Introduction to Computer Music

Week 2

Version 2, revised 3 Sep 2018

Roger B. Dannenberg

Topics Discussed: **Unit Generators, Implementation, Functional Programming, Wavetable Synthesis, Scores in Nyquist, Score Manipulation**

1 Unit Generators

In the 1950's Max Mathews conceived of sound synthesis by software using networks of modules he called *unit generators*. A unit generator Unit generators (sometimes called *ugens*) are basic building blocks for signal processing in many computer music programming languages.

Unit generators are used to construct synthesis and signal processing algorithms in software. For example, the simple unit generator `osc` generates a sinusoidal waveform at a fixed frequency. `env` generates an "envelope" to control amplitude. Multiplication of two signals can be achieved with a `mult` unit generator (created with the `*` operator), so `osc(c4) * env(0.01, 0.02, 0.1, 1, 0.9, 0.8)` creates a sinusoid with amplitude that varies according to an envelope.

Figure 1 illustrates some unit generators. Lines represent audio signals, control signals and numbers.

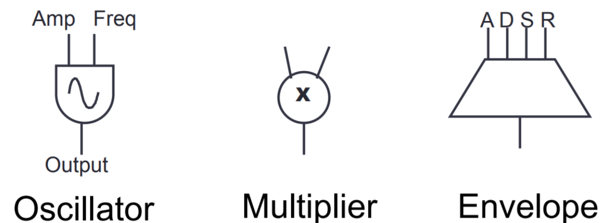

Figure 1: Some examples of Unit Generators.

In many languages, unit generators can be thought of as interconnected objects that pass samples from object to object, performing calculations on them. In Nyquist, we think of unit generators as functions with sounds as inputs and outputs. Semantically, this is an accurate view, but since sounds can be very large (typically about 10MB/minute), Nyquist uses a clever implementation based on *incremental lazy evaluation* so that sounds rarely exist as complete arrays of samples. Instead, sounds are typically computed in small chunks that are "consumed" by other unit generators and quickly deleted to conserve memory.

Figure 2 shows how unit generators can be combined. Outputs from an oscillator and an envelope generator serve as inputs to the multiply unit generator in this figure.

Figure 3 shows how the "circuit diagram" or "signal flow diagram" notation used in Figure 2 relates to the functional notation of Nyquist. As you can see, whereever there is output from one unit generator to the input of another as shown on the left, we can express that as nested function calls as shown in the expression on the right.

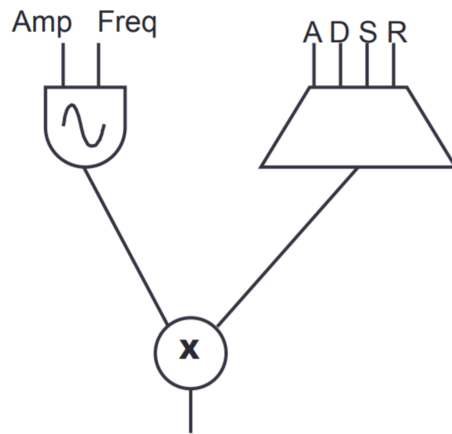

Figure 2: Combining Unit Generator.

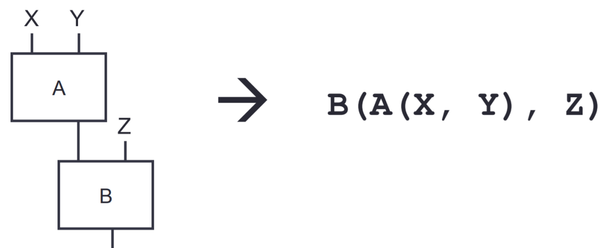

Figure 3: Unit Generators in Nyquist.

1.1 Some Basic Unit Generators

The `osc` function generates a sound using a table-lookup oscillator. There are a number of optional parameters, but the default is to compute a sinusoid with an amplitude of 1.0. The parameter 60 designates a pitch of middle C. (Pitch specification will be described in greater detail later.) The result of the `osc` function is a sound. To hear a sound, you must use the play command, which plays the file through the machine's D/A converters. E.g. you can write `play osc(c4)` to play a sine tone.

