A Functional Language for Sound Synthesis With Lazy Evaluation and Behavioral Abstraction

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

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

DRAFT Submitted to the IEEE Computer Society 1990 International Conference on Computer Languages

Abstract

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

Key Words: Computer Language, Functional Programming, Lazy Evaluation, Behavioral Abstraction, Computer Music, Audio, Music Synthesis, Time.

1. Introduction

Fugue is a language for composition and sound synthesis. Features of Fugue include:

1. A full interactive environment based on Lisp,
2. A language which does not force a high-level distinction between the "score" and the "orchestra",
3. Support for behavioral abstraction,
4. The ability to work in terms of time including actual and perceptual start and stop times, and
5. A time- and memory-efficient implementation.

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

Fugue semantics include behavioral abstraction as introduced by Arctic[1] and Canon[2]. The motivation for behavioral abstraction is the idea that one should be able to describe behaviors that respond appropriately to their environment. For example, stretching a sound may mean one thing in the context of the synthesis of a single tone, and another in the context of producing a drum roll. It almost never means to compute a short sound and then resample it to make it longer (causing an undesireable drop in pitch). Fugue allows the programmer to describe abstract behaviors that "know" how to stretch, transpose, change loudness, and shift in time. Transformation operators are provided to operate on these abstractions.

Composition requires that sounds be placed simultaneously together, in sequence, and at arbitrary offsets. Thus, time is an important element in the semantics of languages for expressing music. Because musical sounds often have attack and release portions, we make a distinction between the absolute first and last samples of a sound and the perceptual start and end to which other sounds should be aligned.

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

2. Related Work

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

Music V takes a semi-functional approach to sound *generation*. Unit generators in Music V are similar to functions in that they can be combined using functional composition. The resulting instruments can be applied to parameter lists; however, instruments cannot be combined with other instruments, nor can scores be constructed hierarchically. This division between sound manipulators (or generators) and parameter lists results in two coordinated languages and a corresponding separation between the

"orchestra" and "score", as the instrument definitions and note lists are called. Other consequences include a non-interactive environment because instruments are compiled before any notes can be processed. An interesting aspect of Music V is the idea that an *instance* of an instrument is created for each note specified in the score.

Kyma[3, 4] and the Sun/Mercury Workstation[5] take a different approach, treating sound manipulators and sound generators as objects that can be "patched" together. This results in intuitive systems for synthesis, but there are problems when these systems are extended for composing larger structures. A level of indirection is required to manipulate graphs of unit generators which in turn manipulate sounds rather than to manipulate sounds directly. Various extensions have become necessary in order to handle graphs that change over time, but this also adds complexity to programs. Symbol processing and working with data structures are also difficult with these graph-oriented program representations. Also, both systems run all objects in synchrony, thereby assuming a global sample rate for digital signals.

SRL[6] is a signal processing language that represents signals as parameterized computations. SRL signals are immutable objects that can be reused. SRL supports lazy evaluation and function caching by retaining a symbolic representation of all signals. SRL lacks in many musically useful concepts such as the starting time and duration of signals and behavioral abstraction. Also the user must explicitly free buffers when they are no longer needed.

Formes[7, 8] takes an object-oriented approach to the computation of functions of time, but Formes was not designed to compute audio directly. Formes was originally designed to compute control information for the Chant[9] synthesis system.

Arctic[1, 10, 11], like Formes, is intended to compute functions of time that in turn control audio signal processes. Unlike Formes, Arctic is a functional language in which functions of time are primitive data types. Functions of time and real numbers are the only data types implemented in Arctic, so there is difficulty implementing many state-oriented algorithms.

Fugue integrates a score language with sound analysis and synthesis capabilities. The next four sections will describe Fugue's facilities for music representation and for signal processing. Then, we present a more detailed description of the implementation of Fugue followed by conclusions. The next section presents a few hypothetical examples of applications of Fugue, and the last section draws conclusions and describes areas for further research.

3. Programs as Scores

Historically, musical scores have been for the most part static data structures created by composers. Traditional scores do not express computation beyond a few simple abbreviations to indicate repeats and alternate endings. There is a good reason for this: traditional scores are data intensive and contain a wealth of detailed information. By and large, composers have traditionally wanted to express the results of their creativity, not the *process* of creation.

