Return a string representation of the description in *t*. For example, if *t* describes a scalar `int`, this function returns "`int`"; if *t* describes a relatively size array of size 2, this function returns "`sample[2r]`".

`char * ltyp_tostr (ltyp * t, void * value)`

Return a string representation of the object pointed to by *value*, interpreting it according to *t*. Currently, only scalars are supported.

`lerr ltyp_fromstr (void * buf, ltyp * t, char * str)`

Convert the string *str* to an appropriate value according to *t* and store the result in caller-allocated *buf*. Currently, only scalars are supported.

5.3 Ports

Ports have type `port`, a structure with the following public members, which should be treated as read-only:

- `char * name`: The port's name.
- `enum port_direction direction`: The port's direction, either `IN` or `OUT`.
- `ltyp * type`: The port's type.

The available operations on ports are the following. Recall that connections between ports are not FIFOs; refer to Section 3.2 for the semantics of connections.

`void l_send (port * p, void * buf)`

Send the contents of *buf* on port *p*. *buf* must hold one object whose type is the same as *p*'s.

`void l_recv (port * p, void * buf)`

Receive an object from *p* and put it in *buf*, which must be large enough to hold it (the object's type is that of *p*).

`port * port_addrof (char * name, unsigned n_ports, port * ports[])`

Search for the port named *name* in the array of ports *ports* whose length is *n_ports*. If found, return a pointer to it; otherwise, return `NULL`.

`bool port_ready (port * p)`

Return true if *p* is ready for data transfer (i.e. connected or set), false otherwise.

`void port_VERIFY(port * p)`

If `NDEBUG` is defined, this macro expands to nothing; otherwise, it verifies the internal consistency of port *p* and aborts the program (using `l_ASSERT()`) if a problem is found. In particular, its first check is if *p* `== NULL`. Modules should use it like an assertion to ensure the integrity of their port objects.

5.4 Timing information

The following functions solicit information about system timing information and the current virtual time. Functions querying the current virtual time will return the virtual time of the completion of the most recently-completed tick.

`freq samples_per_second (void)`

Return the sample rate, i.e. the number of samples per second.

`dur seconds_per_sample (void)`

The inverse of the above; return the duration of one sample in seconds.

`unsigned samples_per_tick (void)`

Return the duration of one tick in samples.

`unsigned samples_calc (void)`

Return the number of samples since the last entry of the go state from stop (i.e., pausing and reentering go does not affect the count, but stopping and reentering go will reset it), up to the end of the previous tick.

`dur seconds_calc (void)`

Return the duration in seconds of the sample count returned by `samples_calc()`.

5.5 Error handling

λ has a fairly elaborate system for handling errors. λ keeps a global error string, and functions using the error handling system return an error flag, either `OK` if they were successful or `ERR` if an error occurred. In this case they can either set the error string to a new value or prepend a string to the existing one (if the error originated inside a called function which also uses this error handling mechanism). Successful functions must not change the error string. The public functions are as follows.

`char * err_get (void)`

Return a pointer to the error string. It is safe to pass the returned pointer (or a pointer to any character in the error string) to other `err_` functions, but the caller should not change the error string except by these functions.

`char * err_copy (void)`

Return a pointer to a newly-allocated copy of the error string. The caller is responsible for deallocating the copy when necessary.

`void err_set (char * fmt, ...)`

Expand *fmt* and the following arguments according to `printf()` rules; set the error string to the result.

`void err_prepend (char * fmt, ...)`

Expand *fmt* and the following arguments according to `printf()` rules; prepend the result to the error string. Note that no characters are inserted between the two strings; if delimeters are desired they must be specified explicitly.

`void err_ignore_on (void)`

Ignore any attempts to change the error string until `err_ignore_off()` is called. Ignore-unignore pairs cannot be nested.

`void err_ignore_off (void)`

Stop ignoring attempts to change the error string.

5.6 Terminal output

λ provides terminal output functions which are workalikes for `printf()` and friends. They provide hints about where output should go without sending directly to the output streams, to allow for things like logging, etc.

Modules should use these functions sparingly in the go state, because terminal I/O can be quite slow. Also, modules should not print error messages directly; they should use the facilities in the previous section.

`void pr (char * fmt, ...)`

Just like `printf()` except that nothing is returned. *fmt* is a format string according to `printf()` rules and appropriate arguments follow; the result is printed on standard output.

`void debug_pr (char * fmt, ...)`

Like `pr()`, but text is only printed if debugging output has been activated.

5.7 Oscillators

λ provides an oscillator API. Several common waveforms are provided, and it is extendable; a general prototype for oscillator functions is given below. The oscillator data type is `osc`.

`osc * osc_new (double phase, double duty)`

Allocate space for and initialize a new oscillator, and return a pointer to it. Cannot fail. *phase* gives the initial phase of the waveform (in radians); in most cases zero is fine. Some waveforms require a "duty cycle" parameter as well, given in *duty*. Consult the documentation of the specific oscillator used for the interpretation of the duty cycle parameter.

`void osc_free (osc * os)`

Deallocate the oscillator *os*.

`void osc_waveform (sample * buf, osc * os, freq f, unsigned count)`

Produce *count* samples according to waveform *waveform* at frequency *f* and store them in caller-provided buffer *buf*. State is kept with *os*, and it is guaranteed that n samples calculated with these functions are identical whether they were calculated with a single call or several in turn. It is the caller's responsibility to ensure that *buf* is large enough.

This is a general prototype for a family of functions referred to collectively as oscillators, each implementing a different waveform.

λ provides the following oscillators, which produce the corresponding waveforms: `osc_sine`, `osc_triangle`, `osc_square`, and `osc_sawtooth`. All of these ignore the duty cycle parameter.

5.8 Miscellaneous

The following are miscellaneous functions provided by λ for use by modules.

```
void * l_malloc (size_t size)
void * l_realloc (void * p, size_t size)
void l_free (void * p)
```

These functions are replacements for the standard library `malloc()`, `realloc()`, and `free()` functions, except that they cannot fail. If the memory request cannot be satisfied, they abort the program with an "out of memory" message instead of returning an error.

(Admittedly, this is sweeping a valid run-time error under the rug. The justification is that it makes the program much simpler, and in the days of large virtual memories performance would be so degraded as to make the program useless long before virtual memory was exhausted.)

`void l_ASSERT(int` *`expression`*`)`

If `NDEBUG` is defined, this macro expands to nothing. Otherwise, if *`expression`* is zero, this macro prints a message explaining that the program failed one of its internal sanity checks, where the failure occurred (file and line number), and brief instructions on how to report a bug; then it aborts the program. (This macro is a drop-in replacement for the C standard library sanity checker, the `assert()` macro, which prints a cryptic "assertion failed" message which is useful to programmers but confusing for normal users.)

There is an excellent chapter on using assertions effectively in Maguire [27].

Appendix A

Miscellany

This appendix contains various extra little things that don't belong in the main part of the paper but that the reader might find interesting.

A.1 Existing Non-Commercial Software

This section discusses briefly some of the other computer music software available; it is of necessity incomplete. There are hundreds of computer music-related projects in the the world. For Linux systems, the interested reader is referred to Phillips [31].

Csound [6] is the latest incarnation of the MUSIC N series of computer music languages with a genealogy leading back to the 1960's. Csound is very powerful, flexible, highly portable, and well-documented, but the learning curve is quite steep. Plain Csound is not interactive and not designed for real-time use, but there are a number of interactive and graphical wrappers for the program.

jMax, from the prestigious IRCAM (Institute of Research and Coordination in Acoustics/Music) in France, is a powerful "graphical programming environment for interactive real-time audio applications" [23]. Its paradigm is similar to λ's—a network of communicating subprograms. The Max series of programs also has an extensive history, and the various Maxen have seen a good deal of professional use. jMax runs on MacOS, Linux, and some other Unix variations. It is licensed under the GNU Public License, like λ, so there is some possiblity of source-code-level algorithm research.

Nyquist [11] is an interactive Lisp-like environment for synthesis and composition. It can work in real-time and is quite portable, running on MacOS, Windows, and several variations of Unix. Nyquist is due to Roger Dannenburg at Carnegie Mellon University.

Aura [12], also due to Roger Dannenburg at CMU, bills itself as "a new sound synthesis system designed for portability and flexibility" [13]. It runs on top of a real-time interactivity library called W [14]. Aura's paradigm is also similar to λ's, but it aims to be a much more heavyweight solution.

A.2 Future of λ

λ is by no means finished with the acceptance of this honors project. There are a few design changes that the author has in mind. The specification of port lists ought to be more general. Right now the module programmer is forced to have a predetermined limit on the number of ports, and to make many similar port the programmer must list them out. Optional FIFO semantics for port connections would be useful, for example in the case of a speech synthesizer whose input is a string rather than a signal of some sort. Also, the command line interface is awful and needs to be redesigned. Finally, there is some code cleanup to be done (at last count, there were 31 "fix me" reminders in the code).

In terms of features, the immediate next step is to program a useful graphical interface. After that the focus shifts to creating some production-quality modules—hopefully there will be some third-party help with this. In particular λ needs a powerful sequencer and modules for communication with MIDI gear. Along with all this extra code should be an automated testing framework and some sort of benchmarking method. At some point there will be a 1.0 release when λ is ready for general users.

Beyond 1.0 the future is somewhat hazy. The author would like λ to run on non-Unix systems—MacOS and Windows in particular, perhaps BeOS; that is a job for a third party, however. Also, λ will eventually run on multiprocessor machines, first shared memory multiprocessors and then clusters of workstations. Another goal is to program λ to take advantage of pro-quality gear such as DSP coprocessors and professional sound cards.

A.3 Colophon

The author works on a Debian GNU/Linux workstation and edits text with GNU Emacs. λ is written in the C programming language; the author compiles and links with the GNU `gcc` toolchain. He chases bugs with the GNU `gdb` debugger, and the automatic build system is Automake, Autoconf, and GNU Make. This paper is written in the LaTeX markup language. The list of references is held in a BibTeX database, and figures are drawn with Sketch and Gnuplot.

Appendix B

References

The following is a list of references used in the writing of this paper. For further reading the author recommends particularly: regarding programming, Bentley [1, 2], Brooks [7], Maguire [27]; regarding computer music, Dodge [18]; and regarding free software, the GNU Project's philosophy page [19] and the Debian Free Software Guidelines [30].

[1] Jon Bentley. *Programming Pearls.* Addison-Wesley, Reading, MA, 1989.

[2] Jon Bentley. *More Programming Pearls.* Addison-Wesley, Reading, MA, 1990.

[3] Michael Berry. An introduction to GrainWave. *Computer Music Journal*, 23(1):57–61, Spring 1999.

[4] W. Bolton. *Fourier Series.* Longman Scientific & Technical, Essex, England, 1995.

[5] Richard Boulanger. Introduction to sound design in Csound. In Richard Boulanger, editor, *The Csound Book.* The MIT Press, Cambridge, MA, 2000. Available online at `http://mitpress.mit.edu/e-books/csound/fpage/pub/csbook/Boulanger1/Boulang.html`.

[6] Richard Boulanger. The Csound front page. `http://mitpress.mit.edu/e-books/csound/frontpage.html`, date unavailable.

[7] Frederick P. Brooks, Jr. *The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley, Reading, MA, second edition, 1995.

[8] Alexandre Burton and Jean Piché. Cecelia v.2.0. `http://www.musique.umontreal.ca/electro/CEC/index.html`, July 1999.

[9] Roger B. Dannenburg. The implementation of Nyquist, a sound synthesis language. *Computer Music Journal*, 21(3):71–82, Fall 1997.

[10] Roger B. Dannenburg. Machine tongues XIX: Nyquist, a language for composition and sound synthesis. *Computer Music Journal*, 21(3):50–60, Fall 1997.

[11] Roger B. Dannenburg. Nyquist, a sound synthesis and composition language. `http://www.cs.cmu.edu/~rbd/nyquist.html`, July 2000.

[12] Roger B. Dannenburg. Aura: Real time distributed objects for interactive multimedia. `http://www.cs.cmu.edu/~music/aura/`, date unavailable.

[13] Roger B. Dannenburg and Eli Brandt. A flexible real-time software synthesis system. In *Proceedings of the 1996 International Computer Music Conference*, pages 270–273. International Computer Music Association, August 1996. Available online at `http://www.cs.cmu.edu/People/rbd/papers/aura.pdf`.

[14] Roger B. Dannenburg and Dean Rubine. Toward modular, portable real-time software. In *Proceedings of the 1995 International Computer Music Conference*, pages 65–72. International Computer Music Association, September 1995. Available online at `http://www.cs.cmu.edu/People/rbd/papers/wpaper.pdf`.

[15] Giovanni de Poli. Audio signal processing by computer. In Goffredo Haus, editor, *Musicprocessing*, pages 73–105. A-R Editions, Inc., Madison, WI, 1993.

[16] François Déchelle, Ricardo Borghesi, Maurizo de Cecco, Enzo Maggi, Butch Rovan, and Norbert Schnell. jMax: An environment for real-time musical applications. *Computer Music Journal*, 23(3):50–58, Summer 1999.

[17] Charles Dodge and Thomas A. Jerse. *Computer Music: Synthesis, Composition, and Performance*. Schirmer Books, New York, first edition, 1985.

