The Canon Score Language[1]

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

ABSTRACT

Canon is both a notation for musical scores and a programming language. Canon offers a combination of declarative style and a powerful abstraction capability which allows a very high-level notation for sequences of musical events and structures. Transformations are operators that can adjust common parameters such as loudness or duration. Transformations can be nested and time-varying, and their use avoids the problem of having large numbers of explicit parameters. Behavioral abstraction, the concept of making behavior an arbitrary function of the environment, is supported by Canon and extends the usefulness of transformations. A non-real-time implementation of Canon is based on Lisp and produces scores that control MIDI synthesizers.

Introduction

Canon is a computer language designed to help composers create note-level control information for hardware synthesizers or synthesis software. Canon was motivated by my need for a simple yet powerful language for teaching second-semester students of electronic and computer music, and also by my interest in putting concepts from the language Arctic (Dannenberg 1986, Rubine 1987) to practical use in a MIDI studio.

Canon is something of a cross between a music notation and a programming language. The main idea in Canon is to combine and transform simple scores to form more complex ones. Since time is an essential ingredient in music, Canon provides several ways of specifying and manipulating the timing and synchronization aspects of music. Aside from time, various transformations can be performed to affect loudness, pitch, and timbre.

In contrast to note lists, sequencers, and other approaches in which music is represented as data to be operated upon by an editor or other program, Canon scores are themselves programs. This allows scores to be parameterized and to incorporate arbitrary calculations including

---

[1]Published as: Dannenberg, ‘‘The Canon Score Language,’’ ***Computer Music Journal***, 13(1) (Spring 1989), pp. 47-56.

compositional algorithms. The same could be said of languages like Formes (Cointe 1984, Rodet 1984) and Pla (Schottstaedt 1983), but Canon goes further to blur the distinction between data and program. This is possible because Canon is a declarative language. Rather than specifying sequences of statements to manipulate variables and affect the computation, Canon programs are written in a more descriptive style that can be similar in form to note list data.

One of the powers of Canon is its ability to express complex parameterized behaviors. I call this capability *behavioral abstraction* by analogy to the programming language term *procedural abstraction*[2]. The goal of behavioral abstraction is to describe a class of behaviors or performances, instances of which may vary according to context. For example, suppose I want to generate a drum roll sound. This is easily done by repeating a very short note some number of times, say 50. Now suppose I incorporate the drum roll into a score and stretch the score by a factor of two. What happens to the drum roll? If the stretching is applied directly to note list data, the 50 notes of the roll would be stretched, slowing the rate of the roll. With Canon, it is possible to write a drum roll that fills in with more notes to obtain the desired duration when then roll is lengthened. The ability to provide specialized behavior in response to transformations allows the composer to express abstract concepts like ‘‘drum roll’’ which are not tied to a particular set or number of notes. These concepts actually extend the meaning of transformations. A transformation like ‘‘stretch’’ is now operating at a musical concept level rather than at the data manipulation level.

In the next section, the elementary expressions (called score primitives) for creating and combining notes are presented as is the facility for defining new operators. In the following three sections, transformation operators are described. Then, control structures for iteration and conditional evaluation are presented. After some short examples of Canon programs, time-varying transformations are presented. The implementation of Canon is described, and this is followed by a short survey of related work and conclusions.

Score Primitives and Operator Definition

Canon allows the composer/programmer to create, manipulate, and combine scores. A score is usually obtained by combining other scores, but there a few built-in, or primitive scores from which other scores are composed. Before describing these primitives, a brief explanation of Lisp (Kornfeld 1980, Touretzky 1984) syntax (used by Canon) is presented as a guide to readers who are not familiar with Lisp. Lisp programmers should skip ahead to ‘‘Primitive Operators’’.

Lisp Syntax

[2]Virtually every procedural language provides some form of procedural abstraction through the definition of new subroutines or procedures. A subroutine is an abstraction in the sense that it represents a class of actions (procedures); a concrete instance of the abstraction is made each time the subroutine is invoked, and each instance might be created with a different set of parameters.