Consequently, lists of notes and their attributes have been the standard form of machine-readable score for many years. As data structures, note lists can be transformed in time, pitch, or along any other dimension defined by attributes. It is generally easier to generate and manipulate data structures than programs. For example, making all the notes in a section louder is easy to do if the notes are represented as data. On the other hand, note lists suffer from the fact that they are not programs. In particular, there

comes a time when "loudness" (and every other note list parameter) must be interpreted to produce or control sound. The point at which interpretation starts defines the boundary between the "score" and the "orchestra". This is an undesirable distinction because it separates the expression of music into two levels, one level with, and one without an underlying computational semantics.

In addition to this problem, note lists are flat structures without hierarchy or abstraction capabilities. Suppose a composer wishes to express a drum roll in a note list. Assuming a single drum stroke is the only available sound, the composer must enter many drum strokes into the note list to simulate a roll. Making the drum roll crescendo (get louder) or last longer requires the composer to edit at the level of individual drum strokes.

Fugue provides an elegant and hierarchical way to express scores that combines the best qualities of note lists and executable programs. In Fugue, two constructs are used to combine sounds or notes into larger structures. The first combines sounds sequentially (for example to form a melody) and the second mixes sounds simultaneously (for example to form a chord):

`(seq s1 s2 s3 ...)`
arranges each sound sequentially in time.

`(sim s1 s2 s3 ...)`
arranges each sound simultaneously in time.

Thus, Fugue can easily model note lists. Fugue also provides various transformation operators that provide much of the power one gets from the ability to edit note lists as data structures. Since Fugue can express abstract behaviors — such as drum rolls, individual notes, and synthesis algorithms — in a single language, Fugue is more flexible and comprehensive than previous composition languages.

4. Behavioral Abstraction

Fugue allows the programmer/composer to define *behaviors* that describe how to generate a sound within a context. Behaviors that are not language primitives are hierarchical compositions of other behaviors. Once defined, a behavior can have many *instances*, each of which is evaluated in a different context and/or with different parameters. In this way, concepts such as "drum roll" or "glissando" can be defined once but applied in many different contexts. In other words, these behaviors can be instantiated with varying duration, loudness, pitch, articulation, and other qualities.

A potential liability of transformations on behaviors is that it is not always obvious how a behavior should be transformed. For example, in stretching a section of music that contains a trill, we do not necessarily want the trill to slow down, and we almost certainly do not to stretch the audio signal, causing the pitch to drop! Fugue provides defaults for transformations, but allows the programmer/composer to override the defaults with more appropriate transformations.

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

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

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

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

Instead, suppose we wish to change the pitch. We could write

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

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

Behaviors are ultimately thinly disguised Lisp functions, and they are defined using `defun`, for example:

```
(defun trill (pitch interval)
   (seqrep (i (truncate (/ *stretch* 0.16)))
      (stretch-abs 0.08
         (seq (note pitch)
              (note (+ pitch interval))))))
```

A trill is a rapid alternation between two pitches. When a trill is stretched, the rate of alternation stays the same and more notes are played to increase the total duration. In this example, the number of repetitions is computed in terms of `*stretch*`, part of the context in which `trill` is evaluated. Ordinarily, this context would apply to sub-expressions in `trill`, but `stretch-abs` is used to override the default (dynamically scoped) context with a duration of 0.08 seconds. Each sequence of two notes inside the `stretch-abs` context will have a duration of 0.16 seconds, and this sequence is repeated within `seqrep` to obtain the entire trill.

Several points should be emphasized in this example. First, the trill is evaluated within a context that, in addition to stretch, may specify a loudness contour, pitch transposition, and other transformations that are passed on to `note` without change. Notice that the programmer does not have to mention or even be aware of all aspects of the context, since elements of the context not mentioned will simply be inherited, in this case, by `note`. Second, notice that at the heart of the definition of `trill`, there is a list of two notes, illustrating how Fugue retains a strong declarative and data-oriented "feel" of traditional scores even though Fugue can express computations. Finally, notice that the entire behavior is free of side-effects. This property opens the possibility of lazy and/or parallel evaluation.

5. Transformations

The context is manipulated by a set of primitive operations. These operations are used to perform cutting and splicing, stretching, controlling the amount of *legato* (sound overlap), loudness, and pitch. Transformation operations manipulate the environment in which sounds are computed and may be applied from the score level all the way down to sound generation primitives.

