A LANGUAGE FOR COMPOSITION

Dan Hitt
Computer Science
Stanford University
Stanford, California 94309
USA

Yee On Lo
Computer Science
Stanford University
Stanford, California 94309
USA

ABSTRACT: This is a report on our research in developing a language \( L \) for composition. We are writing for users, implementors, and would-be implementors of languages for musical purposes. Our hope is that our considerations and discussions of language features may be of some benefit to them in their work. We outline some of our goals, and a sample piece with \( L \).

0. Introduction. Our motivation in designing \( L \) is to develop a tool for accomplishing certain compositional tasks in a way that is most easy and natural to us. Our hope is that \( L \) will complement some of the interesting, existing musical languages such as Pla [5], Common Music [7], etc. (just as \( \TeX \) and PostScript complement each other), and that designers of future languages can perhaps get some ideas from our efforts.

1. Goals—Overall. Our overall goal, of which \( L \) is only a part, is to use the computer for more than just a signal processor which generates acoustic events. We want to take advantage of the symbol processing capabilities of the computer for compositional purposes, and we use \( L \) to that end. We are trying to engineer \( L \) to have as many properties as possible of an ideal language. For us, some of these qualities are summarized by:

(i) In an ideal language, the natural hierarchies in a musical composition can be easily exposed and manipulated.

(ii) As much fundamental musical knowledge as possible should be built into the language; it should be prepared to handle cliché practices. (By way of comparison, \( \TeX \) is designed to handle the clichés of paragraph formation—e.g., putting optimal spacing between words.)

(iii) An ideal language should facilitate painless "upgrading" of a composition from one medium to another. E.g., a composition originally designed for realization on MIDI should have a smooth path to a realization with sampled sound.

(iv) An ideal language should make possible "analytic by synthesis": a scholar should be able to write a program to produce a piece of music, say a Bach or a Debussy. The language should make it possible for a short, insightful program to be written (one whose length approached the true Kolmogorov complexity of the piece). Such a program would lend credence to an analysis of a piece in a very concrete way.

(v) An ideal language should be Turing-equivalent to (i.e., as powerful as) the standard general purpose languages. As a practical matter, this means it should support standard constructs such as recursion.

(vi) The data structures of an ideal language should be controllable by other software entities. In particular, it should lend itself to use by programs with artificial intelligence (we are in fact codeveloping \( L \) and a society of agents program so that the former can be a tool of the latter [CH]). Thus it should contain mechanisms that facilitate such as automated analogical reasoning; it should contain built-in provisions for abstraction and elaboration.

(vii) An ideal language should run on a variety of machines, including PDPs.

(viii) Novice computer programmers should be able to easily produce results in an ideal language.

2. Meeting the Goals. Meeting goals (v) and (vi) lead us to embed \( L \) in a host language: and we choose Lisp because of goal (ii): Lisp has many features which can be used to accommodate musical needs, such as its scheme of rational computation built into its arithmetic (musical quantities are often intrinsically fractions as opposed to floating point numbers). This makes it easier to build in a large body of musical circles. To meet goal (vii) we stick to a subset of Lisp and in particular make no use of CLS. Meeting goal (iii)
mean that our data structures should be as device-independent as possible; e.g., changing from one computer to another or to some other form should not require a redefinition of the one in statement (21) below. Meeting goal (vii) requires a body of code which hides some of the chores of Lisp; the sample session below shows what we think should be hidden.

Goal (i) is met by the recursive nature of fragments (defined below) together with the possibility of elaboration and abstraction; the resources of Lisp are also available for creating further structure if desired. As to goal (iv), analysis by synthesis, we have tried to provide the right means of transforming structures so that close to the minimum information must be supplied to create or modify a structure. This, for example, should make it possible to show how common cores of data are expressed in different ways in a piece [Li].

3. A Session with L. L is based on one primitive object, three data structures, and functions for manipulating them. We illustrate their use with a sample session. We start by calling up Lisp, and then loading L. Assuming the master commands for setting up L are in a file with name "L", this would be done by typing

> (load "L"
)(0)

Here, "L" is the prompt from Lisp. The primitive object is the event, which is analogous to what is called a note in some languages. It has attributes of pitch, onset time, amplitude, formal duration, duration, timbre, and effects (a structure from which miscellaneous parameters are hang). The first five of these are generally numbers (although pitch and amplitude may have a non-numerical value to indicate rest or silence). Formal duration is provided for use by reasoning and mauscript software, where as duration is for use in actually generating a duration. An event could be created by calls such as

> (create-event)

(1)

or

> (create-event :pitch 60 a-name)

(2)

or in many other ways. Direct creation of an event would not be the usual route; it is more typical to create events in groups, called fragments (the first of the data structures we've allowed middle-outer to some end). and (3) and (4) would just do a simple entry-point into L. In (2) and (3) a-name (actually a-name) becomes a name for the created event. In (3), the pitch of an event is set to 60, which is our (and MIDI's) convention for middle-C (c4). The pitch (or any other attribute) of a-name could be modified by a statement such as

> (modify-event a-name :pitch 72)

(4)

Statement (4) makes a-name have pitch c5. We make no restriction on the values the pitch attribute can have (i.e., they need not be integral), and they can be converted to frequencies by a simple exponential transformation. Now, the pitch e created in statements (1) and (2) are supplied by the current context (the second of the data structures). L maintains a stack of contexts; the first context is created when L is loaded.

