Common Music: A Music Composition Language in Common Lisp and CLOS
Author(s): Heinrich Taube
Source: *Computer Music Journal,* Vol. 15, No. 2 (Summer, 1991), pp. 21-32
Published by: The MIT Press
Stable URL: https://www.jstor.org/stable/3680913
Accessed: 02-10-2019 01:34 UTC

Heinrich Taube
Center for Computer Research in
Music and Acoustics (CCRMA)
Department of Music
Stanford University
Stanford, California 94305-8180 USA
hkt@ccrma.stanford.edu

Common Music: A Music Composition Language in Common Lisp and CLOS

Common Lisp (Steele 1990) and the Common Lisp Object System (Bobrow 1988) provide an environment uniquely suited to the exploratory and incremental process of musical composition. Common Music is a high-level music composition language built in Common Lisp and CLOS designed to support computer composition in a variety of score formats through a common protocol. Initially prototyped on a Symbolics Lisp machine, the system runs under most Common Lisp environments found on standard computers such as Apple Macintosh and NeXT machines. Common Music is currently offered at CCRMA as part of course work for computer music classes and workshops. It is available through anonymous file transfer (ftp) on the machine ccrma-ftp in the file named /pub/cm.tarfile.Z

Background

Over the past few years there has been a marked increase in the general availability of both hardware and software specifically related to computer composition. What used to be possible only at computer centers such as CCRMA and IRCAM—the development of complex environments capable of supporting serious musical research and exploration—has become possible for composers who are not affiliated with a particular computing center. Common Music grew out of the author's desire to take advantage of this recent trend and to use compositional resources in addition to the Foonly F4 computer and Samson Box synthesizer used at CCRMA. One of the central features of Common Music might be termed "connectivity," the ability to write compositions for different sound synthesis packages without changing the manner in which compositions are described. Common Music has been designed to support the incremental addition of new output formats without making changes to existing code.

Portability is a central aim of Common Music. Having spent the past decade developing a very large library of compositional procedures in the Pla and SAIL programming languages, the author has first-hand experience with the perils of working in nonstandard hardware and software environments. Common Music is implemented entirely in Common Lisp and CLOS, which allows the system to be compatible across a whole host of computer platforms without any modification to the source code at all. In addition, the emerging CLIM (Common Lisp Interface Manager) protocol will allow Common Music in the near future to support a window- and menu-based environment that is compatible over a variety of windowing systems.

The functionality of Common Music grew out of the author's long experience with Pla, a composition language developed by Bill Schottstaedt at CCRMA (Schottstaedt 1983), as well as Leland Smith's SCORE language (Smith 1976). Some of the compositional tools found in Common Music actually began as a library of SAIL code that was linked to Pla through its foreign function interface. A central aim of Common Music has been to provide a certain "backward compatibility" with Pla's most important features, but framed in an open-ended architecture such that the language is powerful enough to be useful for serious compositional work as well as being easily adaptable to the individual needs of different composers. The author considers Pla to be the "intellectual ancestor" to Common Music, and many of the stylistic conventions found in the earlier system are deliberately supported in Common Music. Some conventions that are not found in Pla, such as the pattern generation facilities, are based at least in part on con-

Computer Music Journal, Vol. 15, No. 2, Summer 1991,

structs that are present in Pla or analogous to constructs found in other languages.

System Overview

Common Music consists of two independent software modules. The first is a run-time system that is responsible for generating musical score files. The run-time system consists of a score event scheduling protocol and an extendible class hierarchy of scores, score parts, and events that the scheduler operates on. Common Music provides a number of top-level macros such as `defscorefile` and `with-part` to define a simple high-level interface to the run-time system. The second component of Common Music is a compositional toolbox that includes functions and class descriptions related to the general activity of musical composition. One of the most powerful features of the toolbox is a pattern description language based on a class of object called `item-stream`. The item stream facility is discussed in detail below.