`(at delay b)` "shifts" the behavior `b` by `delay` seconds.

`(extent beg end b)`
"clips" the behavior `b`, extracting the portion of `b` from `beg` seconds through `end` seconds.

`(stretch factor b)`
"stretches" `b` by `factor`.

`(sustain factor b)`
assuming `b` is composed of multiple events, causes the events within `b` to be overlapped by a factor. Factors greater than one cause a more legato feel, while factors less than one will feel more staccato.

`(loud factor b)`
alter the loudness of `b` by `factor`.

`(transpose amount b)`
transpose `b` by `amount`.

Although we speak of transformations as operating on behaviors, the transformations actually modify the context in which the behavior is evaluated rather than modifying a computed sound. This gives the abstract behaviors a chance to behave appropriately in context.

6. Signal Processing

In addition to its use in writing scores at the sonic event and musical note level, Fugue is intended as a versatile system for the analysis, synthesis, and processing of sound. Thus far, our efforts have focused on building an extensible kernel for Fugue and implementing some simple synthesis primitives. In the current implementation, sounds may be obtained using a generalized oscillation function or by loading sounds from files.

`(osc pitch dur sound phase)`
creates a specified `pitch` for `dur` seconds with the given `phase`, treating the given `sound` as one period of a periodic waveform. This primitive can be used to implement an oscillator as well as a sampler.

`(sfload name)` loads a sound file to create an internal sound value.

`(cue sound)` play the `sound` without transposition, adjusting for current time offset, volume, and extent.

`(s-compose samples)`
convert the lisp array `samples` into a sound.

Multiplication is used to modulate a sound with an envelope, and addition is used to "mix" sounds together.

7. Implementation

Fugue is implemented in a combination of C[12] and XLisp[13], which is in turn implemented in C. We use XLisp because it is fairly easy to extend with a new primitive type, and it is also easy to interface with C programs for signal processing. The use of two languages reflects our goal to provide an interactive and efficient environment. Lisp provides convenient and powerful interaction. C, on the other hand, allows for efficient implementation of low-level functions. While it might be possible to use a more efficient compiled Lisp, most of the computation time is taken by signal processing primitives, so the Lisp interpretation represents only a small overhead.

The transformation context in Fugue is implemented within Lisp. Operators such as `transpose` are macros that bind an element of the context (`*transpose*` in this case) and then evaluate the embedded expression. The binding is restored upon exit. The context is simply a set of global variables.

New synthesis techniques can be introduced into Fugue by combining existing behaviors or by writing new sound synthesis algorithms in C and making them callable from Lisp. Ultimately, we hope to move the inner loops of our synthesis algorithms into DSP microcode.

Several steps have been taken to increase time and memory efficiency, including multiple sample-rates and lazy evaluation. Multiple sample rates allow "control" signals to exist at a low sample rate as in

Music-11[14] and Csound[15], reducing time and memory requirements. However, there is no other distinction between control signals and audio signals. When it becomes necessary to manipulate two sounds with different sample rates, linear interpolation is used by default to resample the lower sample rate signal at the higher sample rate. Other types of interpolation can be used explicitly via function invocations. Since there is no distinction between control and audio signals, filters can be used to modify spectra or to smooth envelopes, and multiplication can be used uniformly for gain control, amplitude envelopes, or audio-rate amplitude modulation.

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

To illustrate the advantages of lazy evaluation, imagine a computation of the form

```
for i := 1 to 100 do
        myPiece := myPiece + MakeNote(i);
```

In Fugue, a roughly equivalent program would be:

```
(seqrep (i 100) (make-note i))
```

where `seqrep` is a control construct that concatenates some number of instances of a behavior, in this case 100 copies of `make-note`.

Since sounds are immutable values, each addition requires that new storage be allocated and initialized. If `myPiece` is very large compared to `MakeNote(i)`, most of the time will be spent copying samples to form the sum. This problem is avoided using lazy evaluation in which sample computations are deferred whenever possible. The most common sound manipulations are simply recorded in a data structure. These "primitive" manipulations currently consist of: amplitude scaling, extent (extracting a portion of a sound), shifting in time, stretching in time, and adding (mixing) sounds. When one or more of these sound manipulations are performed on a sound, no new sound samples are computed; only a new data structure is created to reflect the aggregate effect of these manipulations: addition causes a tree to be built, while the other manipulations simply affect values in the data structure. When an actual sample is needed (by a filter function, for example), this data structure must be "flattened" or "normalized", so that we may easily extract samples which correspond to the sound we want.