A context is similar to a collection of defaults. These include such things as default pitch, default time, and a default scale (and many others—default timbre, etc.). A scale for us is a pair of functions, an index-to-pitch and a pitch-to-index. (Built-in scales include major and minor, chromatic, and whole tone; users who know a little Lisp can create others as well.) A context thus carries some of the functionality of what is called a part in other languages.

As we mentioned, events are normally created in groups, or fragments: statements (5), (6), and (7) below create a fragment whose representation in standard notation is given by (8):

> (create-list y x 0 1 2 2 1 2 0 4)

(5)

(5) just makes a list whose first element is x (for rest) and whose name a y (or Y); it could equally well be created by the Lisp statement

> (setf y '((0 1 2 3 1 2 0 4))

(6a)

We will transform y into a fragment, but the context needs to be right, so we adjust it:

> (adjust-context

:scale major

:note-duration 1/16

:time-update-mechanism by-increments

(6)

ICMC GLASGOW 1999 PROCEEDINGS

238
:delta-time 1/16
:time 0
:timeing 4/4

Statement (6) sets the context: As long as the context is on top of the context stack, it will govern positions of $z$ (unless it is overridden). The fragment creation is then done by

> (create-fragment $y$ $m$)

which causes $m$ to hold a representation of

If desired, we can modify $m$ by a command such as

> (adjust-fragment $m$
:position 0
:duration double

Positions are counted from 0 at the beginning of a fragment, or from -1 at the end; using either scheme, frag now would hold

We can extend this example to show some more things, including how output is produced:

> (push-context
:origin 4
:note-duration 1/8
:delta-time 1/8
:time 1/2

Statement (12) pushes a new context on the stack, and adjusts the default time information.

> (create-fragment $x$ $l$)

Next, we link the fragments together:

> (link-fragments $m$ $x$ $l$ $n$)

Then we pop back to the original context for creation of a bass line

> (pop-context
:origin -7
:time 1/2

Then we do the bass creation:

> (create-fragment $y$ $b$)

Note that we only had to create the list $y$ only once; this kind of re-use would be typical in many kinds of perceptually organizeable music: something is created once, then used (and modified) many times.

The contents of $n$ are

because of a default governing how fragments are linked (the default is stored in the context, and can be changed if desired—the default is that the head of $m$ is merged with the tail of $n$).

We can now output our work with a few lines:

> (adjust-context
:midi-outfile-name a-file

ICMC GLASGOW 1990 PROCEEDINGS
219
Statement 18 makes a default file name (which can be done or changed any time the context is accessed). Successive mid-dumps will generate files named "o-file.0", "o-file.1", etc.

> (commit-to-canvas b a)

puts both fragments in our third data-structure, the canvas (which is held in the same structure which holds the context stack). The contents of the canvas will be:

Finally, we write to a file

> (write-midi-file)

At this point, we may want to quit L; before doing so we may want to save some of the software objects we've created. We do this with a call to archive, e.g.:

> (archive y b 20)

Statement (22) saves them to disk (where they can be retrieved by calls to unarchive). Note that there is a distinction between writing the binary midi file; the archive command stores an ascii representation of the given objects.

4. Further Features. We can't be exhaustive about L in such a short space, but we want to mention that the language has features for dealing with abstraction and elaboration of fragments. A simple example of elaboration might be to add a modest to an event in a fragment, but in general much more than this can be done. Abstraction works in the opposite direction, and automatic reasoning software in conjunction with L can provide restrictions such as from

5. Conclusion. We have tried to explain our goals for L and the choices these seem to force (i.e., that L be embedded in Common Lisp, but done so with sparing use of the host language's features). We have shown how to use a small subset of the statements available in L and these illustrate the possibility of using L without any knowledge of Lisp beyond what it takes to load a file.

We have not discussed the interface of L to software which performs automated reasoning, for example by analogical means; this will appear elsewhere [LH]. For lack of space, we have not discussed how the design of the language naturally supports structure-oriented timbre composition using methods such as kinematic synthesis [L, L0]. We also have not given a catalogue of the features available, partly because L is still in a nascent stage, but we hope to publish a comprehensive treatment at some point. We solicit comments from interested parties.

6. Acknowledgements. We would like to acknowledge the financial assistance and support of Professor Ullman and the Computer Science Department at Stanford University.

7. References.

ICMC GLASGOW 1990 PROCEEDINGS

240