Common Music defines all of its data structures in terms of CLOS class objects, and the procedural interface to the system is defined in terms of CLOS generic functions. Generic functions permit Common Music to present a unified interface to the system's functionality by permitting class-specific behavior to be modularized into "methods." Generic functions are similar in some sense to message passing in other object-oriented languages, except that generic functions in CLOS may dispatch on the type, or class, of more than one argument. Methods on a generic function are said to be applicable if the type, or class, of data supplied as arguments to the generic function meet the requirements of the method. More than one method may be applicable to a given set of argument data. Primary methods may, for example, be combined with `:before`, `:after` or `:around` methods on the same arguments. Since CLOS supports run-time method and class definition, the system is highly adaptable to individual needs, and there is no distinction between system and user data structures and functions. For example, Common Music currently has predefined methods on the generic functions `realize-score` and `scorefile-event` to implement score file output in Pla, Common Lisp Music, Music-N, MusicKit, and MIDIfile syntax. Dynamically extending the system to include a new score file format is a very easy process: one simply adds whatever new methods are necessary to the existing generic functions to implement the new style.

Parts and Score File Definition

Defining New Classes of Parts

Score parts are objects that define the syntax and semantics of output events written to score files. A score part class declares slots and parameters (a parameter is a slot that contributes a value to an output event), used by instances of the class during score file creation. A slot or parameter declared for a part class is, by definition, defined for any subclass or instance of that part. The propagation of slots and parameters from a class to its subclasses is called inheritance. The example shown in Fig. 1 is an abbreviated graph of the class inheritance lattice related to NeXT MusicKit parts. It illustrates that a score part class may inherit slots and parameters from a variety of classes that are "mixed in" to produce the actual set of slots, parameters, and behaviors defined for a terminal class such as `FmlviPoly`.

New classes of score parts may be declared using the `defpart` macro. This macro defines the new part class, analyzes its slot and parameter declarations, and creates a new method for the generic function `scorefile-event` that the scheduler will invoke to evaluate instances of the new class during score file creation. The ability to compute a new `scorefile-event` method for each new part class allows the system to highly optimize output event handling of part parameters and score file events.

The `defpart` macro allows parameters to be declared as `required` (a value must be supplied per output event), `optional` (if no output value is supplied the parameter will be missing from the output event), or `message` (if an output value is supplied it will be preceded by a message string). For the `message` case, the message itself is either specified by the user in the `defpart` declaration or else it is computed from the name of the parameter

Fig. 1. A portion of the class inheritance lattice for the Music Kit score part class `FmViPoly`.

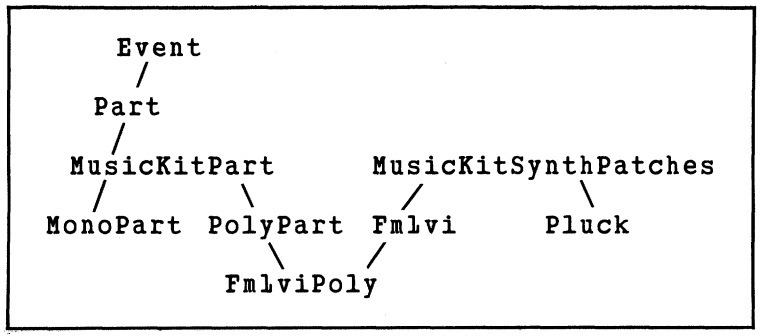

Fig. 2. The declaration of a new class of Common Lisp Music part called `fm`. *The default values of the* `carrier` *and* `modulator` *slots are in a 1:1 ratio.*

```
(defpart fm (clm-part)
         (name time pitch duration &message amplitude index
          ampfn carrier modulator)
  pitch
  duration
  amplitude
  index
  ampfn
  (carrier :initform 1)
  (modulator :initform 1))
```

and the syntax of the new part. The syntax of the output events for a class of part is inherited from the superclass of the part. This can either be one of the predefined part classes—`pla-part`, `clm-part`, `musickit-part`, `csound-part`, `midifile-part`—or a user-defined class that inherits from one of these basic classes. Figure 2 demonstrates the use of `defpart` to declare a new class of score parts.