It is often convenient to construct signals in Nyquist using a list of (time, value) breakpoints which are linearly interpolated to form a smooth signal. The function `pwl` is a versatile unit generator to create Piece-Wise Linear (PWL) signals and will be described in more detail below.

An envelope constructed by `pwl` is applied to another sound by multiplication using the multiply (`*`) operator. For example, you can make the simple sine tone sound smoother by giving it an envelope:

```
play osc(c4) * pwl(0.03, 1, 0.8, 1, 1)
```

While this example shows a smooth envelope multiplied by an audio signal, you can also multiply audio signals to achieve what is often called ring modulation. For example:

```
play osc(c4) * osc(g4)
```

1.2 Evaluation

Normally, Nyquist expressions (whether written in SAL or Lisp syntax) evaluate their parameters, then apply the function. If we write `f(a, b)`, Nyquist will evaluate `a` and `b`, then pass the resulting values to function `f`.

Sounds are different. If Nyquist evaluated sounds immediately, they could be huge. Even something as simple as multiply could require memory for two huge input sounds and one equally huge output sound. Multiplying two 10-minute sounds would require 30 minutes' worth of memory, or about 300MB. This might not be a problem, but what happens if you are working with multi-channel audio, longer sounds, or more parameters?

To avoid storing huge values in memory, Nyquist uses lazy evaluation. Sounds are more like promises to deliver samples when asked, or you can think of a sound as an object with the *potential* to compute samples. Samples are computed only when they are needed. Nyquist Sounds can contain either samples or the potential to deliver samples, or some combination.

1.3 Unit Generator Implementation

What is inside a Unit Generator and how do we access it? If sounds have the potential to deliver audio samples on demand, sounds must encapsulate some information, so sounds in Nyquist are basically represented by the unit generators that produce them. If a unit generator has inputs, the sound (represented by a unit generator) will also have references to those inputs. Unit generators are implemented as objects that contain internal state such as the phase and frequency of an oscillator. This state is used to produce samples when the unit generator is called upon.

Although objects are used in the implementation, programs in Nyquist do not have access to the internal state of these objects. You can pass sounds as arguments to other unit generator functions and you can play sounds or write sounds to files, but you cannot access or modify these sound objects. Thus, it is more correct to think of sounds as *values* rather than objects. "Object" implies state and the possibility that the state can change. In contrast, a sound in Nyquist represents a long stream of samples that will eventually be computed and whose values are predetermined and immutable.

Other languages often expose unit generators as mutable objects and expose connections between unit generators as "patches" that can be modified. In this model, sound is computed by making a pass over the graph of interconnected unit generators, computing either one sample or a small block of samples. By making repeated passes over the graph, sound is incrementally computed.

While this incremental block-by-block computation sounds efficient (and it is), this is exactly what happens with Nyquist, at least in typical applications. In Nyquist, the `play` command demands a block of samples, and all the Nyquist sounds do some computation to produce the samples, but they are "lazy" so they only compute incrementally. In most cases, intermediate results are all computed incrementally, used, and then freed quickly so that the total memory requirements are modest.

2 Storing Sounds or Not Storing Sounds

If you write

`play` *sound-expression*

then *sound-expression* can be evaluated incrementally and after playing the samples, there is no way to access them, so Nyquist is able to free the sample memory immediately. The entire sound is never actually present in memory at once.

On the other hand, if you write:

`set var =` *sound-expression*

then initially `var` will just be a reference to an object with the *potential* to compute samples. However, if you `play var`, the samples must be computed. And since `var` retains a reference to the samples, they cannot be deleted. Therefore, as the sound plays, the samples will build up in memory.

In general, you should *never assign sounds to global variables* because this will prevent Nyquist from efficiently freeing the samples.

2.1 Functional Programming in Nyquist

Programs are expressions!

As much as possible, Nyquist programs should be constructed in terms of functions and values rather than variables and assignment statements.