Note that this requires only one memory allocation, as opposed to the many smaller allocations that would be necessary if samples were generated by each operation. Also, successive combinations of amplitude scaling, extent, stretch, and shift do not cost any significant computation or storage allocation. All of this is hidden from the user and is performed automatically.

8. Storage Management

Sounds are managed at two levels in our implementation. At the Lisp level, the core of the Lisp interpreter treats sounds as pointers (memory addresses). When operations are performed on sounds, the interpreter passes a sound pointer to a C function that implements the operation. Similarly, the garbage collector treats sounds like any other Lisp data.

The garbage collector[16] locates and marks all data that can be accessed directly or indirectly, starting from variables and the run-time stack. Anything that cannot be accessed is placed on a free list for reuse at a later time. In the case of sounds, the garbage collector passes the sound pointer to a function — the destructor for sounds — before reclaiming the pointer storage.

The pointers managed by Lisp point to small Fugue structures that keep track of transformations such as amplitude scaling and time shifting. Since there may be several different structures that represent various transformations of a single sample, we store the actual sound samples separately. Each sound structure has a pointer to the samples. We use reference counts[17] to keep track of how many structures are referencing a given set of samples so that we can deallocate the samples when they are no longer referenced.

An alternative to the current memory management system would be to represent sound structures and even sound samples within Lisp. This would allow the garbage collector to collect sounds directly. We rejected this approach in order to make our sound management system more efficient and less dependent upon a particular Lisp implementation. Another alternative would be to store large sounds in files rather than in virtual memory. This would facilitate computing long compositions that take 100MB of storage or more. (Ten minutes of stereo audio sampled at 50KHz and stored as 32-bit floating point values takes 120MB of storage.)

9. An Example

To illustrate how the implementation works, we will show what the memory structures look like at each step of a sequence of operations. The operations will be:

```
(setf MYSOUND (sfload "mysound"))
(setf DEMO (stretch 3.0 (scale 2.0 (seq (cue MYSOUND)
                                        (cue MYSOUND)))))

(play DEMO)
```

The first line sets the variable MYSOUND to a sound stored in the file "mysound". The resulting configuration is shown in figure 9-1. The second line evaluates a score consisting of two copies of MYSOUND in sequence. Since only time shifting and addition are involved here, essentially no computation takes place, and the resulting configuration appears in figure 9-2. Finally, the third line forces the system to produce samples for DEMO. The representation for DEMO is replaced by one in which the actual samples have been computed and storage for samples has been allocated as shown in figure 9-3. Note that this last step is the only time that: (1) new storage for sample data is allocated, and (2) a new sound sample is actually computed.

10. Conclusion

Fugue is a new language that provides high-level operations on sounds. Fugue is unique in that it spans a range of computational tasks from score manipulation to synthesis within a single integrated language. Fugue already has an efficient implementation running on Unix workstations. We intend to improve this further by taking advantage of virtual copy and mapped file capabilities of the Mach (Accetta *et. al.* 1986) operating system and a DSP chip for signal processing. We also plan to extend Fugue with more sound functions from other systems such as Moore's Cmusic[18], Vercoe's Csound[15], NeXT's Sound Kit[19], and Lansky's Cmix[20].

Another possibility for investigation is the use of Fugue in parallel computation. Because of its

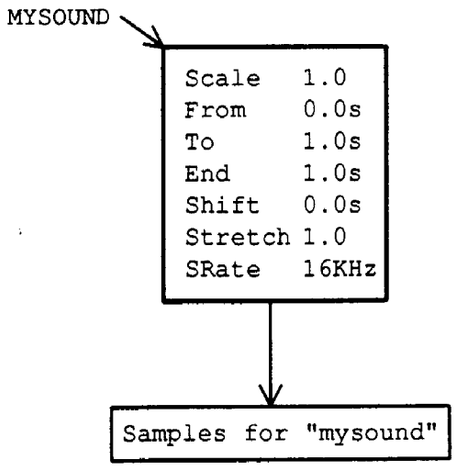

Figure 9-1: MYSOUND after (setf MYSOUND (SFLoad "mysound")), assuming the duration of MYSOUND is 1.0 Sec.