[18] Charles Dodge and Thomas A. Jerse. *Computer Music: Synthesis, Composition, and Performance*. Schirmer Books, New York, second edition, 1997.

[19] Free Software Foundation. Philosophy of the GNU Project. `http://www.gnu.org/philosophy/`, April 2001.

[20] comp.dsp frequently asked questions. `http://www.bdti.com/faq/`, January 2000.

[21] Hubert S. Howe. *Electronic Music Synthesis: Concepts, Facilities, Techniques*. W. W. Norton & Company, Inc., New York, 1975.

[22] Takebumi Itakagi, Peter D. Manning, and Alan Purvis. Distributed parallel processing: Lessons learned from a 160-transputer network. *Computer Music Journal*, 21(4):42–54, Winter 1997.

[23] jMax home page. `http://www.ircam.fr/jmax`, 1999.

[24] Matti Karjalainen, Vesa Välimäki, and Tero Tolonen. Plucked-string models: From the Karpus-Strong algorithm to digital waveguides and beyond. *Computer Music Journal*, 22(3):17–32, Fall 1998.

[25] Brian W. Kernighan and Rob Pike. *The Practice of Programming*. Addison-Wesley, Reading, MA, 1999.

[26] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall PTR, Englewood Cliffs, NJ, second edition, 1988.

[27] Steve Maguire. *Writing Solid Code*. Microsoft Press, Redmond, WA, 1993.

[28] Peter Manning. *Electronic and Computer Music*. Clarendon Press, Oxford, second edition, 1993.

[29] Steve McConnell. *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, Redmond, WA, 1993.

[30] Bruce Perens. Debian free software guidelines. `http://www.debian.org/social_contract`, October 2000.

[31] Dave Phillips. Sound & MIDI software for Linux. `http://sound.condorow.net/`, December 2000.

[32] Jean Piché and Alexandre Burton. Cecilia: A production interface to Csound. *Computer Music Journal*, 22(2):52–55, Summer 1998.

[33] Nick Porcaro, David Jaffe, Pat Scandalis, Julius Smith, Tim Stilson, and Scott van Dyne. SynthBuilder: A graphical rapid-prototyping tool for the development of music synthesis and effects patches on multiple platforms. *Computer Music Journal*, 22(2):35–43, Summer 1998.

[34] Curtis Roads and John Strawn, editors. *Foundations of Computer Music*. The MIT Press, Cambridge, MA, 1985.

[35] John Strawn, editor. *Digital Audio Signal Processing: An Anthology*. William Kaufmann, Inc., Los Altos, CA, 1985.

[36] Barry Vercoe. A beginning tutorial. part of [6], date unavailable.

[37] Godric Wilkie. Sound glossary. `http://www.gozen.demon.co.uk/godric/sound/fglossary.html`, January 1999.

Appendix C

Source Code

The following pages list λ's C source code. Only the source itself is listed; build scripts (Makefiles) and other miscellany are not included. Of course, no one is silly enough to type in this source manually and try to compile it, and besides it is probably out of date anyway. A proper packaging of the machine-readable source is available at the λ home page, `http://reidster.net/lambda/`.

include/lambda.h

```
/* $Id: lambda.h,v 1.8 2001/04/20 05:48:10 reid Exp $ */

/* This file is the interface file for the module API and should be included
   by all modules. Note, however, that it contains things which are *not* part
   of the defined API (e.g. structure member names, etc.); don't troll this
   file for how-to information. The API manual is the definitive reference --
   if you can't figure it out from there, please file a bug report. */

#ifndef LAMBDA_H
#define LAMBDA_H

#include <config.h>
#include <stdlib.h> /* for size_t */


/*** Constants ***/

/* Global parameter defaults. */
#define DEFAULT_SAMPLE_RATE 44100
#define DEFAULT_SAMPLES_PER_TICK 256
#define DEFAULT_INPUT_MICROSECONDS 100000

/* Modules' magic number. */
#define MOD_MAGIC_1_0_0 0x6c010000
```

```
/* Maximum number of dependencies a port can have. */
#define MAX_PORT_DEPS 15


/*** Types ***/

typedef enum {
    OK,
    ERR,     /* general error */
    ERR_EOF, /* end-of-file encountered */
} lerr;

/* Maximum size of a scalar object described by an ltyp structure. */
#define L_SIZEOF_MAX sizeof(double)

/* The following mess defines, with the help of autoconf, integers with
   specific sizes. C9x defines integers with specific types in <inttypes.h>,
   so trying those first is a possible FIXME. */

#if SIZEOF_INT == 2
    typedef int int16;
    typedef unsigned int uint16;
#elif SIZEOF_SHORT == 2
    typedef short int16;
    typedef unsigned short uint16;
#else
    "can't find a 16 bit integral type"
#endif

#if SIZEOF_INT == 4
    typedef int int32;
    typedef unsigned int uint32;
#elif SIZEOF_LONG == 4
    typedef long int32;
    typedef unsigned long uint32;
#else
    "can't find a 32 bit integral type"
#endif

#if SIZEOF_LONG_LONG == 8
    typedef long long int64;
    typedef unsigned long long uint64;
#else
    "can't find a 64 bit integral type"
#endif

/* Some more normal typedefs. */
typedef unsigned char byte;
typedef int bool;
typedef double sample;
```

```
typedef double freq;
typedef double dur;
typedef double level;

/* The list of "raw" ltyp type descriptors. Order here is very important; see
   the comment in lcore/types.c. */
typedef enum {
    LTYP_BOOL,
    LTYP_BYTE,
    LTYP_INT,
    LTYP_LONG,
    LTYP_INT16,
    LTYP_UINT16,
    LTYP_INT32,
    LTYP_UINT32,
    LTYP_INT64,
    LTYP_UINT64,
    LTYP_FLOAT,
    LTYP_DOUBLE,
    LTYP_SAMPLE,
    LTYP_FREQ,
    LTYP_DUR,
    LTYP_LEVEL
} raw_ltyp;

/* The list of "meta" ltyp descriptors -- whether an ltyp describes a scalar,
   array, etc. Order here is not important. */
typedef enum {
    SCALAR,   /* n_elements ignored */
    ARRAY,    /* n_elements is size of array */
    ARRAY_REL /* n_elements * samples_per_tick() is size of array */
} meta_ltyp;

/* Da-da-da-dum, the ltyp type descriptor structure. */
typedef struct {
    meta_ltyp meta_type;
    raw_ltyp raw_type;
    unsigned n_elements;
} ltyp;

/* A couple of helper types for port and friends. */
enum port_direction {IN, OUT};
enum port_strictness {NONSTRICT = 0, STRICT};

/* Another central data structure, the port. This type is used for both input
   and output ports; the differences are as follows.

   Input ports: refcount is not used and should always be zero. If source is
   non-NULL, then the port is connected and source points to the source output
   port and contents points to source->contents. If source is NULL and
```

```
   contents is non-NULL, then the port is set to the value pointed to by
   contents. Otherwise, the port is unconnected and unset.

   Output ports: source is not used and should always be NULL. refcount is the
   number of connections, and contents points to the value of the port. */
struct port_struct {
    char * name;
    struct awid_struct * parent;
    enum port_direction direction;
    ltyp * type;
    enum port_strictness strict;
    char ** deps;
    unsigned refcount;
    struct port_struct * source;
    void * contents;
};
typedef struct port_struct port;

/* Port declaration. */
typedef struct {
    char * name;
    enum port_direction direction;
    meta_ltyp meta_type; /* this and the next two are identical to meta_ltyp */
    raw_ltyp raw_type;
    unsigned n_elements;
    enum port_strictness strict;
    char * depends[MAX_PORT_DEPS + 1];
} port_decl;

/* Port declaration list terminator. A port declaration with name == NULL
   terminates the list of declarations. The rest of this is arbitrary. */
#define END_PORT_DECLARATIONS { NULL, IN, SCALAR, 0, LTYP_INT, STRICT, {} }

/* The oscillator data structure. */
typedef struct {
    double phase;
    double duty;
} osc;


/*** Functions ***/

/* Functions defined by modules. */
lerr lm_go(void ** context, unsigned n_ports, port * ports[]);
lerr lm_pause(void * context);
lerr lm_stop(void * context);
lerr lm_tick(void * context);

/* Assertions. (l_assert.c) */
void _l_assert(char * file, int line);
```

```
#ifndef NDEBUG
#define l_ASSERT(expr) {if (!(expr)) _l_assert(__FILE__, __LINE__);}
#else
#define l_ASSERT(expr)
#endif

/* Type manipulation. (types.c) */
size_t l_sizeof(ltyp * t);
ltyp * ltyp_new(meta_ltyp meta_type, raw_ltyp raw_type, unsigned n_elements);
void ltyp_free(ltyp * t);
bool ltyp_eq(ltyp * t1, ltyp * t2);
char * ltyp_str(ltyp * t);
char * ltyp_tostr(ltyp * t, void * value);
lerr ltyp_fromstr(void * buf, ltyp * t, char * str);

/* Port operations. (ports.c) */
void l_send(port * p, void * buf);
void l_recv(port * p, void * buf);
port * port_addrof(char * name, unsigned n_ports, port * ports[]);
bool port_ready(port * p);
void _port_verify(port * p);
#ifndef NDEBUG
#define port_VERIFY(p) _port_verify(p)
#else
#define port_VERIFY(p)
#endif

/* Timing information. (timing.c) */
inline freq samples_per_second(void);
inline dur seconds_per_sample(void);
inline unsigned samples_per_tick(void);
inline unsigned samples_calc(void);
inline dur seconds_calc(void);

/* Oscillators. (osc.c) */
osc * osc_new(double phase, double duty);
void osc_free(osc * os);
void osc_sine(sample * buf, osc * os, freq f, unsigned count);
void osc_square(sample * buf, osc * os, freq f, unsigned count);
void osc_triangle(sample * buf, osc * os, freq f, unsigned count);
void osc_sawtooth(sample * buf, osc * os, freq f, unsigned count);

/* Memory management. (memory.c) */
void * l_malloc(size_t size);
void * l_realloc(void * p, size_t new_size);
void l_free(void * p);

/* Terminal output. (term_output.c) */
void pr(char * fmt, ...);
void error_pr(char * fmt, ...);
```

```
void debug_pr(char * fmt, ...);

/* Error handling. (errors.c) */
char * err_get(void);
char * err_copy(void);
void err_set(char * fmt, ...);
void err_prepend(char * fmt, ...);
void err_ignore_on(void);
void err_ignore_off(void);

#endif /* !LAMBDA_H */
```

lcore/lcore.h

```
/* $Id: lcore.h,v 1.6 2001/04/20 05:48:27 reid Exp $ */

/* This file defines all interfaces exported by lcore source files for use by
   other parts of lcore. */

#ifndef LCORE_H
#define LCORE_H

#include "lambda.h"

#include <stdio.h> /* for FILE structure */


/*** Crud ***/

/* FIXME: for some reason we're not finding the prototypes for these two
   functions. Look into it... */
#include <stdarg.h>
int asprintf(char ** ptr, const char * fmt, ...);
int vasprintf(char ** ptr, const char * fmt, va_list ap);


/*** Constants ***/

/* When speed is limited (i.e. when running in "real-time mode"), leave this
   much buffer time -- i.e., if a tick finishes calculation early, sleep until
   this much time is left. This should be as small as possible.

   FIXME: make this a command-line option. */
#define SLUSH_USECS 1000

/* If usleep()ing by less than this, busy-wait instead of actually sleeping. */
#define USLEEP_BUSYWAIT_USECS 10

/* The maximum length of an error string, including terminator byte. */
```

```
#define ERR_CHARS_MAX 4096


/*** Types ***/

/* Module. */
typedef struct {
    void * handle;
    char * name;
    char * version;
    char * desc;
    port_decl * decls;
    lerr (*go)(void **, unsigned, port **);
    lerr (*pause)(void *);
    lerr (*stop)(void *);
    lerr (*tick)(void *);
} module;

/* Arena widget. */
struct awid_struct {
    char * name;
    module * mod;
    unsigned n_ports;
    port ** ports;
    void * context;
    struct awid_struct * next;
};
typedef struct awid_struct awid;

/* Set of possible arena states. */
typedef enum { STOP, PAUSE, GO } state_t;

/* Arena. */
typedef struct {
    awid * first;
    state_t state;
    bool dirty;
} arena;


/*** Functions ***/

/* cli.c */
lerr cli_process_line(FILE * fp);
void cli_prompt(void);

/* modules.c */
module * load_module(char * name);
void unload_module(module * mod);
```

```
/* arena.c */
lerr arena_clear(void);
lerr arena_load(char * mod_name, char * awid_name);
lerr arena_delete(char * awid_name);
lerr arena_connect(char * src_awid_name, char * src_port_name,
                   char * dst_awid_name, char * dst_port_name);
lerr arena_disconnect(char * src_awid_name, char * src_port_name,
                      char * dst_awid_name, char * dst_port_name);
lerr arena_set(char * dst_awid_name, char * dst_port_name, char * str);
lerr arena_unset(char * dst_awid_name, char * dst_port_name);
lerr arena_print(void);
lerr arena_ready(void);

/* awidgets.c */
awid * awid_new(char * mod_name, char * awid_name);
void awid_free(awid * a);
awid * awid_addrof(char * name);
port * awid_get_port(char * awid_name, char * port_name);
void awid_pr(awid * a);
lerr awid_check_deps(awid * a);

/* states.c */
lerr enter_go(void);

lerr enter_pause(void);
lerr enter_stop(awid * limit);
lerr do_tick(void);
char * state_str(void);
state_t state(void);

/* ports.c */
port * port_new(struct awid_struct * parent, port_decl * decl);
void port_free(port * p);

/* timing.c */
uint64 usecs_wall(void);
void l_usleep(uint64 usecs);
void timing_init(freq sr, unsigned tl, bool limit);
void timing_pausego(void);
void timing_stopgo(void);
void timing_tick_start(void);
void timing_tick_end(void);
double timing_utilization(void);
void timing_stats_pr(void);

/* util.c */
bool l_feof(FILE * fp);
int min(int a, int b);

#endif /* !LCORE_H */
```