Avoid sequences of statements and use nested expressions instead. Compose functions to get complex behaviors rather than performing sequential steps. An example of composing a nested expression is:

```
f(g(x), h(x))
```

An exception is this: Assigning expressions to local variables can make programs easier to read by keeping expressions shorter and simpler. However, you should *only assign values to local variables once*. For example, the previous nested expression could be expanded using local variables as follows (in SAL):

```
;; rewrite exec f(g(x), h(x))
begin with gg, hh ;; local variables
  set gg = g(x)
  set hh = h(x)
  exec f(gg, hh)
end
```

2.2 Eliminating Global Variables

What if you want to use the same sound twice? Wouldn't you save it in a variable?

Generally, this is a bad idea, because, as mentioned before, storing a sound in a variable can cause Nyquist to compute the sound and keep it in memory. There are other technical reasons not to store sounds in variables – mainly, sounds have an internal start time, and sounds are immutable, so if you compute a sound at time zero and store it in a variable, then try to use it later, you will have to write some extra code to derive a new sound at the desired starting time.

Instead of using global variables, you should generally use (global) functions. Here is an example of something to avoid:

```
;; this is NOT GOOD STYLE
set mysound = pluck(c4)
;; attempt to play mysound twice
;; this expression has problems but it might work
play seq(mysound, mysound)
```

Instead, you should write something like this:

```
;; this is GOOD STYLE
function mysound() return pluck(c4)
play seq(mysound(), mysound())
```

Now, `mysound` is a *function* that *computes* samples rather than storing them. You *could* complain that now `mysound` will be computed twice and in fact some randomness is involved so the second sound will not be identical to the first, but this version is preferred because it is more memory efficient and more "functional."

3 Waveforms

Our next example will be presented in several steps. The goal is to create a sound using a wavetable consisting of several harmonics as opposed to a simple sinusoid. We begin with an explanation of harmonics. Then, in order to build a table, we will use a function that adds harmonics to form a wavetable.

3.1 Terminology – Harmonics, etc

The shape of a wave is directly related to its spectral content, or the particular frequencies, amplitudes and phases of its components. Spectral content is the primary factor in our perception of timbre or tone color. We are familiar with the fact that white light, when properly refracted, can be broken down into component colors, as in the rainbow. So too with a complex sound wave, which is the composite shape of multiple frequencies.

So far, we have made several references to sine waves, so called because they follow the plotted shape of the mathematical sine function. A perfect sine wave or its cosine cousin will produce a single frequency known as the fundamental. Once any deviation is introduced into the sinus shape (but not its basic period), other frequencies, known as harmonic partials are produced.

Partials are any additional frequencies but are not necessarily harmonic. Harmonics or harmonic partials are integer (whole number) multiples of the fundamental frequency (f) (1f, 2f, 3f, 4f...). Overtones refers to any partials above the fundamental. For convention's sake, we usually refer to the fundamental as partial #1. The first few harmonic partials are the fundamental frequency, octave above, octave plus perfect fifth above, 2 octaves above, two octaves and a major 3rd, two octaves and a major fifth, as pictured in Figure 4 for the pitch "A." After the eighth partial, the pitches begin to grow ever closer and do not necessarily correspond closely to equal-tempered pitches, as shown in the chart. In fact, even the fifths and thirds are slightly off their equal-tempered frequencies. You may note that the first few pitches correspond to the harmonic nodes of a violin (or any vibrating) string.

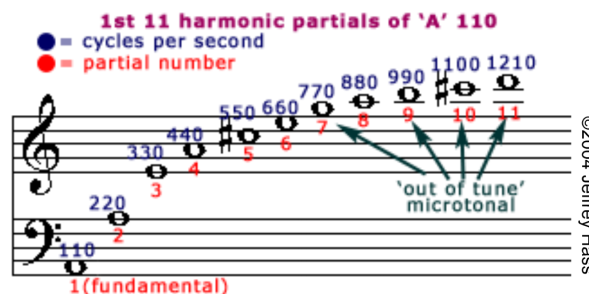

Figure 4: Relating harmonics to musical pitches.

3.2 Creating a Waveform by Summing Harmonics

In the example below, the function `mkwave` calls upon `build-harmonic` to generate a total of four harmonics with amplitudes 0.5, 0.25, 0.125, and 0.0625. These are scaled and added (using +) to create a waveform which is bound temporarily to `*table*`.