functional style, Fugue programs contain explicit parallelism in the form of the sim (for "simultaneous") construct. Even when there are data dependencies such as in the seq construct, lazy evaluation often defers signal computations so that the data dependencies can be resolved immediately. Then the signal processing can proceed in parallel. If sounds in Fugue were implemented as streams, then more parallelism could be obtained by lazily evaluating streams. There are many opportunities for compilation and optimization of Fugue behaviors that we have not explored.

Many of the ideas of Fugue seem appropriate for computer graphics and computer animation. The idea of behavioral abstraction seems to fit nicely with graphical transformations, e.g. "make this desk longer" or "make this tree bigger", and also with action in animations, for example "run faster". In computer animation, Fugue's notions of explicit timing and constructs for parallel and sequential behavior might be useful. For images, new constructs might be added to represent spacial as well as temporal relationships.

Currently, the context in Fugue is part of the language definition. A more general-purpose language might result if the context and transformation operators for changing the context could be defined explicitly by the programmer.

11. Acknowledgments

The authors would like to express thanks to the Carnegie Mellon School of Computer Science for making this work possible. This paper is based on an extended abstract appearing in the *Proceedings of the International Computer Music Conference*[21].

References

1. Dannenberg, R. B., "Arctic: A Functional Language for Real-Time Control", *1984 ACM Symposium on LISP and Functional Programming*, ACM, August 1984, pp. 96-103.

2. Dannenberg, Roger B., "The Canon Score Language", *Computer Music Journal*, Vol. 13, No. 1, Spring 1989, pp. 47-56.

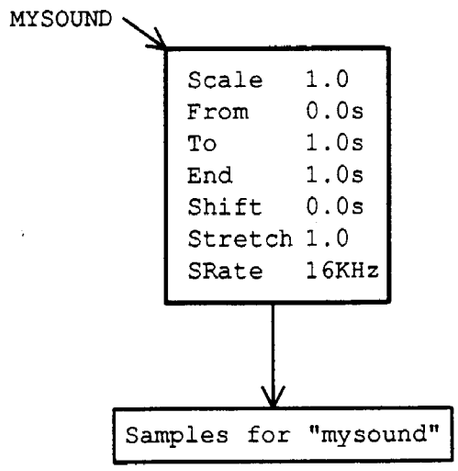

Figure 9-1: `MYSOUND` after `(setf MYSOUND (SFLoad "mysound"))`, assuming the duration of `MYSOUND` is 1.0 Sec.

functional style, Fugue programs contain explicit parallelism in the form of the `sim` (for "simultaneous") construct. Even when there are data dependencies such as in the `seq` construct, lazy evaluation often defers signal computations so that the data dependencies can be resolved immediately. Then the signal processing can proceed in parallel. If sounds in Fugue were implemented as streams, then more parallelism could be obtained by lazily evaluating streams. There are many opportunities for compilation and optimization of Fugue behaviors that we have not explored.

Many of the ideas of Fugue seem appropriate for computer graphics and computer animation. The idea of behavioral abstraction seems to fit nicely with graphical transformations, e.g. "make this desk longer" or "make this tree bigger", and also with action in animations, for example "run faster". In computer animation, Fugue's notions of explicit timing and constructs for parallel and sequential behavior might be useful. For images, new constructs might be added to represent spacial as well as temporal relationships.

Currently, the context in Fugue is part of the language definition. A more general-purpose language might result if the context and transformation operators for changing the context could be defined explicitly by the programmer.

11. Acknowledgments

The authors would like to express thanks to the Carnegie Mellon School of Computer Science for making this work possible. This paper is based on an extended abstract appearing in the *Proceedings of the International Computer Music Conference*[21].

References

1. Dannenberg, R. B., "Arctic: A Functional Language for Real-Time Control", *1984 ACM Symposium on LISP and Functional Programming*, ACM, August 1984, pp. 96-103.