Score File Definition

Score files are created using the `defscorefile` or `defscore` macros. Both of these macros create an instance of the appropriate class of score to process the descriptions of score part instances defined in the body of the macro. The musical output events that result from processing the part descriptions are directed to a data file associated with the score object. The `defscorefile` macro assumes that all the musical events created by the various part descriptions will be in a single format and contained in a single score file. The `defscore` macro makes no assumptions about either the number of files or formats a score is defined across. The ability to distribute a score over a number of score files is useful if some target program has processing limitations that necessitate construction of the full score in multiple passes, or if several formats are actually required to produce the full composition. For example, a composer who is writing a piece for tape and instruments might want to create all the parts in the same `defscore` in order to share the identical compositional context, even though different performance media are actually involved.

The musical event descriptions contained in a score file are the output events of a collection of score part instances associated with the score. To create a score file, a score object processes its parts in a scheduling queue by evaluating each part at its specified time with the generic function `scorefile-event`. When a part instance is evaluated by `scorefile-event` it is able to de-

termine the content of its next output event and its next run time. If a part indicates that no more output events are forthcoming, the score object will remove that instance from its scheduling queue and invoke an optional clean-up function on the terminated part. When there are no more parts left to execute in the queue, the score is complete and the musical event descriptions in the score file are saved. It should be noted that a score does not need to contain the complete set of parts prior to score file execution. New parts may be dynamically instantiated at any time during the process, and the system will automatically link them to the current scheduling queue. The basic class of parts, called `part`, is very useful for dynamic part creation. An instance of `part` may be scheduled in a score even though its class does not have any score file output defined for it. This enables instances of `part` to be used as schedule-based control structures. For example, a `part` might be used to sprout instances of other classes of parts that actually do contribute output events to the score file. Objects other than parts may be scheduled in the same scheduling queue; all that is necessary is that there be a `scorefile-event` method defined for that class of object. For example, there is a hybrid MIDI score part whose primary `scorefile-event` method executes a "note on" message and then schedules a future MIDI "note off" message in the same queue calculated from the current values of the part's `time` and `duration` slots. Appendix 1 to this article provides a simple example of `defscorefile`.

The Item Stream Facility

Item streams are objects that organize musical data into patterns. Item streams are classified, or typed, with respect to both their content (data type) and their pattern. The data type of an item stream determines how the elements in the stream are to be parsed and interpreted. There are currently six broad classes of abstract data types, `item`, `note`, `interval`, `rhythm`, `amplitude`, and `number`, which are further specialized to 11 different types of item stream data: `item`, `note`, `pitch`, `degree`, `interval`, `step`, `voicing`, `series`, `rhythm`, `amplitude`, and `number`. The pattern type of an item stream determines how the data are to be organized and generated by the stream. There are currently eight predefined, or primitive, pattern types: `sequence`, `cycle`, `heap`, `random`, `palindrome`, `accumulation`, `graph`, and `function`. Composers may use these patterns directly in the organization of musical material or may manipulate them as elements in the construction of larger, arbitrarily complex patterns. Appendix 2 contains several examples of item stream descriptions and the resulting musical output.

Building Patterns with Item Streams

The most powerful feature of the item stream facility is the ability to construct arbitrarily complex patterns out of the small set of basic patterns that are defined by the system. Large patterns are assembled out of the small set of pattern primitives in much the same manner that one might assemble large constructions from a set of building blocks. However, instead of stacking different shapes of blocks one upon the other, the composer embeds different patterns of item streams one inside the other. Every type of item stream permits each element of data to be either an irreducible, atomic element, or an item stream of a compatible data type. When the access function `item` encounters an embedded item stream, it does not return that stream as its value, but rather recurses on the elements of the substream according to the pattern that it implements. Items from the substream will continue to be read until that stream yields control back to the enclosing stream at the end of its current period. The surrounding stream is then free to consider other elements according to the pattern that it implements. Item streams may be nested to any depth, or combined in any order.