Canon (and Lisp) have essentially two types of things to notate: *operators* and *operands*[3]. For example, in the expression `(+ 1 2)`, ‘‘`+`’’ is an operator, and ‘‘`1`’’ and ‘‘`2`’’ are operands. (Thus, Canon’s ‘‘`(+ 1 2)`’’ is mathematic’s ‘‘1+2’’.) The syntax used by Canon generalizes to operators that have more or less than two operands. Canon uses the following syntax for *all* of its expressions involving operators:

(*operator operand-1 operand-2 ... the-last-operand*)

The parentheses are important here: they mark the beginning and ending of the expression. This provides a uniform notation regardless of how many operands there are.

Nested expressions are allowed; for example, ‘‘`(* (+ 1 2) 3)`’’ denotes the calculation of ‘‘(1+2)×3’’; the value is therefore 9. Most Canon operators have symbolic names; for example, `note`, `seq`, and `trans` are Canon operators.

Primitive Operators

The `note` operator produces a score with one note. For example, `(note c4)` produces a middle-C. The `note` operator can take more operands. The full list is

(`note` *pitch duration duty velocity channel*)

Pitch is specified by a number, but variables like `C4` are pre-defined for convenience. (The operands are evaluated so that expressions and variables can be used as well as numbers.) *Duration* is specified in hundredths of seconds until the next event, and variables like `q` (for quarter) and `ht` (for half triplet) are pre-defined as well. The *duty* operand is the percentage of *duration* before the note is turned off. Note that *duration* corresponds to the conventional musical definition only if *duty* is considered to be an articulation indication such as *staccato* or *legato*. For example, if *duty* is 50 and *duration* is 150, then the note will be turned on for .75 seconds (50 percent of 150 hundredths) rather than the full duration of 1.5 seconds. (The remaining .75 seconds will consist of the release of the note followed by silence.) *Velocity* and *channel* refer to the corresponding MIDI parameters (Loy 1985). Any operands not specified take on default values. The default note is a middle C lasting 1 second with 100% duty, a velocity of 100, and channel 1.

The `rest` operator has only one operand that specifies duration. For example, `(rest 200)` specifies a rest (silence) of two seconds (again, the unit of time is the centisecond, or 0.01 seconds).

The `adagio` operator refers not to a tempo but to the language Adagio, a note-list format developed by the author. The operand is a filename: `(adagio "perf")` denotes the score stored in the file named ‘‘perf’’. This operator is useful for incorporating a note-list that was captured from a live performance.

The `seq` operator is used to combine notes into sequences or melodies. For example,

---

[3]In abstract algebra, ‘‘operator’’ is a specific type of function. In Lisp, the term ‘‘function’’ is normally used in place of ‘‘operator’’ to imply a less restricted definition. Since programming is not mathematics, neither the term ‘‘function’’ nor ‘‘operator’’, when taken from a mathematical viewpoint, is very accurate. I am using the terms ‘‘operator’’ and ‘‘operand’’ to appeal to the intuition of the non-programmer in this introduction in hopes that Lispers will forgive me.

```
(seq (note d4) (note c4) (note cs4))
```

denotes a score with three notes in sequence. When a sequence is created, the separation between notes is determined by each note's *duration*. The note is sounded some percentage of the duration according to the *duty* factor. A small *duty* factor produces staccato effects while a large *duty*, e.g. 100 or more produces legato effects. Notes will overlap when *duty* is greater than 100.

The `sim` operator (short for ''simultaneous'') is similar to `seq`, but instead of splicing together a set of scores end-to-end, it performs them all at the same time. The following:

```
(sim (note c4) (note e4) (note g4))
```

denotes a C-major chord.

Defining New Operators

Operators can be defined using the same facility provided by Lisp. Consider this example:

```
(defun up ()
 (seq
  (note c4)
  (note d4)
  (note f4)
  (note g4)))
```

This defines the operator `up` to mean the sequence c4, d4, f4, g4. The operator `defun` (short for ''define function'') normally takes three operands:

`(defun` *name operands score*`)`

The first operand is the name of the operation being defined. The next operand is a list of names of operands for the new operator (see the next example). There are no operand names in this example, so the `up` will have no operands. The third operand of `defun` is a score, which is taken as the meaning of the new operator.