A complete Nyquist waveform is a list consisting of a sound, a pitch, and `T`, indicating a periodic waveform. The pitch gives the nominal pitch of the sound. (This is implicit in a single cycle wave table, but a sampled sound may have many periods of the fundamental.) Pitch is expressed in half-steps, where middle C is 60 steps, as in MIDI pitch numbers. The list of sound, pitch, and `T` is formed in the last line of `mkwave`: since build-harmonic computes signals with a duration of one second, the fundamental is 1 Hz, and the `hz-to-step` function converts to pitch (in units of steps) as required.

```
define function mkwave()
  begin
    set *table* = 0.5 * build-harmonic(1, 2048) +
                  0.25 * build-harmonic(2, 2048) +
                  0.125 * build-harmonic(3, 2048) +
                  0.0625 * build-harmonic(4, 2048)
    set *table* = list(*table*, hz-to-step(1.0), #t)
  end
```

Now that we have defined a function, the last step of this example is to build the wave. The following code calls `mkwave`, whihc sets `*mkwave*` as a side effect:

```
exec mkwave()
```

3.3 Wavetable Variables

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

Note that in Lisp and SAL, global variables often start and end with asterisks (`*`). These are not special syntax, they just happen to be legal characters for names, and their use is purely a convention. As an aside, it is the possibility of using "`*`", "`+`" and "`-`" in variables that forces SAL to require spaces around operators. "`a * b`" is an expression using multiplication, while "`a*b`" is simply a variable.

3.4 Using Waveforms

Now you know that `*table*` is a global variable, and if you set it, `osc` will use it:

```
exec mkwave() ;; defined above
play osc(c4)
```

This simple approach (setting `*table*`) is fine if you want to use the same waveform all the time, but in most cases, you will want to compute or select a waveform, use it for one sound, and then compute or select another waveform for the next sound. Using the global default waveform `*table*` is awkward.

A better way is to pass the waveform directly to `osc`. Here is an example to illustrate:

```
;; redefine mkwave to set *mytable* instead of *table*
define function mkwave()
  begin
    set *mytable* = 0.5 * build-harmonic(1, 2048) +
                    0.25 * build-harmonic(2, 2048) +
                    0.125 * build-harmonic(3, 2048) +
                    0.0625 * build-harmonic(4, 2048)
    set *mytable* = list(*mytable*, hz-to-step(1.0), #t)
  end
```

```
exec mkwave() ;; run the code to build *mytable*

play osc(c4, 1.0, *mytable*) ;; use *mytable*

;; note that osc(c4, 1.0) will still generate a sine tone
;;   because the default *table* is still *sine-table*
```

Now, you should be thinking "wait a minute, you said to avoid setting global variables to sounds, and now you are doing just that with these waveform examples. What a hypocrite!" Waveforms are a bit special because they are

- typically short so they do not claim much memory,
- typically used many times, so there can be significant savings by computing them once and saving them,
- not used directly as sounds but only as parameters to oscillators.

You do not have to save waveforms in variables, but it is common practice, in spite of the general advice to keep sounds out of global variables.

4 Piece-wise Linear Functions: pwl

It is often convenient to construct signals in Nyquist using a list of (time, value) breakpoints which are linearly interpolated to form a smooth signal. The `pwl` function takes a list of parameters which denote (time, value) pairs. There is an implicit initial (time, value) pair of (0, 0), and an implicit final value of 0. There should always be an odd number of parameters, since the final value (but not the final time) is implicit. Thus, the general form of `pwl` looks like:

`pwl`(t1, v1, t2, v2, . . . , tn)

and this results in a signal as shown in Figure 5.

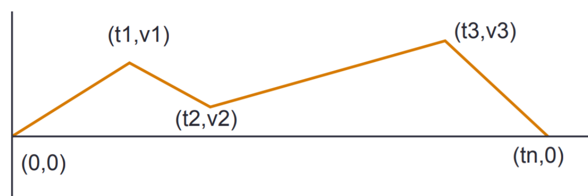

Figure 5: Piece-wise Linear Functions.