The Item Stream Protocol

Because the data and pattern types have been empirically derived over time, the item stream facility permits a composer to extend the basic vocabulary

Fig. 3. Top-level constructor macros implement a simple syntax for defining item streams. They expand into calls to the generic function `make-item-stream`.

```
(pitches A4 EF (pitches B3 B4) C5 in random for 4)

(make-item-stream 'pitch 'random
                  :items
                  (list 'A4 'EF
                        (make-item-stream 'pitch 'cycle
                                          :items
                                          (list 'B3 'B4))
                        'C5)
                  :length 4)
```

of predefined pattern and data types in the system in order to fit more closely his or her own compositional needs. It may be the case that a new basic type will share many characteristics with an existing class, or perhaps simply be a special case of one of the existing classes. In these situations, the new type of item stream may be defined as a subclass of an existing stream and the behavior of the new stream implemented by adding appropriate methods on one or more of the five generic functions that define the item stream protocol: `make-item-stream`, `initialize-instance`, `reinitialize-instance`, `increment-item-stream`, and `item`. Two of these generic functions, `initialize-instance` and `reinitialize-instance`, are part of the standard CLOS object protocol. The other three functions are added by Common Music. The item stream constructor function `make-item-stream` and the item stream reader function `item` are discussed below.

Item Stream Creation

Item streams are created with the generic function `make-item-stream`, which returns an instance of an item stream that implements the proper behavior for the supplied data type, pattern type, and data arguments. In addition, the system defines constructor macros for every data type that support a consistent, English-like syntax for "top-level" item stream creation. These constructor macros are really just "syntactic sugar" and expand into function calls to `make-item-stream` with the proper keyword arguments substituted for the English phrases in the macro body. Figure 3 shows a typical application of the constructor macro `pitches` followed by its expanded form.

Reading Data from Item Streams

The data organized or generated by item streams are accessed, or read, with the generic function `item`. Each call to `item` selects the next datum in a stream as determined by the stream's pattern type and returns it as the primary value of the function call. A secondary value, the state of the item stream as a result of the read operation, is also returned. This second value is useful for making a set of actions dependent on the current state of the stream. For example, an action might be predicated on the fact that a stream has reached the end of its current period, or might be inhibited because the item stream is in the middle of generating a chord. As with all multiple values in Lisp, this second value may simply be ignored if it is not relevant to the caller. In addition to the `item` function, the system provides a set of utilities for manipulating item streams. For example, `read-items` invokes the `item` function on a specified stream for an optional number of times and returns the results in a list. The default number of elements to read is the length of the stream's current period.

Item Stream Periodicity

All item streams exhibit periodicity, which may or may not be reflected in the pattern of the data they return. The period of a stream is marked by a special token, `:end-of-period`, that is returned as the second value of the `item` function if the element returned as the first return value is the last element of the stream's current period. The period length of an item stream is specified when the stream is first created, and (possibly) updated when the stream reinitializes itself at the end of each period. If the period length is specified as a constant number, then the periodicity of the stream will never change. However, if the length is specified as an item stream of numbers, then during reinitialization the length of the next period will be set to whatever value the stream of lengths returns. Even if the period length is specified as a constant value, it may be larger or smaller than the actual number of elements in the stream. Each type of item stream pattern has a default value for its period length. For example, the default period length for `cycle`, `sequence`, and `heap` patterns is the number of elements specified as the data. `Random` patterns have a default period length of a single element. The period length of a stream may be used, among other things, to control the manner in which a pattern type unfolds. Figure 4 demonstrates how the period length of an item stream can be used to generate pattern elements by depth-first or breadth-first enumeration.

The Basic Item Stream Pattern Types

Sequence and Cycle

The `sequence` and `cycle` patterns are the simplest pattern types. Elements of both types are accessed in linear order. Once the last element in a sequence has been reached it will continue to be reselected by the pattern. However, since the last element of the pattern might itself be an item stream of arbitrary complexity, reselecting the last item does not necessarily mean an identical element will be reselected. In contrast to the sequence, the cycle pattern reinitializes itself to the beginning of the data sequence once the last element has been reached. The seamless connection between the last element and the first element causes the cycle to loop continually through its data.

Heap

The `heap` pattern reads an element by randomly selecting it from an unordered collection. The selected element is then removed from the set. When the last element has been read the pattern reinitializes itself back to the full heap again. The order of elements is therefore unpredictable, but the set of elements is exhaustively generated each time one cycles through the items.