lcore/main.c

```
/* $Id: main.c,v 1.7 2001/04/20 05:48:27 reid Exp $ */

/* This file contains the startup/shutdown stuff and the main loop. */

#include "lambda.h"
#include "lcore.h"

#include <getopt.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>


/*** Globals ***/

unsigned input_usecs; /* minimum time to wait before checking for input */
extern bool g_print_debug; /* from term_output.c */


/*** Private prototypes ***/

void setup(int argc, char ** argv);
int shutdown(void);


/*** Public functions ***/

int main(int argc, char ** argv)
{
    lerr result;
    uint64 input_last;

    pr("lambda version %s starting, please mind the rotor wash.\n",
       VERSION);

    setup(argc, argv);

    l_ASSERT(state() == STOP);
    result = OK;
    cli_prompt();
    while (result != ERR_EOF) {
        if (state() == GO) {
            timing_tick_start();
            result = do_tick();
            if (result != OK) {
                pr("\n? %s\n", err_get());
                result = enter_stop(NULL);
```

```
                if (result != OK)
                    pr("? %s\n", err_get());
                cli_prompt();
            }
            if (usecs_wall() > input_last + input_usecs) {
                input_last = usecs_wall();
                if (!l_feof(stdin)) {
                    result = cli_process_line(stdin);
                    if (result == ERR)
                        pr("? %s\n", err_get());
                    cli_prompt();
                }
            }
            timing_tick_end();
        } else {
            input_last = usecs_wall();
            result = cli_process_line(stdin);
            if (result == ERR)
                pr("? %s\n", err_get());
            cli_prompt();
        }
    }

    return shutdown();
}


/*** Private functions ***/

/* Set up the program, including command line option parsing. */
void setup(int argc, char ** argv)
{
    unsigned seed;
    freq sample_rate;
    unsigned tick_len;
    bool limit_realtime;
    int opt, optidx;
    const struct option options[] = {
        {"seed", required_argument, NULL, 's'},
        {"sample-rate", required_argument, NULL, 'r'},
        {"samples-per-tick", required_argument, NULL, 't'},
        {"input-microseconds", required_argument, NULL, 'i'},
        {"debug", no_argument, NULL, 'd'},
        {"no-speed-limit", no_argument, NULL, 'l'},
        {NULL, 0, 0, 0 } };

    /* Set globals to their defaults. */
    sample_rate = DEFAULT_SAMPLE_RATE;
    tick_len = DEFAULT_SAMPLES_PER_TICK;
    limit_realtime = 1;
```

```
    input_usecs = DEFAULT_INPUT_MICROSECONDS;
    g_print_debug = 0;
    seed = time(NULL);

    /* Process command line args. */
    /* FIXME: check validity of input. */
    while ((opt = getopt_long(argc, argv, "s:r:t:i:dl", options, &optidx))
           != -1) {
        switch (opt) {
        case 's':
            seed = (unsigned int) strtoul(optarg, NULL, 10);
            break;
        case 'r':
            sample_rate = (freq) strtod(optarg, NULL);
            break;
        case 't':
            tick_len = (unsigned) strtoul(optarg, NULL, 10);
            break;
        case 'i':
            input_usecs = (unsigned) strtoul(optarg, NULL, 10);
            break;
        case 'd':
            g_print_debug = 1;
            break;
        case 'l':
            limit_realtime = 0;
            break;
        default:
            l_ASSERT(0);
            break;
        }
    }

    /* Initialize subsystems. */
    srand(seed);
    timing_init(sample_rate, tick_len, limit_realtime);

    /* Tell the world. */
    debug_pr("debugging output: enabled\n");
#ifndef NDEBUG
    debug_pr("assertions:       enabled\n");
#else
    debug_pr("assertions:       disabled\n");
#endif
    debug_pr("\n");
    debug_pr("random seed: %u\n", seed);
    debug_pr("sample rate: %f\n", sample_rate);
    debug_pr("samples per tick: %u\n", tick_len);
    debug_pr("input microseconds: %u\n", input_usecs);
    debug_pr("speed limiting: %s\n", limit_realtime ? "yes" : "no");
```

```
}

/* Clean up. */
int shutdown(void)
{
    pr("\nGoodbye.\n");

    return 0;
}
```

lcore/arena.c

```
/* $Id: arena.c,v 1.3 2001/04/20 05:48:27 reid Exp $ */

/* This file implements the arena management stuff. */

#include "lambda.h"
#include "lcore.h"


/*** Globals ***/

arena g_arena = { NULL, STOP, 0 };


/*** Public Functions ***/

/* Cleanly make the arena empty, deleting all widgets. */
lerr arena_clear(void)
{
    awid * cur, * next;

    l_ASSERT(state() == STOP);

    cur = g_arena.first;
    while (cur != NULL) {
        next = cur->next;
        awid_free(cur);
        cur = next;
    }
    g_arena.first = NULL;

    g_arena.dirty = 0;
    return OK;
}

/* Load the module named mod_name and create an arena widget from it named
   awid_name; add that widget to the arena. */
lerr arena_load(char * mod_name, char * awid_name)
```

```
{
    awid * a;

    l_ASSERT(state() == STOP);

    a = awid_new(mod_name, awid_name);
    if (a == NULL)
        return ERR;

    a->next = g_arena.first;
    g_arena.first = a;

    g_arena.dirty = 1;
    return OK;
}

/* Remove the widget named awid_name and deallocate it. */
lerr arena_delete(char * awid_name)
{
    awid * cur, * prev;

    l_ASSERT(state() == STOP);

    if (g_arena.first != NULL) {
        if (!strcmp(g_arena.first->name, awid_name)) {
            cur = g_arena.first;
            g_arena.first = cur->next;
            awid_free(cur);
            g_arena.dirty = 1;
            return OK;
        } else {
            prev = g_arena.first;
            cur = prev->next;
            while (cur != NULL) {
                if (!strcmp(cur->name, awid_name)) {
                    prev->next = cur->next;
                    awid_free(cur);
                    g_arena.dirty = 1;
                    return OK;
                }
                prev = cur;
                cur = prev->next;
            }
        }
    }

    err_set("widget %s does not exist", awid_name);
    return ERR;
}
```

```
/* Connect the output port named src_port_name of the widget named
   src_awid_name to the input port named dst_port_name of the widget named
   dst_awid_name. On success, return OK; on error, set the error string and
   return ERR and do not change anything. */
lerr arena_connect(char * src_awid_name, char * src_port_name,
            char * dst_awid_name, char * dst_port_name)
{
    port * src_p, * dst_p;
    char * s;

    l_ASSERT(state() == STOP);

    if (!(src_p = awid_get_port(src_awid_name, src_port_name))
        || !(dst_p = awid_get_port(dst_awid_name, dst_port_name)))
        return ERR;
    port_VERIFY(src_p);
    port_VERIFY(dst_p);

    if (src_p->direction != OUT) {
        err_set("%s/%s: not an output port", src_p->parent->name,
                src_p->name);
        return ERR;
    }
    if (dst_p->direction != IN) {
        err_set("%s/%s: not an input port", dst_p->parent->name,
                dst_p->name);
        return ERR;
    }
    if (!ltyp_eq(src_p->type, dst_p->type)) {
        err_set("%s/%s -> %s/%s: types of ports do not match",
                src_p->parent->name, src_p->name, dst_p->parent->name,
                dst_p->name);
        return ERR;
    }
    if (dst_p->source != NULL) {
        err_set("%s/%s: already connected to by %s/%s", dst_awid_name,
                dst_port_name, dst_p->source->name,
                dst_p->source->parent->name);
        return ERR;
    }
    if (dst_p->contents != NULL) {
        s = ltyp_tostr(dst_p->type, dst_p->contents);
        err_set("%s/%s: already set to %s\n",
                dst_awid_name, dst_port_name, s);
        l_free(s);
        return ERR;
    }

    src_p->refcount++;
    if (src_p->refcount == 1)
```

```
        src_p->contents = l_malloc(l_sizeof(src_p->type));
    dst_p->source = src_p;
    dst_p->contents = src_p->contents;

    port_VERIFY(src_p);
    port_VERIFY(dst_p);
    g_arena.dirty = 1;
    return OK;
}

/* Delete the connection from the port named src_port_name on the widget named
   src_awid_name to the port named dst_port_name on the widget named
   dst_awid_name. */
lerr arena_disconnect(char * src_awid_name, char * src_port_name,
                char * dst_awid_name, char * dst_port_name)
{
    port * src_p;
    port * dst_p;

    l_ASSERT(state() == STOP);

    if (!(src_p = awid_get_port(src_awid_name, src_port_name))
        || !(dst_p = awid_get_port(dst_awid_name, dst_port_name)))
        return ERR;
    port_VERIFY(src_p);
    port_VERIFY(dst_p);

    if (src_p->direction != OUT) {
        err_set("%s/%s: not an output port", src_p->parent->name,
                src_p->name);
        return ERR;
    }
    if (dst_p->direction != IN) {
        err_set("%s/%s: not an input port", dst_p->parent->name,
                dst_p->name);
        return ERR;
    }
    if (dst_p->source != src_p) {
        err_set("%s/%s -> %s/%s: not connected", src_awid_name,
                src_port_name, dst_awid_name, dst_port_name);
        return ERR;
    }

    l_ASSERT(src_p->refcount > 0);
    src_p->refcount--;
    if (src_p->refcount == 0) {
        l_free(src_p->contents);
        src_p->contents = NULL;
    }
    dst_p->source = NULL;
```

```
    dst_p->contents = NULL;

    port_VERIFY(src_p);
    port_VERIFY(dst_p);
    g_arena.dirty = 1;
    return OK;
}

/* Set the input port named dst_port_name on the widget named dst_awid_name to
   a value whose string representation is s. */
lerr arena_set(char * dst_awid_name, char * dst_port_name, char * s)
{
    port * dst_p;
    bool first;

    if (!(dst_p = awid_get_port(dst_awid_name, dst_port_name)))
        return ERR;
    port_VERIFY(dst_p);

    if (dst_p->direction != IN) {
        err_set("%s/%s: not an input port", dst_awid_name, dst_port_name);
        return ERR;
    }
    if (dst_p->source != NULL) {
        err_set("%s/%s -> %s/%s: already connected", dst_awid_name,
                dst_port_name, dst_p->source->name,
                dst_p->source->parent->name);
        return ERR;
    }

    /* dst_p->contents is altered in-place to avoid calling l_malloc() if it's
       being reset, because it is preferred not to call l_malloc() when not in
       the stop state. Hence the messiness backing out the memory allocation
       if s is not interpretable. */
    if (dst_p->contents == NULL) {
        if (state() != STOP) {
            err_set("can set a previously unset port only in stop state");
            return ERR;
        }
        dst_p->contents = l_malloc(l_sizeof(dst_p->type));
        first = 1;
    } else
        first = 0;
    if (ltyp_fromstr(dst_p->contents, dst_p->type, s) != OK) {
        err_prepend("cannot interpret `%s': ", s);
        if (first) {
            l_free(dst_p->contents);
            dst_p->contents = NULL;
        }
        return ERR;
```

```
    }

    port_VERIFY(dst_p);
    if (first)
        g_arena.dirty = 1;
    return OK;
}

/* Unset the port named dst_port_name on the widget named dst_awid_name. */
lerr arena_unset(char * dst_awid_name, char * dst_port_name)
{
    port * dst_p;

    l_ASSERT(state() == STOP);

    if (!(dst_p = awid_get_port(dst_awid_name, dst_port_name)))
        return ERR;
    port_VERIFY(dst_p);

    if (dst_p->direction != IN) {
        err_set("%s/%s: not an input port", dst_awid_name, dst_port_name);
        return ERR;
    }
    if (!(dst_p->source == NULL && dst_p->contents != NULL)) {
        err_set("%s/%s: not set", dst_awid_name, dst_port_name);
        return ERR;
    }

    l_free(dst_p->contents);
    dst_p->contents = NULL;

    port_VERIFY(dst_p);
    g_arena.dirty = 1;
    return OK;
}

/* Verify that the arena is ready to enter the go state--check that each
   module's port dependencies are satisfied. If so, return OK and clear the
   arena dirty bit; otherwise, return ERR and set the error string
   appropriately. */
lerr arena_ready(void)
{
    awid * a;

    a = g_arena.first;
    while (a != NULL) {
        if (awid_check_deps(a) != OK) {
            l_ASSERT(g_arena.dirty);
            err_prepend("%s: ", a->name);
            return ERR;
```

```
        }
        a = a->next;
    }

    g_arena.dirty = 0;
    return OK;
}

/* Pretty-print the arena. */
lerr arena_print(void)
{
    awid * cur;

    pr("Arena is %s.\n", g_arena.dirty ? "dirty" : "clean");

    cur = g_arena.first;
    while (cur != NULL) {
        awid_pr(cur);
        cur = cur->next;
    }

    return OK;
}
```