2. Dannenberg, Roger B., "The Canon Score Language", *Computer Music Journal*, Vol. 13, No. 1, Spring 1989, pp. 47-56.

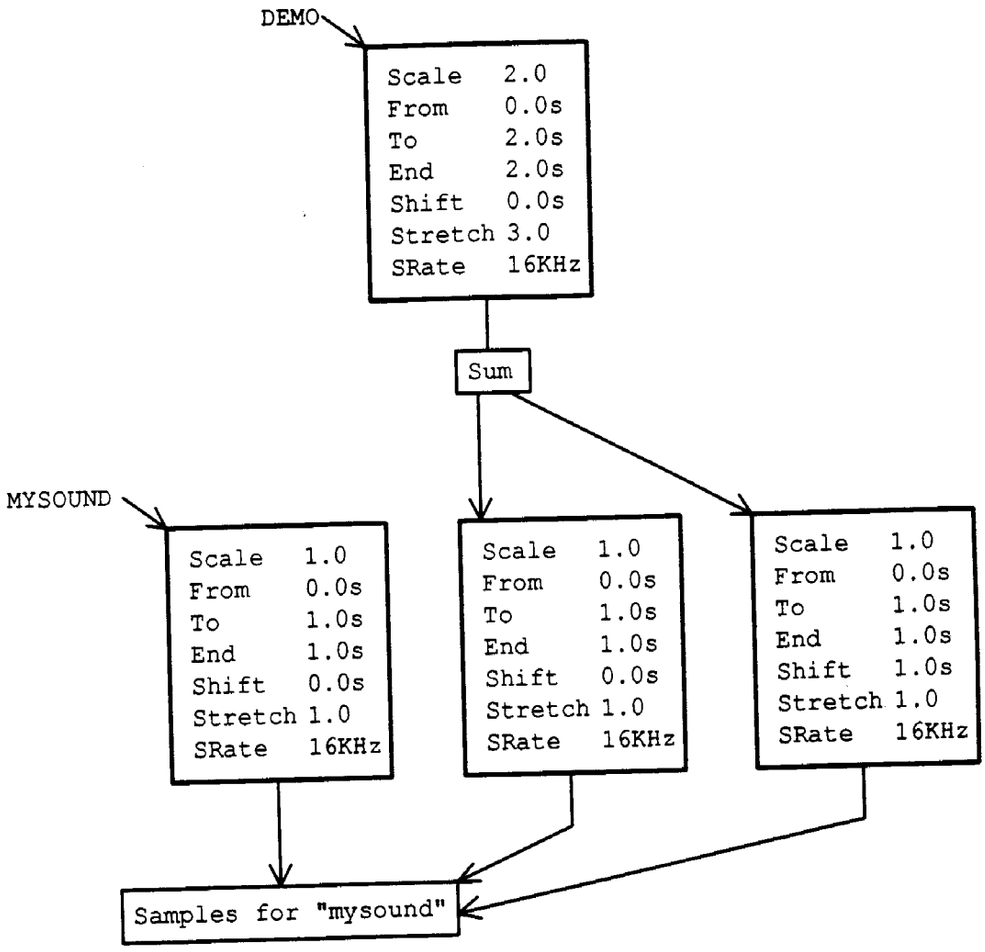

Figure 9-2: DEMO after `(setf DEMO (stretch 3.0 (scale 2.0 (seq (cue MYSOUND) (cue MYSOUND)))))`

3. Scaletti, C., and Johnson, E., "An Interactive Graphic Environment for Object-Oriented Music Composition and Sound Synthesis", *Proceedings of the 1988 Conference on Object-Oriented Languages and Systems*, ACM, 1988, pp. 18-26.

4. Scaletti, Carla, "The Kyma/Platypus Computer Music Workstation", *Computer Music Journal*, Vol. 13, No. 2, Summer 1989, pp. 23-38.

5. Rodet, Xavier and Eckel, Gerhard, "Dynamic Patches: Implementation and Control in the Sun-Mercury Workstation", *Proceedints of the 1988 International Computer Music Conference*, Computer Music Association, 1988, pp. 82-89.

6. Kopec, Gary E., "The Signal Representation Language SRL", *IEEE Transactions on Acoustics, Speech and Signal Processing*, Vol. ASSP-33, No. 4, August 1985, pp. 921-932.

7. Rodet, Xavier and Cointe, Pierre, "FORMES: Composition and Scheduling of Processes", *Computer Music Journal*, Vol. 8, No. 3, Fall 1984, pp. 32-50.