Now that `up` is defined, it can be used just like any other operator. For example, a sequence that plays the ascending pitches twice could be written:

```
(seq (up) (up))
```

Note that even though `up` has no operands, it is still enclosed in parentheses to denote that `up` is an operator.

Operands also work as in Lisp. The following example is a rewrite of the previous definition allowing the starting note to be provided as an operand. The effect is to transpose the ascending scale so that it starts on the pitch provided as the operand:

```
(defun up (root)
 (seq
  (note root)
  (note (+ root 2))
  (note (+ root 5))
  (note (+ root 7))))
```

Here, `root` is the *formal* operand (Pratt 1975) for which *actual* operands are substituted when `up` is used. For example, if we were to write `(up a5)`, Canon would substitute the value of `a5` for `root` in the definition of `up` and then evaluate the resulting score.

This example illustrates how operands can work in combination with a definition facility (`defun`) to provide a flexible and extensible score notation. In particular, structures like phrases, chords, or melodies can be defined and used many times in a composition. However, the example also shows that even providing something as simple as transposition is awkward. The new definition has 18 symbols versus 11 in the original, and the single transposition operand had to be propagated into every note expression. If writing scores to be transposable is a moderate chore, imagine the effort to allow a combination of transposing, stretching, sustaining, time-shifting, dynamics-changing, and other common operations! In the next section, transformations are introduced to avoid awkward parameterization of common types of score manipulation.

Transformations

In Canon, a *transformation* is something that alters a score. Scores can be altered by shifting and stretching them in time, by scaling the duty factor of notes, by transposition, by changing the velocity, and by extracting the notes that fall into a particular time interval.

There are a number of transformation operators. Most of these transformations have two operands: a number giving an amount of transformation and a score to apply the transformation to. The score may be another transformation expression, a primitive, the result of a combining operation (`seq` or `sim`), or an operator defined by `defun` as described earlier. Consider the following:

```
(sim (note c4)
     (at 300 (note a4)))
```

The `at` transformation shifts scores in time. Intuitively, `(at` *x y*`)` means ‘‘at time *x*, do *y*’’ and the `at` operator is reminiscent of the ‘‘`@`’’ operator in 4CED (Abbot 1981) and Arctic (Dannenberg 1986). In this example, the time of the second note is shifted by 3 seconds. Notice that even though the `sim` operator causes both of its operands to start at the same time, the second one starts with 3 seconds of silence due to the time shift.

The `stretch` operator stretches time. In the following example it makes the sequence take three times as long as normal:

```
(stretch 3 (seq (note c4)
                (note c5)))
```

The `sustain` operator alters the duty factors of notes. The following shortens notes to 50 percent of their normal values:

```
(sustain 50 (seq (note c4)
                 (note c5 100 50)))
```

The first operand (`50`) is a percentage, so the duty factor of each note is halved. The first note will last 0.5 seconds and the second will last 0.25 seconds[4]. The start times of the notes remain the same and only the durations change when `sustain` is used.

---

[4]The second note has a specified duty operand of 50, so without the transformation, the note would last 50% of its duration of 100, or 50 centiseconds. The `sustain` transformation halves this.

The `trans` operator transposes scores. The following:

```
(seq (up) (trans 2 (up))
          (trans 12 (up)))
```

takes the first `up` operator defined above and plays it at various transpositions. The transposition is in semitones, and may be positive, negative, or even zero. Notice how much easier it is to use `trans` than it is to parameterize `up` to achieve a similar effect.

The `loud` transformation, changes the loudness (velocity) operand of notes. The value is added to the MIDI velocity and the result is limited to the range from 1 to 127 to accommodate MIDI synthesizers. For example,

```
(loud -20 (seq (note c4) (note c5)))
```

lowers the MIDI velocity of these notes from the default of 100 to 80.