Random

The `random` pattern uses a scheme of weighted probability to determine the next item to be read from the pattern. The basic algorithm is controlled by the optional application of several different types of constraints to any or all of the data. By default, each item in the pattern has an equal chance of being selected and may be reselected in direct succession any number of times. This behavior may be modified for any item by using the full form of item specification in which the item is specified together with an optional probability weight and an optional floor and ceiling on the number of times the item may be directly selected (of course, the item itself could be an entire item stream). The weight of the item represents the probability of selection relative to the weights of all the other items. The default weight for each element is 1; if no further constraints are specified, all the elements will have the same basic probability of being selected. The selection process is further controlled through the specification of minimum and maximum constraints on successive reselection. A minimum floor constrains how many direct repetitions of an element must be made before a new element must be selected. The maximum ceiling constrains how many direct repetitions of an element might be made before a new

Fig. 4. Period length can impact the manner in which patterns unfold. This example shows the definition of an item stream that uses default period lengths to implement a "depth first" heap of three note cycles. Two periods from its generated pattern are shown.

```
(notes (notes c3 c4 c5) (notes d3 d4 d5)
       (notes e3 e4 e5) in heap))

==> D3 D4 D5 C3 C4 C5 E3 E4 E5 || C3 C4 C5 E3 E4 E5 D3 D4 D5
```

Setting the period length of each inner cycle to 1 causes a "breadth first" traversal of the inner cycles

```
(notes (notes c3 c4 c5 for 1) (notes d3 d4 d5 for 1)
       (notes e3 e4 e5 for 1) in heap for 9))

==> D3 C3 E3 C4 E4 D4 E5 D5 C5 || C3 E3 D3 D4 E4 C4 D5 C5 E5
```

element must be selected. In addition to these constraints, a single item in the data set may be optionally tagged with an identifier indicating that the associated item will be the initial element selected from the pattern.

Palindrome

The **palindrome** pattern is a variation of **cycle**. The items provided to this pattern type represent a half period of material. When the item stream reaches the last item of data it changes direction and considers the items in reverse order. (In contrast, the **cycle** pattern jumps from the last element directly back to the first element so the flow of elements is always forward.) If the elements in a palindrome pattern are all atomic, a mirror image is formed. However, elements that are embedded inside substreams will still yield their own patterns. The palindrome pattern permits the first and last elements in the pattern to be optionally elided.

Accumulation

The **accumulation** pattern is a variation of **cycle**. Elements are selected by adding them to the series of previous elements that have been selected so far. When the last element has been added to the accumulation, the pattern reinitializes itself back to its starting state.

Graph

The **graph** pattern arranges items in a directed graph. Each item (or item stream) is represented by a node in the graph. Each node in the graph is uniquely identified by an identifier, which defaults to the item itself if an identifier is not supplied. The links from one node to subsequent nodes are specified in a list of **to-node** identifiers. Items are generated by applying a user-specified function to the current node. This function is responsible for returning the identifier of the **to-node** containing the next item to consider in the pattern. If no function is supplied the default selection function works in the following manner. If the node has only one child identifier, it returns the identifier. If the node has more than one child, the function selects one at random. If the node has no children, an error is signaled.

Function

The **function** pattern represents an escape to user-defined, arbitrary pattern description. Unlike all the other patterns described so far, the elements returned by the function pattern are not directly supplied to the stream. Rather, the user specifies a function to the stream. The system will call this function whenever a new period's worth of elements is needed. Common Music defines a **mirror** macro based on this pattern. A **mirror** alternates periods of a pattern with its strict retrograde.

Item Stream Data Types

Item

The `item` type denotes any Lisp object.