lcore/awidgets.c

```
/* $Id: awidgets.c,v 1.4 2001/04/20 05:48:27 reid Exp $ */

/* This file implements the widget manipulation functions. */

#include "lambda.h"
#include "lcore.h"

/*** Globals ***/

extern arena g_arena;


/*** Public functions ***/

/* Build an arena widget, named awid_name, as an instance of the module named
   mod_name. Sets up all fields but does not link the new awid into the
   arena's linked list -- the next field is set to NULL. On success, return a
   pointer to the node; on error, return NULL and set the error string. */
awid * awid_new(char * mod_name, char * awid_name)
{
    awid * a;
    module * m;
    int i, j;
```

```
if (awid_addrof(awid_name)) {
    err_set("widget name %s is in use", awid_name);
    return NULL;
}

m = load_module(mod_name);
if (m == NULL)
    return NULL;

a = l_malloc(sizeof(awid));
a->mod = m;
a->name = l_malloc(strlen(awid_name) + 1);
strcpy(a->name, awid_name);
a->next = NULL;

a->n_ports = 0;
a->ports = NULL;
for (i = 0; a->mod->decls[i].name != NULL; i++) {
    /* Sanity check, in case the module forgot to terminate the
       declaration list or something else is broken. FIXME: this should go
       into a decls_verify() function. */
    if (!(a->mod->decls[i].direction == IN
          || a->mod->decls[i].direction == OUT)
        || !(a->mod->decls[i].meta_type == SCALAR
             || a->mod->decls[i].meta_type == ARRAY
             || a->mod->decls[i].meta_type == ARRAY_REL)
        || strlen(a->mod->decls[i].name) == 0) {
        err_set("module %s is broken: object `ports[%d]' is invalid",
                a->mod->name, i);
        awid_free(a);
        return NULL;
    }

    /* Verify that the port name is unique. */
    for (j = 0; j < a->n_ports; j++)
        if (!strcmp(a->mod->decls[i].name, a->ports[j]->name)) {
            err_set("module %s is broken: duplicate port name %s",
                    a->mod->name, a->mod->decls[i].name);
            awid_free(a);
            return NULL;
        }

    a->n_ports++;
    a->ports = l_realloc(a->ports, a->n_ports * sizeof(port *));
    a->ports[a->n_ports - 1] = port_new(a, &(a->mod->decls[i]));
}

/* Verify that each port has a valid number of dependencies and that no
   port depends on a nonexistent port. */
```

```
    for (i = 0; i < a->n_ports; i++) {
        for (j = 0; a->ports[i]->deps[j] != NULL; j++) {
            if (j == MAX_PORT_DEPS) {
                err_set("module %s is broken: too many port dependencies",
                        a->mod->name);
                awid_free(a);
                return NULL;
            }
            if (!port_addrof(a->ports[i]->deps[j], a->n_ports, a->ports)) {
                err_set("module %s is broken: port %s depends on nonexistent"
                        " port %s", a->mod->name, a->ports[i]->name,
                        a->ports[i]->deps[j]);
                awid_free(a);
                return NULL;
            }
        }
    }

    return a;
}

/* Free all storage associated with a. Postcondition: a points to garbage. */
void awid_free(awid * a)
{
    int i;

    /* FIXME: need to break all connections involving n. */

    unload_module(a->mod);
    l_free(a->name);
    for (i = 0; i < a->n_ports; i++)
        port_free(a->ports[i]);
    l_free(a->ports);
    l_free(a);
}

/* Return the address of the widget in the arena named name, or NULL if no
   widget matches. */
awid * awid_addrof(char * name)
{
    awid * cur;

    cur = g_arena.first;
    while (cur != NULL) {
        if (!strcmp(cur->name, name))
            return cur;
        cur = cur->next;
    }

    return NULL;
```

```
}

/* If the widget named awid_name has an input port named port_name, return a
   pointer to it; otherwise, set the error string and and return NULL. */
port * awid_get_port(char * awid_name, char * port_name)
{
    awid * a;
    port * p;

    a = awid_addrof(awid_name);
if (a == NULL) {
        err_set("widget %s does not exist", awid_name);
        return NULL;
    }
    p = port_addrof(port_name, a->n_ports, a->ports);
    if (p == NULL) {
        err_set("widget %s has no input port %s", awid_name, port_name);
        return NULL;
    }

    return p;
}

/* Verify that all of an arena widget a's port dependencies are satisfied. If
   so, return OK; if not, return ERR and set the error string. */
lerr awid_check_deps(awid * a)
{
    port * target;
    int i, j;

    for (i = 0; i < a->n_ports; i++) {
        port_VERIFY(a->ports[i]);
        if (port_ready(a->ports[i]) || a->ports[i]->strict) {
            for (j = 0; a->ports[i]->deps[j] != NULL; j++) {
                target = port_addrof(a->ports[i]->deps[j], a->n_ports,
                                     a->ports);
                l_ASSERT(target != NULL);
                if (!port_ready(target)) {
                    if (port_ready(a->ports[i])) {
                        err_set("port %s depends on %s, not satisfied",
                                a->ports[i]->name, target->name);
                        return ERR;
                    } else
                        break;
                }
            }

            if (a->ports[i]->deps[j] == NULL
                && a->ports[i]->strict && !port_ready(a->ports[i])) {
                err_set("port %s is strict and not ready", a->ports[i]->name);
```

```
                return ERR;
            }
        }
    }

    return OK;
}

/* Pretty-print an arena widget a. */
void awid_pr(awid * a)
{
    int i, j;
    char * s;

    pr("Widget %s (%s %s).\n", a->name, a->mod->name,
       a->mod->version);
    for (i = 0; i < a->n_ports; i++) {
        s = ltyp_str(a->ports[i]->type);
        pr("    %s: %s %s", a->ports[i]->name,
           (a->ports[i]->direction == IN) ? "in" : "out", s);
        l_free(s);
        if (a->ports[i]->direction == IN) {
            if (a->ports[i]->source != NULL) {
                pr(" <- %s/%s", a->ports[i]->source->parent->name,
                   a->ports[i]->source->name);
            } else if(a->ports[i]->contents != NULL) {
                s = ltyp_tostr(a->ports[i]->type, a->ports[i]->contents);
                pr(" <- %s", s);
                l_free(s);
            }
        } else {
            /* FIXME: also print connections for output ports */
        }
        pr(" <%s", a->ports[i]->strict ? "strict" : "nonstrict" );
        for (j = 0; a->ports[i]->deps[j] != NULL; j++) {
            pr(" %s", a->ports[i]->deps[j]);
        }
        pr(">", s);
        pr("\n");
    }
}
```

lcore/cli.c

```
/* $Id: cli.c,v 1.9 2001/04/22 18:59:44 reid Exp $ */

/* This file implements the command line interface. */

#include "lambda.h"
```

```
#include "lcore.h"

#include <errno.h>
#include <string.h>


/*** Constants ***/

#define LINE_MAX 512
#define MAX_ARGS 8


/*** Public functions ***/

/* Read and process one line of command input. Returns OK on success, ERR on
   error (and sets the error string), and ERR_EOF on end of file or user exit.

   FIXME: this routine is ugly and should be completely redesigned. (The use
          of goto is not part of this ugliness.) */
lerr cli_process_line(FILE * input)
{
    char line[LINE_MAX];
    char * command;
    char * args[MAX_ARGS];
    int lines, nargs;
    lerr result;
    FILE * fp;

    if (fgets(line, LINE_MAX, input) == NULL) /* EOF */
        return ERR_EOF;

    command = strtok(line, " \n");
    if (command == NULL)
        return OK;
    nargs = 0;
    while (nargs < MAX_ARGS
           && (args[nargs] = strtok(NULL, " /\n")) != NULL)
        nargs++;

    if (!strcmp(command, "go")) {
        if (state() == GO)
            goto state_err;
        if (nargs != 0)
            goto args_err;
        return enter_go();
    }
    else if (!strcmp(command, "pause")) {
        if (state() != GO)
            goto state_err;
        if (nargs != 0)
```

```
            goto args_err;
        result = enter_pause();
        if (result == OK) {
            timing_stats_pr();
            return OK;
        } else
            return result;
    }
    else if (!strcmp(command, "stop")) {
        if (state() == STOP)
            goto state_err;
        if (nargs != 0)
            goto args_err;
        result = enter_stop(NULL);
        if (result == OK) {
            timing_stats_pr();
            return OK;
        } else
            return result;
    }
    else if (!strcmp(command, "load")) {
        if (state() != STOP)
            goto state_err;
        if (nargs != 2)
            goto args_err;
        return arena_load(args[0], args[1]);
    }
    else if (!strcmp(command, "delete")) {
        if (state() != STOP)
            goto state_err;
        if (nargs != 1)
            goto args_err;
        return arena_delete(args[0]);
    }
    else if (!strcmp(command, "connect")) {
        if (state() != STOP)
            goto state_err;
        if (nargs != 4)
            goto args_err;
        return arena_connect(args[0], args[1], args[2], args[3]);
    }
    else if (!strcmp(command, "disconnect")) {
        if (state() != STOP)
            goto state_err;
        if (nargs != 4)
            goto args_err;
        return arena_disconnect(args[0], args[1], args[2], args[3]);
    }
    else if (!strcmp(command, "set")) {
        /* allowed in all states */
```

```
        if (nargs != 3)
            goto args_err;
        return arena_set(args[0], args[1], args[2]);
    }
    else if (!strcmp(command, "unset")) {
        if (state() != STOP)
            goto state_err;
        if (nargs != 2)
            goto args_err;
        return arena_unset(args[0], args[1]);
    }
    else if (!strcmp(command, "print")) {
        /* allowed in all states */
        if (nargs != 0)
            goto args_err;
        return arena_print();
    }
    else if (!strcmp(command, "verify")) {
        if (state() != STOP)
            goto state_err;
        if (nargs != 0)
            goto args_err;
        return arena_ready();
    }
    else if (!strcmp(command, "clear")) {
        if (state() != STOP)
            goto state_err;
        if (nargs != 0)
            goto args_err;
        return arena_clear();
    }
    else if (!strcmp(command, "halt")) {
        err_set("you are not worthy");
        return ERR;
    }
    else if (!strcmp(command, "read")) {
        if (state() != STOP)
            goto state_err;
        if (nargs != 1)
            goto args_err;
        fp = fopen(args[0], "r");
        if (fp == NULL) {
            err_set("can't open %s: %s", args[0], strerror(errno));
            return ERR;
        }
        result = OK;
        lines = 0;
        while (result != ERR_EOF) {
            lines++;
            result = cli_process_line(fp);
```

```
            if (result == ERR) {
                err_prepend("%s:%d: ", args[0], lines);
                break;
            }
        }
        if(fclose(fp)) {
            err_set("error closing %s: %s", args[0], strerror(errno));
            return ERR;
        }
        if(result == ERR) {
            /* already set error string */
            return ERR;
        }
        return OK;
    }
    else if (!strcmp(command, "quit")) {
        if (nargs != 0)
            goto args_err;
        if (state() != STOP)
            enter_stop(NULL);
        return ERR_EOF;
    }
    else {
        err_set("unknown command %s", command);
        return ERR;
    }

    l_ASSERT(0);

args_err:
    err_set("%s: incorrect number of arguments", command);
    return ERR;
state_err:
    err_set("%s: not allowed in %s state", command, state_str());
    return ERR;
}

/* Print the command prompt. */
void cli_prompt(void)
{
    pr("lambda:%5s> ", state_str());
    fflush(stdout);
}
```

lcore/errors.c

```
/* $Id: errors.c,v 1.2 2001/04/20 05:48:27 reid Exp $ */

/* This file contains error handling facilities. Stylistic convention: error
```

```
   messages should not begin with a capital letter nor end with a period. If
   it gets too long the error string will be truncated. Do not think about
   pink elephants. */

#include "lambda.h"
#include "lcore.h"

#include <stdarg.h>
#include <stdio.h>
#include <string.h>


/*** Globals ***/

/* The error string. */
char g_err[ERR_CHARS_MAX] = "harvest the microwave burritos";

/* If true, ignore attempts to change the error string. */
bool ignore = 0;


/*** Private prototypes ***/

void err_prepend_nf(char * s);


/*** Public functions ***/

/* Return a pointer to the error string. It is safe to pass the pointer
   returned to other err_*() functions. */
char * err_get(void)
{
    return g_err;
}

/* Return a pointer to a freshly allocated copy of the error string. */
char * err_copy(void)
{
    char * s;

    s = l_malloc(strlen(err_get()) + 1);
    strcpy(s, err_get());

    return s;
}

/* Expand fmt and following arguments according to printf() rules; set the
   error string to the result. */
void err_set(char * fmt, ...)
{
```

```
    va_list ap;
    char * s;

    if (ignore)
        return;

    va_start(ap, fmt);
    vasprintf(&s, fmt, ap);
    g_err[0] = '\0';
    err_prepend_nf(s);
    l_free(s);
    va_end(ap);
}


/* Expand fmt and following arguments according to printf() rules; prepend the
   result to the error string. */
void err_prepend(char * fmt, ...)
{
    va_list ap;
    char * s;

    if (ignore)
        return;

    va_start(ap, fmt);
    vasprintf(&s, fmt, ap);
    err_prepend_nf(s);
    l_free(s);
    va_end(ap);
}

/* Ignore any attempts to change the error string until err_ignore_off() is
   called. */
void err_ignore_on(void)
{
    ignore = 1;
}

/* Stop ignoring attempts to change the error string. */
void err_ignore_off(void)
{
    ignore = 0;
}


/*** Private functions ***/

/* Prepend s to the error string. */
void err_prepend_nf(char * s)
```

```
{
    size_t delta, count;

    delta = min((int)strlen(s), ERR_CHARS_MAX);
    count = min((int)strlen(g_err) + 1, (int)(ERR_CHARS_MAX - delta - 1));
    memmove(g_err + delta, g_err, count);
    memmove(g_err, s, delta);
    g_err[ERR_CHARS_MAX - 1] = '\0';

    l_ASSERT(g_err[delta + count - 1] == '\0');
}
```