The `extract` operator removes notes before a given start time or after a given stop time. Notes that span the start or stop time are truncated to fit within the interval. The operator takes three operands as follows:

(`extract` *start-time stop-time score*)

Here is an example:

```
(seq (up) (extract 50 300 (up)))
```

This expression plays a copy of the score created by ‘‘`(up)`’’ followed by an abbreviated version created with extract. What you hear is the portion of ‘‘`(up)`’’ between time 50 and time 300. Since the first note of ‘‘`(up)`’’ normally lasts one second (the default note duration), the `extract`’ed version starts in the middle of the first note. Since the stop time is 300, we hear only to the end of the third note, the f4.

Nesting Transformations

It is very useful to combine several transformations. Consider the following example:

```
(trans 5 (loud 10 (up))
```

This score plays ‘‘`(up)`’’ with a slight increase in loudness (velocity) and transposes all of the pitches up by 5.

Time shifting interacts in a fairly subtle way with duration scaling. When we scale the duration of something we expect the duration of its components to be scaled, but it is also natural to expect the time between events to be scaled. Thus, duration scaling *must* affect inter-note time. The rule is very simple: the amount of time shift caused by an `at` transformation is scaled by the current duration factor.

Here is an example to illustrate this point:

```
(stretch 3 (at 10 (note 60)))
```

The `at` operation is inside a `stretch` operation, so the note is shifted not by 10 but by 30 (that is, 3×10) time units.

Transformations are normally relative, so in the following example

```
(stretch 2 (stretch 3 (up)))
```

the ‘‘(up)’’ score is stretched by 3 and then again by 2, so the effect is the same as stretching by 6. Stretch and sustain factors multiply, while shift amounts (from at), loudness changes, and transpositions add.

Absolute Transformations

Sometimes one wants to make sections of a score impervious to transformations. Suppose I want to have a quiet section at a certain loudness independent of how loud the rest of the piece is, or perhaps I want a theme to enter at a fixed tempo regardless of how surrounding sections are stretched. Most of the transformation operators have a related operator that fixes a given dimension of the transformation environment and shields it against further transformation. These operators are at-abs, stretch-abs, sustain-abs, trans-abs, extract-abs, and loud-abs. The ‘‘-abs’’ means absolute as opposed to relative. Here are some examples:

```
(at 50 (at-abs 40 (note 60)))
```

The outer at would normally shift the score by 50, but the inner at-abs fixes the time at 40. The note will be played at time 40.

Now, consider

```
(at-abs 50 (at 40 (note 60)))
```

The outer at-abs makes the inner at score occur at time 50, and the inner at then shifts time (relatively) by 40. In this case the note will be played at time 90.

Here is an example using loud-abs:

```
(loud 20 (seq (up)
              (up)
              (loud-abs 0 (note b4))
              (up)))
```

In this example, the ‘‘(note b4)’’ will be played without any change in loudness, and the other operands (copies of ‘‘(up)’’) will have their loudness increased by 20.

Control Structure

Canon has special operators to repeat scores and choose between scores. The operator seqrep repeats a score sequentially. This example plays the note c4 20 times:

```
(seqrep (i 20) (note c4 40 20))
```

The first two operands of the seqrep operator are enclosed in parentheses[5]. The first operand (‘‘i’’ in the example) is a variable name, and the second (‘‘20’’ in the example) is the number of repetitions. The third operand (‘‘(note c4 40 20)’’) is the score to be repeated.

Each time the score is repeated the variable (the first operand) is incremented by one, starting from zero. This fact can be used for some interesting programs. A simple application is a score

[5]This is not standard syntax for operands in general, but it is a form that is commonly used for special control structures in Lisp.

that ‘‘fades out’’ over ten iterations:

```
(seqrep (i 10) (loud (* i -5)
                     (stretch 0.2 (up))))
```

A conditional allows a choice to be made between two scores. Here is an example in which ‘‘`(up)`’’ is played each repetition of a `seqrep` except for the ninth time:

```
(seqrep (i 20) (if (= i 8) (note b4) (up)))
```

The `if` operator always has three operands. The first is a test. In this case, it is the comparison of `i` to the number `8`. This test will be true on the ninth repetition (recall that `i` starts at zero). The second operand gives a score to play if the test is true and the third operand is a score to play if the test is false.