Here are some examples of `pwl`:

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

; a square pulse of height 10 and duration 5.
; Note that the first pair (0, 10) overrides the default initial
; point of (0, 0). Also, there are two points specified at time 5:
; (5, 10) and (5, 0). (The last 0 is implicit). The conflict is
; automatically resolved by pushing the (5, 10) breakpoint back to
; the previous sample, so the actual time will be 5 - 1/sr, where
```

```
; sr is the sample rate.
;
pwl(0, 10, 5, 10, 5)

; a constant function with the value zero over the time interval
; 0 to 3.5. This is a very degenerate form of pwl. Recall that there
; is an implicit initial point at (0, 0) and a final implicit value of
; 0, so this is really specifying two breakpoints: (0, 0) and (3.5, 0):
;
pwl(3.5)

; a linear ramp from 0 to 10 and duration 1.
; Note the ramp returns to zero at time 1. As with the square pulse
; above, the breakpoint (1, 10) is pushed back to the previous sample.
;
pwl(1, 10, 1)

; If you really want a linear ramp to reach its final value at the
; specified time, you need to make a signal that is one sample longer.
; The RAMP function does this:
;
ramp(10) ; ramp from 0 to 10 with duration 1 + one sample period
;
; RAMP is based on PWL; it is defined in nyquist.lsp.
;
```

4.1 Variants of pwl

Sometimes, you want a signal that does not start at zero or end at zero. There is also the option of interpolating between points with exponential curves instead of linear interpolation. There is also the option of specifying time intervals rather than absolute times. These options lead to many variants, for example:

`pwlv(v0, t1, v1, t2, v2, ..., tn, vn)` – "v" for "value first" is used for signals with non-zero starting and ending points
`pwev(v1, t2, l2, ..., tn, vn)` – exponential interpolation, $v_i > 0$
`pwlr(i1, v1, i2, v2, ..., in)` – relative intervals rather than absolute times

See the reference manual for more variants and combinations.

4.2 The Envelope Function: env

Envelopes created by `env` are a special case of the more general piece-wise linear functions created by `pwl`. The form of `env` is

`env(`*t1, t2, t4, l1, l2, l3, dur*`)`

(duration given by *dur* is optional). One advantage of `env` over `pwl` is that `env` allows you to give fixed "attack" and "decay" times that do not stretch with duration. In contrast, the default behavior for `pwl` is to stretch each segment in proportion when the duration changes. (We have not really discussed duration in Nyquist, but we will get there later.)

5 Basic Wavetable Synthesis

Now, you have seen examples of using the oscillator function (or unit generator) `osc` to make tones and various functions (unit generators) to make envelopes or smooth control signals. All we need to do is multiply them together

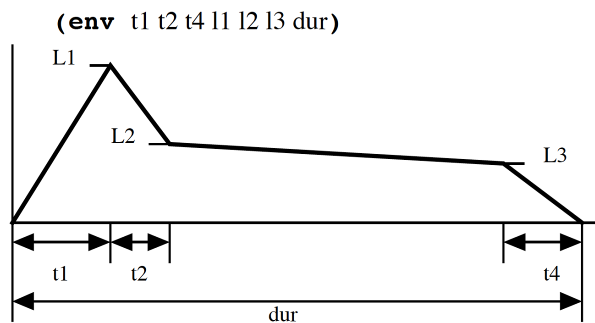

Figure 6: Envelope function env.

to get tones with smooth onsets and decays. Here is an example function to play a "note":

```
; env-note produces an enveloped note.  The duration
;   defaults to 1.0, but stretch can be used to change the duration.
;   Uses mkwave, second version defined above, to create *mytable*.

exec mkwave() ;; run the code to build *mytable*

function env-note(p)
  return osc(p, 1.0, *mytable*) *
         env(0.05, 0.1, 0.5, 1.0, 0.5, 0.4)

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

This is a basic synthesis algorithm called wavetable synthesis. The advantages are:

- simplicity – one oscillator, one envelope,
- efficiency – oscillator samples are generated by fast table lookup and (usually) linear interpolation,
- direct control – you can specify the desired envelope and pitch

Disadvantages are:

- the spectrum (strength of harmonics) does not change with pitch or time as in most acoustic instruments.