lcore/l_assert.c

```
/* $Id: l_assert.c,v 1.3 2001/04/20 05:48:27 reid Exp $ */

/* This file implements the lambda assertion backend function, _l_assert().
   The l_assert() macro is defined in ‘lambda.h’. */

#include "lambda.h"
#include "lcore.h"


/* Print an "assertion failed" message with some information about reporting
   the potential bug and abort. */
void _l_assert(char * file, int line)
{
    error_pr("\n--\nAssertion failed: %s, line %d\n", file, line);
    error_pr("
The program has failed one of its internal sanity checks. This indicates a
bug; please report it by following the instructions in the file README which
came with the distribution package. These instructions are also available on
the lambda website, <http://reidster.net/lambda/>.\n\n");
    abort();
}
```

lcore/memory.c

```
/* $Id: memory.c,v 1.4 2001/04/20 05:48:27 reid Exp $ */

/* This file implements lambda's memory management functions. Memory
   allocation and deallocation should be avoided in the go state (because they
   might cause unpredictable delays). */

/* FIXME: add a memabort() function that prints an out-of-memory error message
   (citing the offending function) and aborts. */

#include "lambda.h"
```

```
#include "lcore.h"

#include <stdlib.h>


/*** Public functions ***/

/* Like malloc(), but cannot fail. Abort the program on failure. */
void * l_malloc(size_t size)
{
    void * p;

    p = malloc(size);
    if (p == NULL) {
        error_pr("\n--\nmalloc(%u) failed\n", size);
        error_pr("
This means FIXME.\n");
        abort();
    }

    return p;
}

/* Like realloc(), but cannot fail. Abort the program on failure. */
void * l_realloc(void * p, size_t new_size)
{
    void * new_p;

    new_p = realloc(p, new_size);
    if (new_p == NULL) {
        error_pr("\n--\nrealloc(%p, %u) failed\n", p, new_size);
        error_pr("
This means FIXME.\n");
        abort();
    }

    return new_p;
}

/* Like free(). */
void l_free(void * p)
{
    free(p);
}
```

lcore/modules.c

```
/* $Id: modules.c,v 1.6 2001/04/20 05:48:27 reid Exp $ */
```

```
/* This file contains the module loading and unloading functions.

   FIXME: (non)portability note: currently we use the linux dynamic loading
   functions directly. There ought to be a wrapper around that stuff. */

#include "lambda.h"
#include "lcore.h"

#include <dlfcn.h>
#include <stdio.h>
#include <string.h>


/*** Public functions ***/

/* Find and load the module named name and return a pointer to it. Return NULL
   and set the error string if an error occurred. */
module * load_module(char * name)
{
    module * m;
    char * s;
    unsigned * magic;

    m = l_malloc(sizeof(module));
    /* FIXME: implement env variable LAMBDA_MOD_PATH */
    s = l_malloc(256);
    sprintf(s, "/home/reid/lambda/modules/%s.lm", name);
    m->handle = dlopen(s, RTLD_NOW);
    l_free(s);
    if (m->handle == NULL) {
        err_set("can't load module %s: %s", name, dlerror());
        goto error;
    }
    magic = dlsym(m->handle, "magic");
    if ((s = dlerror()) != NULL) {
        err_set("module %s is broken: missing object `magic'", name);
        goto error;
    }
    if (*magic != MOD_MAGIC_1_0_0) {
        err_set("module %s is broken: bad magic number %x (expected %x)",
                  name, *magic, MOD_MAGIC_1_0_0);
        goto error;
    }
    m->name = dlsym(m->handle, "name");
    if ((s = dlerror()) != NULL) {
        err_set("module %s is broken: missing object `name'", name);
        goto error;
    }
    m->version = dlsym(m->handle, "version");
```

```
    if ((s = dlerror()) != NULL) {
        err_set("module %s is broken: missing object ‘version’", name);
        goto error;
    }
    m->desc = dlsym(m->handle, "description");
    if ((s = dlerror()) != NULL) {
        err_set("module %s is broken: missing object ‘description’", name);
        goto error;
    }
    m->decls = dlsym(m->handle, "port_decls");
    if ((s = dlerror()) != NULL) {
        err_set("module %s is broken: missing object ‘port_decls’", name);
        goto error;
    }
    m->go = dlsym(m->handle, "lm_go");
    if ((s = dlerror()) != NULL) {
        err_set("module %s is broken: missing object ‘lm_go’", name);
        goto error;
    }
    m->pause = dlsym(m->handle, "lm_pause");
    if ((s = dlerror()) != NULL) {
        err_set("module %s is broken: missing object ‘lm_pause’", name);
        goto error;
    }
    m->stop = dlsym(m->handle, "lm_stop");
    if ((s = dlerror()) != NULL) {
        err_set("module %s is broken: missing object ‘lm_stop’", name);
        goto error;
    }
    m->tick = dlsym(m->handle, "lm_tick");
    if ((s = dlerror()) != NULL) {
        err_set("module %s is broken: missing object ‘lm_tick’", name);
        goto error;
    }

    return m;

error:
    l_free(m);
    return NULL;
}

/* Unload the module mod. Postcondition: mod points to garbage.*/
void unload_module(module * mod)
{
    dlclose(mod->handle);
    l_free(mod);
}
```

lcore/osc.c

```
/* $Id: osc.c,v 1.2 2001/04/20 05:48:27 reid Exp $ */

#include "lambda.h"
#include "lcore.h"

#include <math.h>


/*** Public functions ***/

/* Allocate a new osc object, with specified duty cycle and starting phase.
   Duty cycle must be less than 1.0. */
osc * osc_new(double phase, double duty)
{
    osc * os;

    l_ASSERT(duty < 1.0);

    os = l_malloc(sizeof(osc));
    os->phase = phase;
    os->duty = duty;

    return os;
}

/* Deallocate the osc object os. */
void osc_free(osc * os)
{
    l_free(os);
}

/* The following functions are oscillators of various waveforms. For each, the
   operation is: Produce count samples using the appropriate waveform at
   frequency f (frequencies which are zero or negative are allowed), storing
   them in buf. Keep state using oscillator os, and it is guaranteed that n
   samples generated with these functions are identical whether they were
   generated by one call or several in turn. It is the caller's responsibility
   to ensure that buf is large enough. */

/* Sine wave oscillator. */
void osc_sine(sample * buf, osc * os, freq f, unsigned count)
{
    unsigned i;
    double step;

    step = seconds_per_sample() * f * 2 * M_PI;
    for (i = 0; i < count; i++) {
        buf[i] = sin(os->phase);
```

```
        os->phase += step;
        if (os->phase > 2 * M_PI)
            os->phase -= 2 * M_PI;
        else if (os->phase < -2 * M_PI)
            os->phase += 2 * M_PI;
    }
}
```

lcore/ports.c

```
/* $Id: ports.c,v 1.4 2001/04/20 05:48:27 reid Exp $ */

/* This file implements the data movement functions.

   FIXME: currently a l_send(), l_recv() pair is a two-copy operation.
   Investigate whether reducing it to a one or zero copy operation would be
   worth the decrease in stability. If the copy in l_send() were removed, then
   l_recv() would be accessing module private memory; if the copy in l_recv()
   were also removed, then modules would be accessing each others' private
   memory. */

#include "lambda.h"
#include "lcore.h"

#include <string.h> /* for memcpy() */


/*** Public functions ***/

/* Send the data stored in buf on port p. If the type of the data in buf is
   not described by p->type, the results are undefined. */
void l_send(port * p, void * buf)
{
    port_VERIFY(p);
    l_ASSERT(p->direction == OUT);

    memcpy(p->contents, buf, l_sizeof(p->type));
}

/* Copy the data waiting on port p to local buffer buf. It is the caller's
   responsibility to ensure that buf is large enough; if it is not the results
   are undefined. */
void l_recv(port * p, void * buf)
{
    port_VERIFY(p);
    l_ASSERT(p->direction == IN);

    memcpy(buf, p->contents, l_sizeof(p->type));
```

```
}

/* If there is a port named name in ports (whose size is n_ports), return its
   address; otherwise, return NULL. */
port * port_addrof(char * name, unsigned n_ports, port * ports[])
{
    unsigned i;

    for (i = 0; i < n_ports; i++)
        if (!strcmp(name, ports[i]->name)) {
            port_VERIFY(ports[i]);
            return ports[i];
        }

    return NULL;
}

/* Allocate and initialize a new port according to its declaration decl and
   parent node parent; return a pointer to it. Cannot fail. */
port * port_new(struct awid_struct * parent, port_decl * decl)
{
    port * p;

    p = l_malloc(sizeof(port));
    p->name = decl->name;
    p->parent = parent;
    p->direction = decl->direction;
    p->type = ltyp_new(decl->meta_type, decl->raw_type, decl->n_elements);
    p->strict = decl->strict;
    p->deps = decl->depends;
    p->refcount = 0;
    p->source = NULL;
    p->contents = NULL;

    port_VERIFY(p);
    return p;
}

/* Free a port. Postcondition: p points to garbage. */
void port_free(port * p)
{
    port_VERIFY(p);
    ltyp_free(p->type);
    l_free(p);
}

/* Return 1 if p is ready for data transfer (i.e. connected or set); 0
   otherwise. */
bool port_ready(port * p)
{
```

```
    port_VERIFY(p);

    if (p->direction == IN)
        return (bool) p->contents;
    else
        return (bool) p->refcount;
}

/* Check that a port is internally consistent. If not, abort. Only works if
   assertions are enabled. */
void _port_verify(port * p)
{
    l_ASSERT(p != NULL);
    l_ASSERT(p->strict == STRICT || p->strict == NONSTRICT);
    if (p->direction == IN) {
        l_ASSERT(p->refcount == 0);
        if (p->source != NULL) {
            l_ASSERT(p->contents != NULL);
            l_ASSERT(p->source->contents != NULL);
            l_ASSERT(ltyp_eq(p->type, p->source->type));
        }
    } else if (p->direction == OUT) {
        l_ASSERT(p->source == NULL);
        if (p->refcount > 0)
            l_ASSERT(p->contents != NULL);
    } else
        l_ASSERT(0);
}
```