Examples

We can now see how Canon can be used to express an abstract behavior like ‘‘drum roll’’. The implementation will compute how many notes are needed to fill the available time and use `seqrep` to generate the notes:

```
(defun drum-roll ()
  (sim (rest)
       (seqrep (i (truncate (* 10
                               (eval *dur*)
                               (eval *duty*))))
         (stretch-abs 0.1 (sustain-abs 50 (note c4))))))
```

In Canon, the current duration (in seconds) and duty factor (a fraction) are obtained by evaluating the special variables `*dur*` and `*duty*`, respectively. Their product is on-time in seconds. (The need for the `eval` is described in the Implementation section below.) We also multiply by the number of notes per second (10 in this example) to get the number of notes. This is truncated to get an integer:

```
(truncate (* 10 (eval *dur*) (eval *duty*)))
```

All of this yields a repetition count for the `seqrep`. The score to be repeated represents one drum sound. The note is transformed to have an absolute duration of 0.1 seconds and a 50% duty independent of other transformations. The `seqrep` will end after the last drum note, which is not the full duration unless `*duty*` is 100 percent. To correct this, a rest is ‘‘played’’ simultaneously with the drum roll. The rest will have the full duration by default. Since `sim` takes on the maximum duration of any of its components, `drum-roll` will have the desired duration. Now, `drum-roll` can be used with transformations, for example:

```
(sim (up) (stretch 4 (drum-roll))).
```

Canon can also be used to express compositional algorithms. As a very simple example, let us define an operator that cycles through a list of pitches. The number of notes to generate and a list of pitches will be operands, and transformations will apply to each note.

```
(defun cycle (count pitches)
  (seqrep (i count)
    (note (nth (rem i (length pitches)) pitches))))
```

This definition uses a few new lisp operators: `nth` selects the n^{th} item from a list, `rem` returns the remainder of dividing the first operand by the second, and `length` returns the length of a list. The new operator `cycle` can be used to generate complex textures:

```
(sim (stretch 0.3
        (cycle 40 (list e5 ef5 b4 g4 d5 df5))
     (stretch 0.4
        (cycle 30 (list a3 e3 g3 ef3))
     (stretch 0.25
        (cycle 48 (list b3 e4 d4 fs4))))
```

The `list` operator simply forms a list from its operands.

Time-Varying Transformations

Simple transformations like the ones described are quite powerful, but in music, it is often important to make gradual changes over time. This allows one to express familiar concepts such as crescendo and accelerando in addition to less familiar alterations including making notes successively more staccato. These time-varying transformations are expressed in Canon using special expressions in conjunction with transformation operators. For example,

```
(loud (env (0 0) (1000 30) (2000 10)) (myscore))
```

will impose an amplitude envelope over the course of `myscore`. Each pair of numbers in the envelope represents a breakpoint. The envelope starts at zero, increases to 30 over the first 10 seconds and then decreases to 10 after an elapsed time of 20 seconds. (These times are of course shifted and stretched according to any applied transformations.)

When an envelope is specified for a `stretch` operation, the result is fairly difficult to control because durations are calculated according to the current value of the envelope. This leads to a complex interdependency in which the starting times of notes depend upon durations and duration is a function of starting time. A saner way to deal with time and duration is to provide a function from ‘‘score time’’ to ‘‘performance time’’ (Rogers 1980, Jaffe 1985). The corresponding operators are `warp` and `warp-abs`, and the envelope function provides a mapping from score time to real time. Further details are given in the next section. The following example plays a score at a fast tempo followed by a slow one such that the overall duration (500) is unchanged:

```
(warp (warp-env (0 0) (250 100) (500 500))
      (seq (note c4)
           (note a4)
           (note d4)
           (note cs5)
           (note e4)))
```

To illustrate the operation of the `warp` operator, consider the last note of the sequence. Without the transformation, the note would start at time 400 and end at 500. The time-warping function maps 400 to 340 (derived by linear interpolation between breakpoints) and 500 to 500, so the note will actually be performed at time 340 and last 160 centiseconds.