Often filters are added to change the spectrum over time, and we will see many other synthesis algorithms and variations of wavetable synthesis to overcome this problem.

6 Introduction to Scores

So far, we have seen how simple functions can be used in Nyquist to create individual *sound events*. We prefer this term to *notes*. While a sound event might be described as a note, the term *note* usually implies a single musical tone with a well-defined pitch. A *note* is conventionally described by:

- pitch – from low (bass) to high,
- starting time (notes begin and end),

- duration – how long is the note,
- loudness – sometimes called *dynamics*,
- timbre – everything else such as the instrument or sound quality, softness, harshness, noise, vowel sound, etc.)

while *sound event* captures a much broader range of possible sounds). A *sound event* can have:

- pitch, but may be unpitched noise or combinations,
- time – sound events begin and end,
- duration – how long is the event,
- loudness – also known as *dynamics*,
- potentially many evolving qualities.

Now, we consider how to organize sound events in time using *scores* in Nyquist. What is a *score*? Authors write books. Composers write *scores*. Figure 7 illustrates a conventional score. A score is basically a graphical display of music intended for conductors and performers. Usually, scores display a set of notes including their pitches, timing, instruments, and dynamics. In computer music, we define *score* to include computer readable representations of sets of notes or sound events.

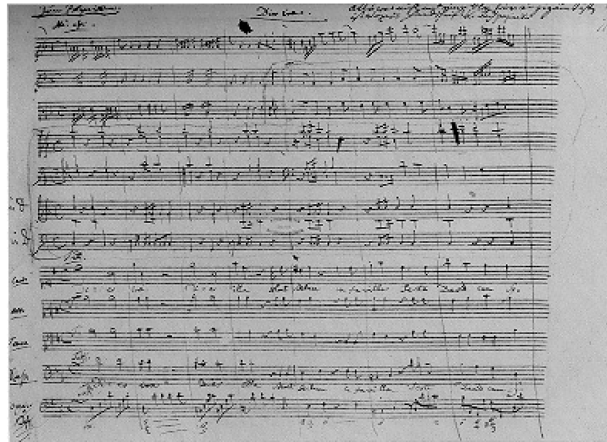

Figure 7: A score written by Mozart.

6.1 Terminology – Pitch

Musical scales are built from two-sizes of pitch intervals: whole steps and half steps, where a whole step represents about a 12 percent change in frequency, and a half step is about a 6 percent change. A whole step is exactly two half steps. Therefore the basic unit in Western music is the half step, but this is a bit wordy, so in Nyquist, we call these *steps*. (Physicists have the unit *Hertz* to denote "cycles per second." Wouldn't it be great if we had a special name to denote half-steps? How about the *Bach* since J. S. Bach's Well-Tempered Clavier is a landmark in the development of the fixed-size half step, or the *Schoenberg*, honoring Arnold's development of 12-tone music. Wouldn't it be cool to say 440 Hertz is 69 Bachs? Or to argue whether it's "Bach" or "Bachs"? But I digress ....)

Since Western music more-or-less uses integer numbers of half-steps for pitches, we represent pitches with integers. Middle C (ISO C4) is arbitrarily represented by 60. Nyquist pre-defines a number of convenient variables to represent pitches symbolically. We have `c4` = 60, `cs4` (C# or C-sharp) = 61, `cf4` (Cb or C-flat) = 59, `b3` (B

natural, third octave) = 59, `bs3` (B# or B-sharp, 3rd octave) = 60, etc. Note: In Nyquist, we can use non-integers to denote detuned or microtonal pitches: 60.5 is a quarter step above 60 (`C4`).

Some other useful facts: Steps are logarithms of frequency, and frequency doubles every 12 steps. Doubling frequency (or halving) is called an interval of an "octave."

6.2 Lists

Scores are built on lists, so let's learn about lists.

6.2.1 Lists in Nyquist

Lists in Nyquist are represented as standard *singly-linked* lists. Every element cell in the list contains a link to the value and a link to the next element. The last element links to *nil*, which can be viewed as pointing to an empty list. Nyquist uses *dynamic typing* so that lists can contain any types of elements or any mixture of types; types are not explicitly declared before they are used. Also, a list can have nested lists within it, which means you can make any binary tree structure through *arbitrary nesting* of lists.