lcore/states.c

```
/* $Id: states.c,v 1.3 2001/04/20 05:48:27 reid Exp $ */

/* This file contains state management stuff and do_tick(). */

#include "lambda.h"
#include "lcore.h"


/*** Globals ***/

extern arena g_arena;


/*** Public functions ***/

/* Execute one tick's worth of calculations. If an error occurs, set the error
   string and return ERR (in this case, the arena is in an inconsistent state
   and should be stopped immediately); return OK on success. This function is
```

```
   only appropriate in the go state. */
lerr do_tick(void)
{
    awid * cur = g_arena.first;

    l_ASSERT(state() == GO);

    while (cur != NULL) {
        if ((* cur->mod->tick)(cur->context) != OK) {
            err_prepend("%s: ", cur->name);
            return ERR;
        }
        cur = cur->next;
    }

    return OK;
}

/* Return the current state. */
state_t state(void)
{
    return g_arena.state;
}

/* Return a string describing the current state. */
char * state_str(void)
{
    switch (state()) {
    case GO:
        return "go";
        break;
    case PAUSE:
        return "pause";
        break;
    case STOP:
        return "stop";
        break;
    default:
        l_ASSERT(0);
    }

    l_ASSERT(0);
    return NULL;
}

/* Attempt to enter the go state. On success, return OK; on failure, set the
   error string and return ERR. */
lerr enter_go(void)
{
    awid * cur;
```

```
    char * s;

    if (g_arena.dirty) {
        err_set("arena is dirty");
        return ERR;
    }

    if (state() == STOP) {
        cur = g_arena.first;
        while (cur != NULL) {
            if((* cur->mod->go)(&(cur->context), cur->n_ports, cur->ports)
               != OK) {
                err_prepend("go failed: %s: ", cur->name);
                /* set all widgets already in go state back to stop, saving
                   error string in case going back to stop fails too. */
                s = err_copy();
                if (enter_stop(cur) != OK)
                    err_prepend("%s\n", s);
                l_free(s);
                return ERR;
            }
            cur = cur->next;
        }
        timing_stopgo();
    } else {
        l_ASSERT(state() == PAUSE);
        timing_pausego();
    }

    g_arena.state = GO;
    return OK;
}

/* Enter the pause state. On success, return OK; on error, set the error
   string and return ERR, but still go to the pause state. Try to set all
   widgets to pause even if some fail. */
lerr enter_pause(void)
{
    awid * cur;
    char ** err_list = NULL;
    int n_errs = 0;

    l_ASSERT(state() == GO);

    cur = g_arena.first;
    while (cur != NULL) {
        if((* cur->mod->pause)(cur->context) != OK) {
            n_errs++;
            err_prepend("%s: ", cur->name);
            err_list = l_realloc(err_list, n_errs * sizeof(char *));
```

```
            err_list[n_errs - 1] = err_copy();
        }
        cur = cur->next;
    }

    g_arena.state = PAUSE;

    if (n_errs == 0)
        return OK;
    else {
        err_set("");
        while (n_errs > 0) {
            err_prepend("\n  %s", err_list[n_errs - 1]);
            l_free(err_list[n_errs - 1]);
            n_errs--;
        }
        err_prepend("pause failed:");
        l_free(err_list);
        return ERR;
    }
}

/* Enter the stop state. If limit != NULL, only set widgets before limit to
   stop. Try to set all appropriate widgets to stop even if some fail. On
   success, return OK; on failure, set the error string and return ERR, but
   still enter the stop state. */
lerr enter_stop(awid * limit)
{
    awid * cur;
    char ** err_list = NULL;
    int n_errs = 0;

    cur = g_arena.first;
    while (cur != limit) {
        l_ASSERT(cur != NULL);
        if((* cur->mod->stop)(cur->context) != OK) {
            n_errs++;
            err_prepend("%s: ", cur->name);
            err_list = l_realloc(err_list, n_errs * sizeof(char *));
            err_list[n_errs - 1] = err_copy();
        }
        cur = cur->next;
    }

    g_arena.state = STOP;

    if (n_errs == 0)
        return OK;
    else {
        err_set("");
```

```
        while (n_errs > 0) {
            err_prepend("\n  %s", err_list[n_errs - 1]);
            l_free(err_list[n_errs - 1]);
            n_errs--;
        }
        err_prepend("stop failed:");
        l_free(err_list);
        return ERR;
    }
}
```

lcore/term_output.c

```
/* $Id: term_output.c,v 1.3 2001/04/20 05:48:27 reid Exp $ */

/* This file implements the terminal output functions. */

#include "lambda.h"
#include "lcore.h"

#include <stdarg.h>
#include <stdio.h>


/*** Globals ***/

/* If true, print debugging output.

   FIXME: main.c shouldn't have to mess with this variable. Add a proper
   term_output_init() function. */
bool g_print_debug;


/*** Public functions ***/

/* Works just like printf(), except nothing is returned -- fmt is a format
   string according to printf() rules and appropriate arguments follow; the
   result is printed on standard output. */
void pr(char * fmt, ...)
{
    va_list ap;

    va_start(ap, fmt);
    vfprintf(stdout, fmt, ap);
    va_end(ap);
}

/* Like pr(), but text goes to stderr instead of stdout. */
```

```
void error_pr(char * fmt, ...)
{
    va_list ap;

    /* fprintf(stderr, "lambda: "); */
    va_start(ap, fmt);
    vfprintf(stderr, fmt, ap);
    va_end(ap);
}

/* Like pr(), but text is printed conditionally depending on whether debugging
   output has been activated. */
void debug_pr(char * fmt, ...)
{
    va_list ap;

    if (g_print_debug) {
        va_start(ap, fmt);
        vfprintf(stdout, fmt, ap);
        va_end(ap);
    }
}
```

lcore/timing.c

```
/* $Id: timing.c,v 1.7 2001/04/20 05:48:27 reid Exp $ */

/* This file contains the time management functions. */

#include "lambda.h"
#include "lcore.h"

#include <sys/time.h>
#include <time.h>


/*** Constants ***/

#define ONE_MILLION 1000000


/*** Globals ***/

/* Sample rate in hertz. */
freq sample_rate;

/* Number of samples per tick. */
unsigned tick_len;
```

```
/* If true, don't calculate faster than real time. */
bool limit_realtime;

/* Number of samples calculated since the last stop->go transition, up to the
   end of the last tick. */
uint64 samples_elapsed;

/* Number of samples calculated since the last transition into the go state,
   up to the end of the last tick. */
uint64 go_samples;

/* Time spent calculating since the last go transition into the go state, up
   to the end of the last tick. */
uint64 go_usecs;

/* Wall time of the last transition into the go state. */
uint64 go_entry_wall;

/* Wall time of the beginning of the last tick. */
uint64 tick_start_wall;


/*** Public functions ***/

/* Return the number of microseconds since the epoch (a specific time in the
   past which is unchanged for the life of the program). This unsigned 64-bit
   quantity rolls over after about 585,000 years. Whether or not that is a bug
   is an exercise for the reader; this program has no provisions for 64-bit
   variables rolling over. */
uint64 usecs_wall(void)
{
    struct timeval tv;

    gettimeofday(&tv, NULL); /* this call cannot fail */
    return tv.tv_sec * (uint64)ONE_MILLION + tv.tv_usec;
}

/* Try to sleep for usecs microseconds. The time slept may end up being longer
   or shorter, but it tries. If usecs is less than USLEEP_BUSYWAIT_USECS,
   busy-wait; otherwise, use nanosleep(). */
void l_usleep(uint64 usecs)
{
    uint64 target;
    struct timespec ts;
    int result;

    if (usecs < USLEEP_BUSYWAIT_USECS) {
        target = usecs_wall() + usecs;
        while (usecs_wall() < target)
            /* nothing */ ;
```

```
    } else {
        ts.tv_sec = usecs / ONE_MILLION;
        ts.tv_nsec = (usecs % ONE_MILLION) * 1000;
        result = nanosleep(&ts, NULL);
        l_ASSERT(result == 0);
    }
}

/* Initialize some globals. FIXME: validate numbers. */
void timing_init(freq sr, unsigned tl, bool limit)
{
    sample_rate = sr;
    tick_len = tl;
    limit_realtime = limit;
}

/* Called on pause->go transition; set variables appropriately. */
void timing_pausego(void)
{
    go_samples = 0;
    go_usecs = 0;
    go_entry_wall = usecs_wall();
}

/* Called on stop->go transition; set variables appropriately. */
void timing_stopgo(void)
{
    samples_elapsed = 0;

    timing_pausego();
}

/* Called on start of a tick; set up some timing stuff. */
void timing_tick_start(void)
{
    tick_start_wall = usecs_wall();
}

/* Called on end of a tick. More timing stuff. If realtime limiting is on and
   we're ahead of schedule, wait until we were supposed to finish this tick
   (leaving some slush time). */
void timing_tick_end(void)
{
    uint64 allowed, actual;

    go_usecs += usecs_wall() - tick_start_wall;
    go_samples += samples_per_tick();
    samples_elapsed += samples_per_tick();

    if (limit_realtime) {
```

```
        allowed = go_samples * seconds_per_sample() * ONE_MILLION;
        actual = usecs_wall() - go_entry_wall + SLUSH_USECS;

        if (allowed > actual)
            l_usleep(allowed - actual);
    }
}

/* Print some statistics about time. FIXME: printf specifiers may be
   non_portable. */
void timing_stats_pr(void)
{
    pr("Since entry of go state from stop:\n");
    pr("  Samples calculated:        %llu\n", samples_elapsed);
    pr("Since entry of go state:\n");
    pr("  Samples calculated:        %llu\n", go_samples);
    pr("  Seconds spent calculating: %5.3f\n",
       (double) go_usecs / ONE_MILLION);
    pr("  Total seconds elapsed:     %5.3f\n",
       (double) (usecs_wall() - go_entry_wall) / ONE_MILLION);
    pr("  Calculation load:          %4.2f\n",
       (double) go_usecs / (go_samples * seconds_per_sample() * ONE_MILLION));
}

inline freq samples_per_second(void)
{
    return sample_rate;
}

inline dur seconds_per_sample(void)
{
    return (dur) 1.0 / sample_rate;
}

inline unsigned samples_per_tick(void)
{
    return tick_len;
}

inline unsigned samples_calc(void)
{
    return samples_elapsed;
}

inline dur seconds_calc(void)
{
    return (dur) samples_elapsed / sample_rate;
}
```

lcore/types.c

```
/* $Id: types.c,v 1.5 2001/04/20 05:48:27 reid Exp $ */

/* This file implements the type management functions. */

#include "lambda.h"
#include "lcore.h"

#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>


/*** Types ***/

/* These variables store the sizes and human-readable names of each of the
   base types and can be indexed by the names of the raw types; e.g.,
   _l_sizeof[LTYP_INT] == sizeof(int). This depends on the fact that arrays
   start at zero and by default enums start at zero as well and increment by
   one for each item. Hence ORDER IS VERY IMPORTANT and must match exactly
   that of the _ltyp enum. */
size_t _l_sizeof[] = {
    sizeof(bool),
    sizeof(byte),
    sizeof(int),
    sizeof(long),
    sizeof(int16),
    sizeof(uint16),
    sizeof(int32),
    sizeof(uint32),
    sizeof(int64),
    sizeof(uint64),
    sizeof(float),
    sizeof(double),
    sizeof(sample),
    sizeof(freq),
    sizeof(dur),
    sizeof(level)
};
char * _ltyp_str[] = {
    "bool",
    "byte",
    "int",
    "long",
    "int16",
    "uint16",
    "int32",
    "uint32",
```

```
    "int64",
    "uint64",
    "float",
    "double",
    "sample",
    "freq",
    "dur",
    "level"
};

/*** Public functions ***/

/* Return the number of bytes required to store an object described by t. */
size_t l_sizeof(ltyp * t)
{
    size_t sz;

    sz = _l_sizeof[t->raw_type];

    switch (t->meta_type) {
    case SCALAR:
        return sz;
        break;
    case ARRAY:
        return sz * t->n_elements;
        break;
    case ARRAY_REL:
        return sz * t->n_elements * samples_per_tick();
        break;
    }

    l_ASSERT(0);
    return 0;
}

/* Allocate and initialize a new ltyp object, and return a pointer to it.
   Cannot fail. */
ltyp * ltyp_new(meta_ltyp meta_type, raw_ltyp raw_type, unsigned n_elements)
{
    ltyp * t;

    l_ASSERT(meta_type >= SCALAR && meta_type <= ARRAY_REL);
    l_ASSERT(!(meta_type == ARRAY && n_elements == 0));
    l_ASSERT(!(meta_type == ARRAY_REL && n_elements == 0));
    l_ASSERT(raw_type >= LTYP_INT && raw_type <= LTYP_LEVEL);

    t = l_malloc(sizeof(ltyp));
    t->meta_type = meta_type;
    t->raw_type = raw_type;
    t->n_elements = n_elements;
```

```
    return t;
}

/* Free t; postcondition: t points to garbage. */
void ltyp_free(ltyp * t)
{
    l_free(t);
}

/* Return 1 if t1 and t2 are equivalent, 0 otherwise. */
bool ltyp_eq(ltyp * t1, ltyp * t2)
{
    if (   t1->meta_type == t2->meta_type
        && t1->raw_type == t2->raw_type
        && t1->n_elements == t2->n_elements)
        return 1;
    else
        return 0;
}

/* Create a human-readable string representation of t (e.g. "int"), and store
   it in freshly allocated memory (it is the caller's responsibility to free
   the memory). Return a pointer to it. Cannot fail. */
char * ltyp_str(ltyp * t)
{
    char * str, * subscript;

    if (t->meta_type == ARRAY || t->meta_type == ARRAY_REL) {
        /* check out the fab ternary operator */
        asprintf(&subscript, "[%u%s]", t->n_elements,
                 (t->meta_type == ARRAY_REL) ? "r" : "");
        if (subscript == NULL) {
            error_pr("asprintf() failed. This means FIXME.\n");
            abort();
        }
    } else {
        subscript = "";
    }

    asprintf(&str, "%s%s", _ltyp_str[t->raw_type], subscript);
    if (str == NULL) {
        error_pr("asprintf() failed. This means FIXME.\n");
        abort();
    }
    if (strcmp(subscript, ""))
        l_free(subscript);

    return str;
}
```

```
/* Create an appropriate string representation of *value according to t, and
   store it in freshly allocated memory (it is the caller's responsibility to
   free the memory). Return a pointer to it. Cannot fail.

   FIXME: sample, freq, dur, and level should be printed more elegantly. */
char * ltyp_tostr(ltyp * t, void * value)
{
    char * str;

    switch (t->raw_type) {
    case LTYP_BOOL:
        asprintf(&str, "%d", *((bool *) value));
        break;
    case LTYP_BYTE:
        asprintf(&str, "%u", *((byte *) value));
        break;
    case LTYP_INT:
        asprintf(&str, "%d", *((int *) value));
        break;
    case LTYP_LONG:
        asprintf(&str, "%ld", *((long *) value));
        break;
    case LTYP_INT16:
        asprintf(&str, "%d", *((int16 *) value));
        break;
    case LTYP_UINT16:
        asprintf(&str, "%u", *((uint16 *) value));
        break;
    case LTYP_INT32:
        asprintf(&str, "%ld", (long) *((int32 *) value));
        break;
    case LTYP_UINT32:
        asprintf(&str, "%lu", (unsigned long) *((uint32 *) value));
        break;
    case LTYP_INT64:
    case LTYP_UINT64:
        /* FIXME */
        l_ASSERT(0);
        break;
    case LTYP_FLOAT:
        asprintf(&str, "%g", *((float *) value));
        break;
    case LTYP_DOUBLE:
        asprintf(&str, "%g", *((double *) value));
        break;
    case LTYP_SAMPLE:
        asprintf(&str, "%g", *((sample *) value));
        break;
    case LTYP_FREQ:
```

```
        asprintf(&str, "%g", *((freq *) value));
        break;
    case LTYP_DUR:
        asprintf(&str, "%g", *((dur *) value));
        break;
    case LTYP_LEVEL:
        asprintf(&str, "%g", *((level *) value));
        break;
    default:
        l_ASSERT(0);
    }

    if (str == NULL) {
        error_pr("asprintf() failed. This means FIXME.\n");
        abort();
    }

    return str;
}

/* Interpret str appropriately according to t; store the result in buf. If buf
   is too small to hold a value of the type described by t, the results are
   undefined. On success, return OK; otherwise, set the error string, do not
   modify the contents of buf, and return ERR.

   FIXME: may as well self-allocate memory, since ltyp_tostr() does it.
   FIXME: interpret arrays too.
   FIXME: do range checking. */
lerr ltyp_fromstr(void * buf, ltyp * t, char * str)
{
    byte val[L_SIZEOF_MAX];
    char * tailptr;

    if (t->meta_type != SCALAR) {
        err_set("ltyp_fromstr() cannot translate arrays");
        return ERR;
    }

    errno = 0;
    switch (t->raw_type) {
    case LTYP_BOOL:
        *((bool *) val) = (bool) strtol(str, &tailptr, 0);
        if (*((bool *) val) != 0 && *((bool *) val) != 1) {
            err_set("boolean value must be 1 or 0");
            return ERR;
        }
        break;
    case LTYP_BYTE:
        *((byte *) val) = (byte) strtoul(str, &tailptr, 0);
        break;
```

```
case LTYP_INT:
    *((int *) val) = (int) strtol(str, &tailptr, 0);
    break;
case LTYP_LONG:
    *((long *) val) = (long) strtol(str, &tailptr, 0);
    break;
case LTYP_INT16:
    *((int16 *) val) = (int16) strtol(str, &tailptr, 0);
    break;
case LTYP_UINT16:
    *((uint16 *) val) = (uint16) strtoul(str, &tailptr, 0);
    break;
case LTYP_INT32:
    *((int32 *) val) = (int32) strtol(str, &tailptr, 0);
    break;
case LTYP_UINT32:
    *((uint32 *) val) = (uint32) strtoul(str, &tailptr, 0);
    break;
case LTYP_INT64:
case LTYP_UINT64:
    /* FIXME */
    l_ASSERT(0);
    break;
case LTYP_FLOAT:
    *((float *) val) = (float) strtod(str, &tailptr);
    break;
case LTYP_DOUBLE:
    *((double *) val) = (double) strtod(str, &tailptr);
    break;
case LTYP_SAMPLE:
    *((sample *) val) = (sample) strtod(str, &tailptr);
    break;
case LTYP_FREQ:
    *((freq *) val) = (freq) strtod(str, &tailptr);
    break;
case LTYP_DUR:
    *((dur *) val) = (dur) strtod(str, &tailptr);
    break;
case LTYP_LEVEL:
    *((level *) val) = (level) strtod(str, &tailptr);
    break;
default:
    l_ASSERT(0);
}

if (errno != 0) {
    err_set("%s", strerror(errno));
    return ERR;
}
```

```
    if (*tailptr != '\0') {
        err_set("junk at end of string");
        return ERR;
    }

    memcpy(buf, val, l_sizeof(t));
    return OK;
}
```