Implementation

Canon is implemented in Lisp.[6] Canon operators are, for the most part, Lisp macros which expand into directly executable Lisp. There is no special Canon interpreter.

A key idea in Canon is that all scores are evaluated in an environment that affects notes and that can be altered by transformations. The existence of the environment allows transformations to be performed without explicit operands. Furthermore, environments are dynamically scoped, which means a transformation applies to all subcomponents of a score, no matter how deeply they are nested.

Because the Lisp used for Canon is statically scoped, dynamic scoping of the environment is simulated using a set of global variables: `*time*`, `*dur*`, `*duty*`, `*transpose*`, `*velocity*`, `*start*`, `*stop*`, and `*warp*`. Transformations are macros that save a component of the environment (e.g. `stretch` saves `*dur*`) modify the environment, evaluate the score, and then restore the original environment.

In the original implementation, the environment values were numbers, and the environment was modified upon entry to a transformation by adding or multiplying to obtain a new number. For example, if `*velocity*` was 60 and `(loud 15 (s))` was entered, then `*velocity*` would be set to 75 while `(s)` was evaluated.

To handle time-varying functions, the environment is generalized to contain an expression which, when evaluated, returns the appropriate value. Since the global value `*time*` always represents the current time, other components of the environment can easily be made functions of time. Under this new implementation, expressions are created when a transformation is entered. Returning to our `(loud 15 (s))` example, `*velocity*` will now be set to the expression `(+ 15 60)` rather than to the number 75. To obtain the current value of velocity, one must evaluate `*velocity*` as an expression. This is easily done by applying the Lisp function `eval`. We will now see how this deferred evaluation can be beneficial.

The function `env`, used to create envelopes for transformations, returns an expression that will evaluate the envelope at the time point specified by `*time*`. The envelope breakpoints are first transformed by the current values of `*dur*`, `*time*`, and `*warp*`. Then, the new breakpoints are made the argument of `interp`, a function for finding the value of the envelope at `*time*`. For example, if `*time*` is 500, `*dur*` is 100, and `*warp*` is `Nil` (i.e. no time mapping) then `(env (0 0) (100 5))` returns `(interp '((500 0) (600 5)))`. If this expression is evaluated when `*time*` is 550, the result will be 2.5.

If `*velocity*` is 60 and we evaluate `(loud (env (0 0) (100 5)) (s))` then the new value for `*velocity*` becomes `(+ (interp '((500 0) (600 5))) 60)`. Thus, a fairly simple *symbolic* representation of the environment allows us to represent and compose time-varying transformations. The cost of `eval` is high, but it is only invoked at

---

[6]The current implementation uses Xlisp, a small interpreted lisp which we have extended to provide access to functions of the CMU MIDI Toolkit. The Xlisp interpreter, Canon, a program editor, and MIDI utilities all run comfortably on a Macintosh computer with 512 KBytes of RAM.

points where the `eval` is really needed, so the overall cost of maintaining the environment is low.

The function `warp-env` differs from `env` in that `env` produces a function from the current real time (after warping) to a value, whereas `warp-env` produces a function from score time (before warping) to real time. This means that `warp-env` does not apply the current mapping in `*warp*` to the envelope breakpoints as does `env`.

How does the environment affect scores? The note operator is a function that computes a note description in accordance with all elements of the current environment. Thus, `note` is a fairly complex function. Once written, however, it can be widely used. In particular, `note` is called to generate each note read from a score file in order to implement the `adagio` operator.

For various implementation reasons, the `note` operator writes note descriptions to a file. No Lisp data structure is ever constructed to represent scores (except for the Canon programs themselves), and for convenience, the Adagio language is used as the output file representation.