8. Cointe, Pierre and Rodet, Xavier, "Formes: an Object & Time Oriented System for Music Composition and Synthesis", *1984 Symposium on LISP and Functional Programming*, ACM,

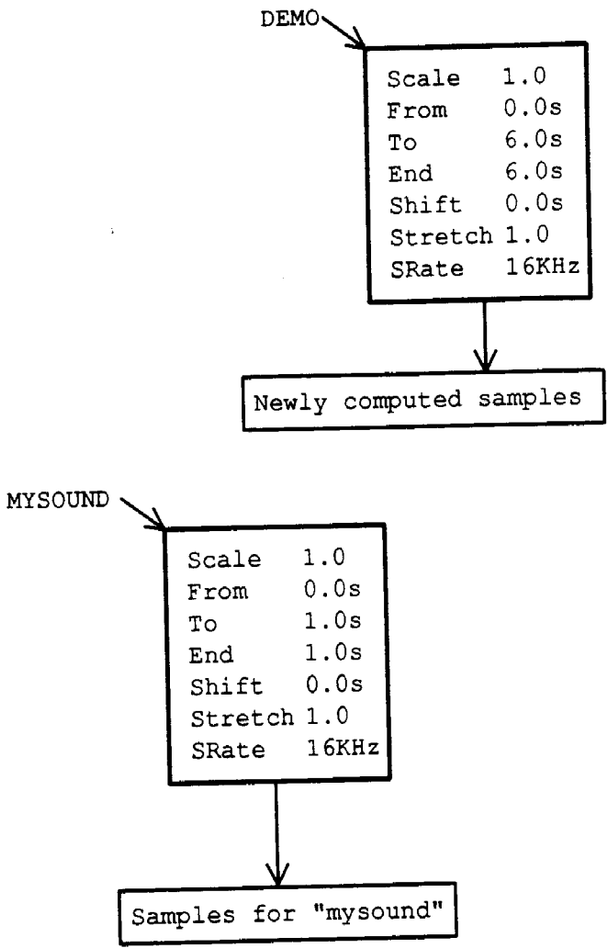

Figure 9-3: DEMO after (play DEMO)

1984, pp. 85-95.

9. Rodet, Xavier, Potard, Yves, and Barriere, Jean-Baptiste, "The CHANT Project: From Synthesis of the Singing Voice to Synthesis in General", *Computer Music Journal*, Vol. 8, No. 3, Fall 1984, pp. 15-31.

10. Dannenberg, Roger B., McAvinney, Paul, Rubine, Dean, "Arctic: A Functional Language for Real-Time Systems", *Computer Music Journal*, Vol. 10, No. 4, Winter 1986, pp. 67-78.

11. Rubine, Dean and Dannenberg, Roger B., "Arctic Programmer's Manual and Tutorial", Tech. report CMU-CS-87-110, Carnegie Mellon, 1987.

12. Kernighan, Brian M. and Richie, Dennis M., *The C Programming Language*, Prentice-Hall, Englewood Cliffs, 1978.

13. Betz, David, "XLISP: An Experimental Object-oriented Language, Version 1.7", (program documentation)

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

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

16. Schorr, H. and Waite, W., "An Efficient and Machine Independent Procedure for Garbage Collection in Various List Structures", *Communications of the ACM*, Vol. 10, No. 8, 1967, pp. 501-506.

17. Pratt, Terrence W., *Programming Languages: Design and Implementation*, Prentice-Hall, Englewood Cliffs, N. J., 1975.

18. Moore, F. R., "The Computer Audio Research Laboratory at UCSD", *Computer Music Journal*, Vol. 1, No. 6, 1982, pp. 18-29.

19. Jaffe, D., and Boynton, L., "An Overview of the Sound and Music Kit for the NeXT Computer", *Computer Music Journal*, Vol. 13, No. 2, 1989, pp. 48-55.

20. Lansky, Paul, Princeton Univ., *CMIX*, 1987.

21. Dannenberg, Roger B., and Fraley, Christopher Lee, "Fugue: Composition and Sound Synthesis With Lazy Evaluation and Behavioral Abstraction", *Proceedings of the 1989 International Computer Music Conference*, Computer Music Association, San Francisco, 1989, (to appear)