lcore/util.c

```
/* $Id: util.c,v 1.4 2001/04/20 05:48:27 reid Exp $ */

/* This file implements various utility functions. */

#include "lambda.h"
#include "lcore.h"

#include <errno.h>
#include <stdio.h>
#include <string.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>


/*** Public functions ***/

/* Return the minimum of a and b. */
inline int min(int a, int b)
{
    return (a < b) ? a : b;
}

/* Return 1 if reading from fp would result in an end-of-file condition, 0
   otherwise. */
bool l_feof(FILE * fp)
{
    fd_set s;
    struct timeval timeout = { 0, 0 };
    int ready;

    FD_ZERO(&s);
    FD_SET(fileno(fp), &s);

    l_ASSERT(FD_ISSET(fileno(fp), &s));

    ready = select(FD_SETSIZE, &s, NULL, NULL, &timeout);
    /* FIXME: handle ENOMEM separately. */
```

```
    l_ASSERT(ready != -1);

    return (bool) !ready;
}
```

modules/io-oss.c

```
/* $Id: io-oss.c,v 1.6 2001/04/20 05:48:27 reid Exp $ */

/* Reference: Open Sound System Programmer's Guide, v1.11, 2000/11/07
              http://www.opensound.com/pguide/oss.pdf */

/* FIXME:

   - Set better fragment size.
   - Implement proper full duplex.
   - Other real time stuff.
   - Optimize for speed.
   - Provide for device file selection at runtime. */

#include "lambda.h"

#include <errno.h>
#include <fcntl.h>
#include <limits.h>
#include <string.h>
#include <sys/ioctl.h>
#include <sys/soundcard.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>


/*** Constants ***/

/* which device file to use? */
#define OSS_DEVICE "/dev/dsp"


/*** Identification ***/

unsigned magic = MOD_MAGIC_1_0_0;
char name[] = "io-oss";
char version[] = "1.0d1";
char description[] = "Plays and records sound via the OSS drivers.";


/*** Ports ***/
```

```
port_decl port_decls[] = {
    { "play-right",   IN, ARRAY_REL, LTYP_SAMPLE, 1, NONSTRICT,
                      { } },
    { "play-left",    IN, ARRAY_REL, LTYP_SAMPLE, 1, NONSTRICT,
                      { "play-right" } },
    { "record-right", OUT,  ARRAY_REL, LTYP_SAMPLE, 1, NONSTRICT,
                      { } },
    { "record-left",  OUT,  ARRAY_REL, LTYP_SAMPLE, 1, NONSTRICT,
                      { "record-right" } },
    END_PORT_DECLARATIONS
};


/*** Types ***/

typedef struct {
    port * play_r, * play_l;
    port * rec_r, * rec_l;
    int fd;
    bool play_active, play_stereo, rec_active, rec_stereo;
    int16 * buf_oss;
    sample * buf_r, * buf_l;
} context_t;


/*** Private prototypes ***/

lerr config_dsp(context_t * x);


/*** Public functions ***/

lerr lm_go(void ** context, unsigned n_ports, port * ports[])
{
    context_t * x;

    x = l_malloc(sizeof(context_t));
    x->play_active = 0;
    x->play_stereo = 0;
    x->rec_active = 0;
    x->rec_stereo = 0;

    x->play_r = port_addrof("play-right", n_ports, ports);
    l_ASSERT(x->play_r != NULL && x->play_r->direction == IN);
    if (port_ready(x->play_r))
        x->play_active = 1;
    x->play_l = port_addrof("play-left", n_ports, ports);
    l_ASSERT(x->play_l != NULL && x->play_l->direction == IN);
    if (port_ready(x->play_l))
        x->play_stereo = 1;
```

```
    x->rec_r = port_addrof("record-right", n_ports, ports);
    l_ASSERT(x->rec_r != NULL && x->rec_r->direction == OUT);
    if (port_ready(x->rec_r))
        x->rec_active = 1;
    x->rec_l = port_addrof("record-left", n_ports, ports);
    l_ASSERT(x->rec_l != NULL && x->rec_r->direction == OUT);
    if (port_ready(x->rec_l))
        x->rec_stereo = 1;

    l_ASSERT(x->play_active || x->rec_active);

    if (config_dsp(x) != OK) {
        /* config_dsp will have already set the error string */
        l_free(x);
        return ERR;
    }

    x->buf_r = l_malloc(sizeof(sample) * samples_per_tick());
    x->buf_oss = l_malloc(sizeof(int16) * samples_per_tick());
    if (x->play_stereo || x->rec_stereo) {
        x->buf_l = l_malloc(sizeof(sample) * samples_per_tick());
        x->buf_oss = l_realloc(x->buf_oss,
                               2 * sizeof(int16) * samples_per_tick());
    }

    *context = x;

    return OK;
}

lerr lm_pause(void * context)
{
    context_t * x = context;

    if (ioctl(x->fd, SNDCTL_DSP_POST) == -1) {
        err_set("pause failed for %s: %s", OSS_DEVICE, strerror(errno));
        return ERR;
    }

    return OK;
}

lerr lm_stop(void * context)
{
    context_t * x = context;

    close(x->fd);
    l_free(x->buf_oss);
    l_free(x->buf_r);
```

```
    if (x->play_stereo || x->rec_stereo)
        l_free(x->buf_l);
    l_free(x);

    return OK;
}

lerr lm_tick(void * context)
{
    context_t * x = context;
    int i, offset;

    /* In the loops below, conversions from sample to int16 are implicit. */

    if (x->play_active) {
        l_recv(x->play_r, x->buf_r);
        if (x->play_stereo)
            l_recv(x->play_l, x->buf_l);

        offset = 0;
        for (i = 0; i < samples_per_tick(); i++) {
            /* left sample comes first */
            if (x->play_stereo) {
                x->buf_oss[offset] = x->buf_l[i] * SHRT_MAX;
                offset++;
            }

            x->buf_oss[offset] = x->buf_r[i] * SHRT_MAX;
            offset++;
        }

        /* FIXME: handle errors */
        write(x->fd, x->buf_oss,
              sizeof(int16) * samples_per_tick() * (x->play_stereo ? 2 : 1));
    }

    /* FIXME: recording not implemented! */

    return OK;
}


/*** Private functions ***/

/* Configure the sound card. */
lerr config_dsp(context_t * x)
{
    int i, flags, width, channels, rate;

    /* parameters */
```

```
width = AFMT_S16_NE; /* 16 bits signed, host endian */
channels = x->play_stereo ? 2 : 1;
rate = samples_per_second();
debug_pr("io-oss: configuring %s for 16 bits, %d channels, %d Hz\n",
         OSS_DEVICE, channels, rate);


/* open the audio device */
if (x->play_active && x->rec_active)
    flags = O_RDWR;
else if (x->play_active)
    flags = O_WRONLY;
else
    flags = O_RDONLY;
x->fd = open(OSS_DEVICE, flags);
if (x->fd == -1) {
    err_set("can't open %s: %s", OSS_DEVICE, strerror(errno));
    return ERR;
}
/* set sample width to 16 bits, native endian */
i = width;
if (ioctl(x->fd, SNDCTL_DSP_SETFMT, &i) == -1) {
    err_set("can't set %s to 16 bits: %s", OSS_DEVICE, strerror(errno));
    return ERR;
}
if (i != width) {
    err_set("%s does not support 16 bits", OSS_DEVICE);
    return ERR;
}
/* set mono/stereo as appropriate */
i = channels;
if (ioctl(x->fd, SNDCTL_DSP_CHANNELS, &i) == -1) {
    err_set("can't set %s to %d channels: %s", OSS_DEVICE, channels,
            strerror(errno));
    return ERR;
}
if (i != channels) {
    err_set("%s does not support %d channels", OSS_DEVICE, channels);
    return ERR;
}
/* set sample rate */
i = rate;
if (ioctl(x->fd, SNDCTL_DSP_SPEED, &i) == -1) {
    err_set("can't set %s to %d Hz: %s", OSS_DEVICE, rate,
            strerror(errno));
    return ERR;
}
if (i != rate) {
    err_set("%s does not support %d Hz", OSS_DEVICE, rate);
    return ERR;
```

```
    }

    /* whew */
    return OK;
}
```