The one remaining aspect of implementation is the handling of the operators `seq` and `sim`. By convention, Canon operators return the ending time of the last note or rest and they restore any changes made to the environment during their evaluation. The `seq` operator evaluates each component score in order. After each score is evaluated, `*time*` is set to the end of that score and the next score is evaluated. In keeping with the convention, `seq` restores `*time*` to its original value after all scores have been evaluated and returns the ending time of its last component.

The `sim` operator is even simpler than `seq` because the environment is the same for every component score. The result returned by `sim` is the maximum of the ending times returned by all of its components. In fact, since evaluating operands and returning the largest is the only requirement for `sim`, it is functionally equivalent to the Lisp operator `max`.

Notice that in the implementation, notes are not generated in time order. Instead, notes are generated in the order they are encountered as the Canon program is evaluated. After the score is computed, the resulting notes are read in, sorted, and performed.

Related Work

Canon is directly related to Arctic (Dannenberg 1986), and Canon semantics are almost identical to those of Arctic. The standard data type in Arctic is a time-varying real-valued function rather than a score, so Arctic has a different set of operators. The idea of a transformation environment and the ability to create abstract behaviors is a common theme in the two languages.

Formes (Cointe 1984) has been used to program at both the note and the control function level. Formes programs are less declarative than Canon programs in that Formes objects contain state information that must be initialized and modified as time progresses. This tends to force the programmer to be aware of implementation details. An advantage of the Formes approach is that output is computed in time order, so it is possible to have simultaneous activities that influence one another.

Formula (Anderson 1986) is a real-time system based on the use of multiple tasks for expressing scores and allowing various transformations. One very interesting property of Formula is that time-varying transformations (like Canon's envelopes) can be programmed procedurally and each envelope runs as a separate process. Formula does not have any built-in mechanisms for nesting transformations except for time-deformation analogous to the `warp` operator in Canon.

Pla (Schottstaedt 1983), like Formes, is an object-oriented language, but Pla seems more note-list oriented. A central idea in Pla is the use of objects called voices to interpret note lists. This idea is also used in Greenberg's (1987, 1988) Object LOGO Music Environment (Krakowsky 1986). Neither of these languages supports nested transformations in the sense of Formes or Canon. On the other hand, Formes and Canon do not have built-in facilities for building voices or performers that interpret note-list data. The ''Examples'' section above and in the cited Formes paper show that this is possible, however.

Conclusions

Canon is a simple yet effective language for expressing scores. Canon is particularly useful when scores can be expressed as combinations of parameterized components and various transformations. The most important aspects of Canon are its declarative style, its ability to support nested transformations, and its ability to define abstract behaviors. The declarative style allows programs to resemble structured note lists rather than conventional programs. The focus is on *what* to play *when* and with what *transformations* rather than on manipulating data structures, setting variables, and executing control structures. Nested transformations allow scores to be altered at the level of notes, phrases, and entire works. In general, no extra programming is required to make transformations applicable to composer-defined operators. Abstract behaviors allow the composer to build new operators that respond to transformations ''intelligently,'' for example by increasing the number of notes in a drum roll when the duration is stretched. Abstract behaviors enhance the usefulness of transformations because one can define the effect of transformations on a new operator. Once implemented, the abstract behavior can be used in a variety of situations.

There is currently no interface for Canon except a Lisp editor and a real-time recording program to allow the capture and subsequent manipulation of real-time performances. Although a more graphic view of Canon programs would be nice, this would be difficult to achieve because Canon has the full expressive power of a programming language. Another approach would be to provide an interface between a graphics-oriented music editor and Canon so that at least the top-level structure of a composition could be manipulated graphically (Dannenberg 1986).

The `drum-roll` example illustrates that writing abstract behaviors in Canon is not a trivial task. In this case, Canon's declarative style does not seem to offer much advantage over a procedural approach. In fact, the `drum-roll` code even looks procedural, although one could argue that it is declarative in the formal sense. Perhaps there are ways of making the notation simpler. At least, a few macros might be written to handle common operations such as computing the nominal on-time of the sound.