Note, Pitch, and Degree

The `note`, `pitch`, and `degree` types correspond to the name (symbol), frequency (floating point number), and position (integer) of the steps in a musical scale. Each type parses the elements of the stream as note references and returns the note attribute specified by the type. For example, the `pitch` data type will accept either name, degree, or pitch references but always returns the frequency of the note reference. All note references in Common Music are made with respect to some scale object. If no scale is specifically indicated, the value of `*standard-scale*` is used to provide a context for the reference. Common Music defines three generic functions, `note`, `pitch`, and `degree`, to provide a mapping between the three types. Composers have the ability to define new scales according to their compositional needs. The system defines three classes of scales: equal-interval scales, gapped scales, and scales with no fixed relation between consecutive notes or consecutive octaves. A scale may have more or less than 12 degrees per octave, and octave tunings may be based on powers other than two. The constructor macro `defscale` evaluates one or more octave declarations to construct the appropriate type of scale based on the number of cents per octave, the name and enharmonic names of each degree, and the number of cents per interval that have been declared.

Interval, Step, and Voicing

The three interval types all parse data elements as interval distances (integers) from a transposition offset. Each interval generated by the stream is combined in some fashion with the offset to determine the actual value returned by the `item` function. The default transposition level is zero, which means that no transposition occurs. By specifying a nonzero transposition level (as a degree, note, or pitch) the interval stream may be shifted to a particular scale degree. The stream may then serve as a template to generate notes, pitches, or degrees relative to the given offset. It is also possible to specify the transposition offset as an item stream of offsets. This allows the transposition level of the stream to evolve over time according to the pattern that the offset stream generates. The ability to specify transposition levels as item streams, and the ability of both note streams and interval streams to generate degrees, means that the different types may be combined in pattern building. Interval streams may be embedded as degree references inside note streams, and note streams may serve as transposition offset generators to interval streams. In addition, there is a special "mixin" to interval streams that enable them to coerce generated degrees to the pitches or notes of some particular scale.

The three types of interval streams are differentiated according to how they generate intervals and how they increment their transposition offsets. The `interval` type is the simplest case. Each call to the `item` function returns a value that is determined by transposing the selected interval to the transposition level specified for that period. It reselects an offset at the beginning of every period. The `step` type returns the current transposition level as its value, and then increments the level by each new interval it generates. As with the interval type the transposition level is reselected once every period. A `step` stream is particularly useful for describing scalar patterns because each interval is treated as a distance relative to whatever the previous value returned by the stream was. The `voicing` type updates the transposition offset every time an interval is read from the stream. The transposition offset stream therefore evolves in parallel with the interval stream itself. This is useful for harmonizing, or voicing, the offset stream. Figure 5 demonstrates the `step` and `voicing` types combined in a single example.

Chords

A `chord` stream interprets either note or interval references as members of a chord. Because chords are written so frequently, there is a special read-time macro that uses the square brackets, `[ ]`, to

Fig. 5. A voicing stream is used to harmonize Messiaen's Mode 2 transposed to Middle C. The chords are randomly selected from three possible chord templates.

```
(voicings [0 3 11][0 2 6 11][0 9 11]
          in random
          upon (steps 1 2 from 'C4 for 8)
          for 8 returning note))

==>[C4 A4 B4][CS4 E4 C5][DS4 F4 A4 D5][E4 CS5 DS5]
   [FS4 A4 F5][G4 A4 CS5 FS5][A4 FS5 GS5][AS4 CS5 A5]
```

create a chording stream out of the delimited elements. For example, a C major chord on middle C may be written `[C4 E G]`. When the generic function `item` reads a member from a chord, the second value of the function call will be the special token `:chording` if there are more members left in the chord, or else `:end-of-period`. The `:chording` state enables a single note to be distinguished from the same note in the context of a chord. This distinction is useful in score definition when a score part would like to inhibit or enable actions based on the fact that a number of separately generated note events occur simultaneously in the score. The system macro `unless-chording`, for example, is based on this distinction and insures that all forms appearing in the body of the macro will be evaluated only if the part is not in the middle of generating a chord.