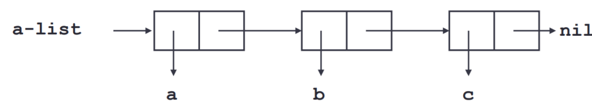

Figure 8: Lists in Nyquist.

6.2.2 Notation

Although we can manipulate pointers directly to construct lists, this is frowned upon. Instead, we simply write expressions for lists. In SAL, we use curly brace notations for literal lists, e.g. `{a b c}`. Note that the three elements here are literal symbols, not variables (no evaluation takes place, so these symbols denote themselves, not the values of variables named by the symbols). To construct a list from variables, we call the `list` function with an arbitrary number of parameters, which are the list elements, e.g. `list(a, b, c)`. These parameters are evaluated as expressions in the normal way and the values of these expressions become list elements.

6.2.3 Literals, Variables, Quoting, Cons

Consider the following:

```
set a = 1, b = 2, c = 3
print {a b c}
```

This prints: `{a b c}`. Why? Remember that the brace notation {} does not evaluate anything, so in this case, a list of the *symbols* `a`, `b` and `c` is formed. To make a list of the *values* of `a`, `b` and `c`, use `list`, which evaluates its arguments:

```
print list(a, b, c)
```

This prints: `{1 2 3}`.

What about numbers? Consider

```
print list(1, 2, 3)
```

This prints: `{1 2 3}`. Why? Because *numbers are evaluated immediately by the Nyquist (SAL or Lisp) interpreter as the numbers are read. They become either integers (known as type FIXNUM) or floating point numbers (known as type FLONUM).*

What if you want to use `list` to construct a list of symbols?

```
print list(quote(a),  quote(b), quote(c))
```

This prints: `{a b c}`. The `quote()` form can enclose any expression, but typically just a symbol. The `quote()` form returns the symbol without evaluation.

If you want to add an element to a list, there is a special function, `cons`:

```
print cons(a, {b})
```

This prints: `{1 b}`. Study this carefully; the first argument becomes the first element of a new list. The *elements* of the second argument (a list) form the remaining elements of the new list.

In contrast, here is what happens with `list`:

```
print list(a, {b c d})
```

This prints: `{1 {b c d}}`. Study this carefully too; the first argument becomes the first element of a new list. The second argument becomes the second *element* of the new list, so the new list has two elements.

7 Scores

In Nyquist, scores are represented by lists of data. The form of a Nyquist score is the following:

```
{ sound-event-1
  sound-event-2
  ...
  sound-event-n }
```

where a *sound event* is also a list consisting of the event time, duration, and an expression that can be evaluated to compute the event. The expression, when evaluated, must return a sound:

```
{ {time-1 dur-1 expression-1}
  {time-2 dur-2 expression-2}
  ...
  {time-n dur-n expression-n} }
```

and where each expression consists of a function name (sometimes called the *instrument* and a list of keyword-value style parameters:

```
{ {time-1 dur-1 {instrument-1 pitch: 60}}
  {time-2 dur-2 {instrument-2 pitch: 62}}
  ...
  {time-n dur-n {instrument-3 pitch: 62 vel: 100}} }
```

Here is an example score:

```
{ {0 1 {note pitch: 60 vel: 100}}
  {1 1 {note pitch: 62 vel: 110}}
  {2 1 {note pitch: 64 vel: 120}} }
```

Important things to note (pardon the pun) are:

- Scores are data. You can compute scores using by writing code and using the list construction functions from the previous section (and see the reference manual for many more).
- Expressions in scores are *lists*, not SAL expressions. The first element of the list is the function to call. The remaining elements form the parameter list.
- Expressions use keyword-style parameters, never positional parameters. The rationale is that keywords label the values, allowing us to pass the same score data to different *instruments*, which may implement some keywords, ignore others, and provide default values for missing keywords.

- keyword parameters also allow us to treat scores as data. For example, Nyquist has a built-in function to find all the `pitch:` parameters and transpose them. If positional parameters were used, the transpose function would have to have information about each instrument function to find the pitch values (if any). Keyword parameters are more "self-defining."