The abstraction of transformations is an area where Canon might be improved. For example, would it be possible to define a new operator `legato` so that we could get more sustained notes by writing `(legato (myscore))`? It is in fact possible to define `legato` as a Lisp macro, but the definition requires some knowledge of the implementation of Canon. New transformations should be easier to define. Similarly, I have only discussed piece-wise linear envelopes for use in time-varying transformations. Can we define arbitrary functions as envelopes? Again, the answer is yes, but only with a fair understanding of Lisp and the Canon implementation. These functions should be constructed using a notation like that used for scores. This is one of the goals of Arctic, in which most of the ''score primitives'' are functions of time.

Canon was written in Lisp for expedience. The Lisp macro facility is particularly useful for implementing transformations, `seq` and `sim`. It would be difficult to implement Canon in a language like C unless a powerful macro processor or compiler were used to add code for managing the environment. Even then, some run-time expression evaluation would be needed for time-varying transformations. A real-time version of Canon could be achieved by creating a task for each simultaneous component whenever a `sim` operator is encountered. An alternative to tasks, which are often large[7] and slow to create, is to use a calculation tree of objects as in Formes. I am currently pursuing this approach in a real-time implementation of Arctic.

Acknowledgments

An Xlisp (Betz 1986) programming environment for the Apple Macintosh was the enabling condition for the implementation of Canon. David Betz wrote Xlisp which has proved to be virtually bug-free and a model of good coding style. Blair Evans and Robert Joseph modified Xlisp to create a nicer programming environment. Their work incorporated John Maloney's port of the CMU MIDI Toolkit which provides the Xlisp to MIDI interface. Finally, the comments of the referees were particularly helpful in improving this exposition.

References

Abbot, C., 1981. ''The 4CED Program.'' *Computer Music Journal* 5(1):13-33.

Anderson, D., and R. Kuivila, 1986. ''Accurately Timed Generation of Discrete Musical Events.'' *Computer Music Journal* 10(3):48-56.

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

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

[7]Usually, each task requires a contiguous stack. Stacks must be large enough to ensure that they do not overflow.

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

Greenberg, G., 1987. ‘‘Procedural Composition.’’ *Proceedings of the 1987 International Computer Music Conference*. San Francisco: Computer Music Association, pp 25-32.

Greenberg, G., 1988. ‘‘Composing With Performer Objects.’’ *Proceedings of the 1988 International Computer Music Conference*. San Francisco: Computer Music Association, pp 142-149.

Jaffe, D., 1985. ‘‘Ensemble Timing in Computer Music.’’ *Computer Music Journal* 9(4):38-48.

Kornfeld, W., 1980. ‘‘Machine Tongues VII: LISP.’’ *Computer Music Journal* 4(2):6-12.

Krakowsky, P., ed., 1986. *Object LOGO Reference Manual.* Cambridge Massachusetts: Coral Software Corp.

Loy, G., 1985. ‘‘Musicians Make a Standard: The MIDI Phenomenon.’’ *Computer Music Journal* 9(4):8-26.

Pratt, T., 1975. *Programming Languages: Design and Implementation.* Englewood Cliffs: Prentice-Hall.

Rodet, X. and P. Cointe, 1984. ‘‘FORMES: Composition and Scheduling of Processes.’’ *Computer Music Journal* 8(3): 32-50.

Rogers, J., J. Rockstroh, and P. Batstone, 1980. ‘‘Music-Time and Clock-Time Similarities Under Tempo Changes.’’ *Proceedings of the 1980 International Computer Music Conference*. San Francisco: Computer Music Association, pp 404-442.

Rubine, D. and Dannenberg, R., 1987. ‘‘Arctic Programmer’s Manual and Tutorial.’’ Carnegie Mellon Computer Science Department Technical Report CMU-CS-87-110.

Schottstaedt, B., 1983. ‘‘Pla: A Composer’s Idea of a Language.’’ *Computer Music Journal* 7(1):11-20.

Touretzky, D., 1984. *LISP: a gentle introduction to symbolic computation.* New York: Harper & Row.