modules/silly-sequencer.c

```
/* $Id: silly-sequencer.c,v 1.1 2001/04/20 05:49:01 reid Exp $ */

#include "lambda.h"


/*** Identification ***/

unsigned magic = MOD_MAGIC_1_0_0;
char name[] = "silly-sequencer";
char version[] = "1.0d1";
char description[] = "A toy sequencer.";


/*** Ports ***/

port_decl port_decls[] = {
    { "f-out", OUT, SCALAR, LTYP_FREQ, 0, STRICT, {} },
};


/*** Types ***/

typedef struct {
    port * out;
} context_t;


/*** Score ***/

#define NOTE_DUR 12
#define NOTE_PCT 1.0
#define TRANSPOSE 0.5

#define C  261.6
#define Cs 277.2
#define D  293.7
#define Ds 311.1
#define E  329.6
#define F  349.2
#define Fs 370.0
#define G  392.0
```

```
#define Gs 415.3
#define A  440.0
#define As 466.2
#define B  493.9

#define N1(X) X, X, X, X,  X, X, X, X,  X, X, X, X,  X, X, X, X, \
              X, X, X, X,  X, X, X, X,  X, X, X, X,  X, X, X, X, 0,
#define N2(X) X, X, X, X,  X, X, X, X, \
              X, X, X, X,  X, X, X, X, 0,
#define N4(X) X, X, X, X, \
              X, X, X, X, 0,
#define N8(X) X, X, X, X, 0,
#define N16(X) X, X, 0,
#define N2D(X) X, X, X, X,  X, X, X, X,  X, X, X, X, \
               X, X, X, X,  X, X, X, X,  X, X, X, X, 0,
#define N4D(X) X, X, X, X,  X, X, \
               X, X, X, X,  X, X, 0,
#define N8D(X) X, X, X, X,  X, X, 0,

/* The Star Spangled Banner */
sample score[] = {
N8(G) N8(E) N4(C) N4(E) N4(G) N2(C*2) N8D(E*2) N16(D*2) N4(C*2) N4(E) N4(Fs)
N2(G) N8(G) N8(G) N4D(E*2) N8(D*2) N4(C*2) N2(B) N8(A) N8(B) N4(C*2) N4(C*2)
   N4(G)
N4(E) N4(C) N8D(G) N16(E) N4(C) N4(E) N4(G) N2(C*2) N8D(E*2) N16(D*2)
N4(C*2) N4(E) N4(Fs) N2(G) N8(G) N8(G) N4D(E*2) N8(D*2) N4(C*2)
N2(B) N8(A) N8(B) N4(C*2) N4(C*2) N4(G) N4(E) N4(C) N8(E*2) N8(E*2)
N4(E*2) N4(F*2) N4(G*2) N2(G*2) N8(F*2) N8(E*2) N4(D*2) N4(E*2) N4(F*2)
N2(F*2) N4(F*2) N4D(E*2) N8(D*2) N4(C*2) N2(B) N8(A) N8(B)
N4(C*2) N4(E) N4(Fs) N2(G) N4(G) N4(C*2) N4(C*2) N8(C*2) N8(B)
N4(A) N4(A) N4(A) N4(D*2) N8(F*2) N8(E*2) N8(D*2) N8(C*2) N4(C*2) N2(B) N8(G)
   N8(G)
N4D(C*2) N8(D*2) N8(E*2) N8(F*2) N1(G*2) N8(C*2) N8(D*2) N4D(E*2) N8(F*2)
   N4D(D*2) N2D(C*2)
};


/*** Public functions ***/

lerr lm_go(void ** context, unsigned n_ports, port * ports[])
{
    context_t * x;

    x = l_malloc(sizeof(context_t));
    x->out = port_addrof("f-out", n_ports, ports);
    *context = x;
    return OK;
}

lerr lm_pause(void * context)
```

```
{
    return OK;
}

lerr lm_stop(void * context)
{
    l_free(context);
    return OK;
}

lerr lm_tick(void * context)
{
    context_t * x = context;
    freq note = 0.0;

    if (seconds_calc() * NOTE_DUR < sizeof(score) / sizeof(freq))
        note = score[(int)(seconds_calc() * NOTE_DUR)] * TRANSPOSE;

    l_send(x->out, &note);
    return OK;
}
```

modules/sin.c

```
/* $Id: sin.c,v 1.8 2001/04/20 05:48:27 reid Exp $ */

#include "lambda.h"

#include <math.h>
#include <string.h>


/*** Identification ***/

unsigned magic = MOD_MAGIC_1_0_0;
char name[] = "sin";
char version[] = "1.0d1";
char description[] = "Generates a pure sine wave at the given frequency.";


/*** Port declarations ***/

port_decl port_decls[] = {
    { "out", OUT, ARRAY_REL, LTYP_SAMPLE, 1, STRICT, {} },
    { "f",   IN, SCALAR,    LTYP_FREQ,   0, STRICT, {} },
    END_PORT_DECLARATIONS
};
```

```
/*** Types ***/

typedef struct {
    port * out;
    port * f;
    sample * buf;
    osc * os;
} context_t;


/*** Public functions ***/

lerr lm_go(void ** context, unsigned n_ports, port * ports[])
{
    context_t * x;

    x = l_malloc(sizeof(context_t));

    x->out = port_addrof("out", n_ports, ports);
    l_ASSERT(x->out != NULL
             && x->out->direction == OUT
             && port_ready(x->out));
    x->f = port_addrof("f", n_ports, ports);
    l_ASSERT(x->f != NULL
             && x->f->direction == IN
             && port_ready(x->out));

    x->buf = l_malloc(sizeof(sample) * samples_per_tick());
    x->os = osc_new(0.0, 0.0);

    *context = x;

    return OK;
}

lerr lm_pause(void * context)
{
    return OK;
}

lerr lm_stop(void * context)
{
    context_t * x = context;

    l_free(x->buf);
    osc_free(x->os);
    l_free(x);

    return OK;
}
```

```
lerr lm_tick(void * context)
{
    context_t * x = context;
    freq f;

    l_recv(x->f, &f);
    osc_sine(x->buf, x->os, f, samples_per_tick());
    l_send(x->out, x->buf);

    return OK;
}
```

modules/synth-additive.c

```
/* $Id: synth-additive.c,v 1.3 2001/04/22 18:59:44 reid Exp $ */

#include "lambda.h"


/*** Constants ***/

/* If you change this you must change port_decls also. */
#define HARMONICS 16


/*** Identification ***/

unsigned magic = MOD_MAGIC_1_0_0;
char name[] = "synth-additive";
char version[] = "1.0d1";
char description[] = "Simple additive synthesizer with 16 partials.";


/*** Ports ***/

port_decl port_decls[] = {
    { "out",          OUT, ARRAY_REL, LTYP_SAMPLE, 1, STRICT, {} },
    { "out-volume",   IN,  SCALAR,    LTYP_LEVEL,  0, STRICT, {} },
    { "f",            IN,  SCALAR,    LTYP_FREQ,   0, STRICT, {} },
    { "fund-volume",  IN,  SCALAR,    LTYP_LEVEL,  0, STRICT, {} },
    { "par2-volume",  IN,  SCALAR,    LTYP_LEVEL,  0, NONSTRICT,
      { "fund-volume" } },
    { "par3-volume",  IN,  SCALAR,    LTYP_LEVEL,  0, NONSTRICT,
      { "par2-volume" } },
    { "par4-volume",  IN,  SCALAR,    LTYP_LEVEL,  0, NONSTRICT,
      { "par3-volume" } },
    { "par5-volume",  IN,  SCALAR,    LTYP_LEVEL,  0, NONSTRICT,
      { "par4-volume" } },
```

```
    { "par6-volume",  IN,  SCALAR,    LTYP_LEVEL,  0, NONSTRICT,
      { "par5-volume" } },
    { "par7-volume",  IN,  SCALAR,    LTYP_LEVEL,  0, NONSTRICT,
      { "par6-volume" } },
    { "par8-volume",  IN,  SCALAR,    LTYP_LEVEL,  0, NONSTRICT,
      { "par7-volume" } },
    { "par9-volume",  IN,  SCALAR,    LTYP_LEVEL,  0, NONSTRICT,
      { "par8-volume" } },
    { "par10-volume", IN,  SCALAR,    LTYP_LEVEL,  0, NONSTRICT,
      { "par9-volume" } },
    { "par11-volume", IN,  SCALAR,    LTYP_LEVEL,  0, NONSTRICT,
      { "par10-volume" } },
    { "par12-volume", IN,  SCALAR,    LTYP_LEVEL,  0, NONSTRICT,
      { "par11-volume" } },
    { "par13-volume", IN,  SCALAR,    LTYP_LEVEL,  0, NONSTRICT,
      { "par12-volume" } },
    { "par14-volume", IN,  SCALAR,    LTYP_LEVEL,  0, NONSTRICT,
      { "par13-volume" } },
    { "par15-volume", IN,  SCALAR,    LTYP_LEVEL,  0, NONSTRICT,
      { "par14-volume" } },
    { "par16-volume", IN,  SCALAR,    LTYP_LEVEL,  0, NONSTRICT,
      { "par15-volume" } },
    END_PORT_DECLARATIONS
};


/*** Types ***/

typedef struct {
    port * out;
    port * out_vol;
    port * f;
    port * har_vols[HARMONICS];
    osc * oscs[HARMONICS];
    unsigned harmonics;
    sample * buf;
    sample * buf_a;
} context_t;


/*** Public functions ***/

lerr lm_go(void ** context, unsigned n_ports, port * ports[])
{
    context_t * x;
    char * s;
    int i;

    x = l_malloc(sizeof(context_t));
    x->out = port_addrof("out", n_ports, ports);
```

```
    x->out_vol = port_addrof("out-volume", n_ports, ports);
    x->f = port_addrof("f", n_ports, ports);

    x->har_vols[0] = port_addrof("fund-volume", n_ports, ports);
    for (i = 1; i < HARMONICS; i++) {
        /* FIXME: check s for NULL. */
        asprintf(&s, "par%d-volume", i + 1);
        x->har_vols[i] = port_addrof(s, n_ports, ports);
        l_free(s);
        if (!port_ready(x->har_vols[i]))
            break;
    }
    x->harmonics = i;

    for (i = 0; i < x->harmonics; i++)
        x->oscs[i] = osc_new(0.0, 0.0);

    x->buf = l_malloc(sizeof(sample) * samples_per_tick());
    x->buf_a = l_malloc(sizeof(sample) * samples_per_tick());

    *context = x;

    return OK;
}

lerr lm_pause(void * context)
{
    return OK;
}

lerr lm_stop(void * context)
{
    context_t * x = context;
    int i;

    l_free(x->buf);
    l_free(x->buf_a);
    for (i = 0; i < x->harmonics; i++)
        osc_free(x->oscs[i]);
    l_free(x);

    return OK;
}

lerr lm_tick(void * context)
{
    context_t * x = context;
    freq f;
    level out_vol;
    level har_vols[HARMONICS];
```

```
    int i, j;

    l_recv(x->f, &f);
    l_recv(x->out_vol, &out_vol);
    for (i = 0; i < x->harmonics; i++)
        l_recv(x->har_vols[i], &(har_vols[i]));

    for (j = 0; j < samples_per_tick(); j++)
        x->buf_a[j] = 0.0;
    for (i = 0; i < x->harmonics; i++) {
        /* Bail if we go above the Nyquist frequency.
           FIXME: perhaps warn somehow about this? */
        if (f * (i + 1) > samples_per_second() / 2)
            break;
        osc_sine(x->buf, x->oscs[i], f * (i + 1), samples_per_tick());
        for (j = 0; j < samples_per_tick(); j++)
            x->buf_a[j] += x->buf[j] * har_vols[i];
    }

    for (j = 0; j < samples_per_tick(); j++)
        x->buf_a[j] *= out_vol;
    l_send(x->out, x->buf_a);

    return OK;
}
```

modules/synth-fm.c

```
/* $Id: synth-fm.c,v 1.1 2001/04/20 05:49:01 reid Exp $ */

#include "lambda.h"


/*** Identification ***/

unsigned magic = MOD_MAGIC_1_0_0;
char name[] = "synth-fm";
char version[] = "1.0d1";
char description[] = "Simple FM synth; carrier and modulator are sine waves.";


/*** Ports ***/

port_decl port_decls[] = {
    { "f-car", IN,  SCALAR,    LTYP_FREQ,   0, STRICT, {} },
    { "f-mod", IN,  SCALAR,    LTYP_FREQ,   0, STRICT, {} },
    { "index", IN,  SCALAR,    LTYP_DOUBLE, 0, STRICT, {} },
    { "out",   OUT, ARRAY_REL, LTYP_SAMPLE, 1, STRICT, {} }
};
```

```
/*** Types ***/

typedef struct {
    port * f_car, * f_mod;
    port * index;
    port * out;
    osc * o_car, * o_mod;
    sample * buf;
} context_t;


/*** Public functions ***/

lerr lm_go(void ** context, unsigned n_ports, port * ports[])
{
    context_t * x;

    x = l_malloc(sizeof(context_t));

    x->f_car = port_addrof("f-car", n_ports, ports);
    x->f_mod = port_addrof("f-mod", n_ports, ports);
    x->index = port_addrof("index", n_ports, ports);
    x->out = port_addrof("out", n_ports, ports);

    x->o_car = osc_new(0.0, 0.0);
    x->o_mod = osc_new(0.0, 0.0);

    x->buf = l_malloc(samples_per_tick() * sizeof(sample));

    *context = x;
    return OK;
}

lerr lm_pause(void * context)
{
    return OK;
}

lerr lm_stop(void * context)
{
    context_t * x = context;

    l_free(x->buf);
    osc_free(x->o_car);
    osc_free(x->o_mod);
    l_free(x);

    return OK;
```

```
}

lerr lm_tick(void * context)
{
    context_t * x = context;
    freq f_mod, f_car;
    double index, deviation;
    int i;

    l_recv(x->f_mod, &f_mod);
    l_recv(x->f_car, &f_car);
    l_recv(x->index, &index);
    deviation = index * f_mod;

    osc_sine(x->buf, x->o_mod, f_mod, samples_per_tick());
    for (i = 0; i < samples_per_tick(); i++)
        osc_sine(&(x->buf[i]), x->o_car, f_car + deviation * x->buf[i], 1);

    l_send(x->out, x->buf);
    return OK;
}
```