7.1 The score-begin-end "instrument" Event

Sometimes it is convenient to give the entire score a begin time and an end time because the "logical" time span of a score may include some silence. This information can be useful when splicing scores together. To indicate start and end times of the score, insert a "sound event" of the form `{0 0 {score-begin-end 1.2 6}}`. In this case, the score occupies the time period from 1.2 to 6 seconds.

For example, if we want the previous score, which nominally ends at time 3 to contain an extra second of silence at the end, we can specify the time span of the score is from 0 to 4 as follows:

```
{ {0 0 {score-begin-end 0 4}}
  {0 1 {note pitch: 60 vel: 100}}
  {1 1 {note pitch: 62 vel: 110}}
  {2 1 {note pitch: 64 vel: 120}} }
```

7.2 Playing a Score

To interpret a score and produce a sound, we use the `timed-seq()` function. The following plays the previous score:

```
set myscore = {
  {0 0 {score-begin-end 0 4}}
  {0 1 {note pitch: 60 vel: 100}}
  {1 1 {note pitch: 62 vel: 110}}
  {2 1 {note pitch: 64 vel: 120}} }
play timed-seq(myscore)
```

7.3 Making an Instrument

Now you know all you need to know to make scores. The previous example will work because `note` is a built-in function in Nyquist that uses a built-in piano synthesizer. But it might be helpful to see a custom "instrument" definition, making the connection between scores and the wavetable synthesis examples we saw earlier. In the next example, we define a new instrument function that calls on our existing `env-note` instrument. The reason for making a new function to be our "instrument" is we want to use keyword parameters. Then, we modify `myscore` to use `myinstr`. .

```
variable *mytable* ;; declaration avoids parser warnings

function mkwave()
  begin
    set *mytable* = 0.5 * build-harmonic(1, 2048) +
                    0.25 * build-harmonic(2, 2048) +
                    0.125 * build-harmonic(3, 2048) +
                    0.0625 * build-harmonic(4, 2048)
    set *mytable* = list(*mytable*, hz-to-step(1.0), #t)
  end

exec mkwave()

function env-note(p)
```

```
  return osc(p, 1.0, *mytable*) *
         env(0.05, 0.1, 0.5, 1.0, 0.5, 0.4)

;; define the "instrument" myinstr that uses keyword parameters
;; this is just a stub, env-note() does most of the work...
function myinstr(pitch: 60, vel: 100)
  return env-note(pitch) * vel-to-linear(vel)

set myscore = {
  {0 0 {score-begin-end 0 4}}
  {0 1 {myinstr pitch: 60 vel: 50}}
  {1 1 {myinstr pitch: 62 vel: 70}}
  {2 1 {myinstr pitch: 64 vel: 120}} }

play timed-seq(myscore)
```

Note that in `myinstr` we scale the amplitude of the output by `vel-to-linear(vel)` to get some loudness control. "vel" is short for "velocity" which refers to the velocity of piano keys – higher numbers mean faster which means louder. The "vel" scale is nominally 1 to 127 (as in the MIDI standard) and `vel-to-linear()` converts this to a scale factor. We will learn more about amplitude later.

8 Summary

Now you should know how to build a simple wavetable instrument with the waveform consisting of any set of harmonics and with an arbitrary envelope controlled by `pwl` or `env`. You can also write or compute scores containing many instances of your instrument function organized in time, and you can synthesize the score using `timed-seq`. You might experiment by creating different waveforms, different envelopes, using non-integer pitches for microtuning, or notes overlapping in time to create chords or clusters.

9 Acknowledgments

Thanks to Sai Samarth and Shuqi Dai for editing assistance.

Portions of this work are taken almost verbatim from *Music and Computers, A Theoretical and Historical Approach* (http://sites.music.columbia.edu/cmc/MusicAndComputers/) by Phil Burk, Larry Polansky, Douglas Repetto, Mary Robert, and Dan Rockmore. Other portions are taken almost verbatim from *Introduction to Computer Music: Volume One* (http://www.indiana.edu/~eemusic/etext/toc.shtml) by Jeffrey Hass. I would like to thank these authors for generously sharing their work and knowledge.