Rhythm

The `rhythm` type maps logical rhythms to rhythmic values. In this fairly common notation standard, a logical rhythm is notated either by character, `W` (whole note), `H` (half note), `Q` (quarter note), `E` (eighth note), `S` (16th note), or by a numerical equivalent, `4` or `4th`, `8` or `8th`, `32` or `32nd`, and so on. A base rhythm may be modified by preceding it with the triplet character `T`, as in `TQ` or `T32`, and following it with some number of dotted rhythms, as in `Q.` or `64...`. More complicated rhythmic values may be notated as rhythmic expressions using the operators `+` and `-`, as in `W+S` or `TH-32nd`. Every instance of a rhythm stream has its own private tempo factor. A default value for rhythm stream tempo is stored as the value of the global variable `*standard-tempo*`. The initial value for this variable is 60, which corresponds to the metronome marking of 60. Tempo factors may be constant values, item streams, or a special class of interpolating envelopes that allow accelerandi and deaccelerandi to be conveniently expressed.

Amplitude

The `amplitude` type maps a system of logical amplitudes to amplitude values. A logical amplitude is a number between 0 and 1.0 or a symbol from the set: `pppp ppp pp p mp mf f ff fff ffff`. An amplitude value is computed from the weight of the logical amplitude, a minimum and maximum amplitude value, and a power curve. Like rhythm streams, each amplitude stream may have its own settings for the amplitude minimum, maximum, and power. The default values for these parameters are stored in global variables that may be reset by the user. The system provides two macros, `crescendo` and `decrescendo`, that are implemented by cyclic amplitude streams.

The Multiple Item Facility

Common Music provides the capability to extend the item stream protocol to include user-defined composite data types. Once a composite data type, or multiple item, has been defined, the item stream facility is able to manipulate its instances in exactly the same manner that it manipulates instances

of the "standard" types. Multiple items are useful for describing pattern behavior that is simultaneously associated with more than one dimension of interest, or for providing a useful level of musical abstraction. All of the item stream data types discussed in the previous sections have been examples of irreducible, or atomic, types. For example, a particular `note`, `rhythm`, or `number` cannot be destructured into constituent elements. The `defmultiple-item` macro defines new classes of `item` that consist of collections of named elements. Common Music supplies a "destructuring" macro called `multiple-item-bind` that binds variables to component values of a multiple item. This allows element values to be accessed without regard to the way in which multiple items are actually implemented.

Since each element of a multiple item is itself an item, it may stand for either a single item or a stream of items. Elements that are single items are "spread" across all the items contained in elements that are in streams. This allows motivic patterns to be captured. An advantage of describing material in this manner (rather than as separate item streams of material) is that because the different dimensions of data are part of a single item, patterns utilizing random generation will never cause the dimensions to become uncoupled or out of phase.

Related Work

There are a number of music languages that address similar compositional concerns or implement functionality related to that found in Common Music. Common Music grew out of many years of programming experience with Pla (Schottstaedt 1983) and SCORE (Smith 1976) at CCRMA. Pla has strongly influenced the schedule-based design of Common Music's run-time system. Pla, like other score languages, has a built-in notion of streams and cyclic pattern description. A primitive version of some of the functionality of the pattern generation facilities found in Common Music actually grew out of a private library of routines that the author implemented in SAIL and invoked as procedures from Pla. Wherever possible, Common Music has tried to incorporate Pla's syntax for describing musical data. For example, the default scale object used by Common Music allows notes to be referenced in exactly the same way they are in the `NOTE:` construct in Pla, and the system of rhythmic notation is almost identical in both systems. HyperScore (Pope 1987) and the Lisp Kernel (Rahn 1990) are score languages that, like Common Music, implement a common substrate to a variety of score formats. HyperScore also uses a standard object-oriented language for representation of musical structure, and defines a class of objects called EventGenerators that perform a compositional service related to what item streams provide in Common Music. Canon (Dannenberg 1989) is an interesting functional language for score description that also shares some similarity to item streams, particularly in the manner in which it allows patterns to be combined, nested, and transformed. The NeXT Scorefile package (McNabb and Jaffe 1989) is a score language implemented in Common Lisp on the NeXT machine that generates score files for the MusicKit.

Future Directions

The vast majority of the features described in this paper are currently implemented in Common Music. The next step in the development of Common Music will be to design structured editors for part definition, score layout, and event list editing. No interface design has gone into the current system because the first release of the Common Lisp Interface Manager (CLIM) occurred in mid-1990 (as this article was being written). CLIM is a portable window management system with powerful interface design tools similar to those currently found only in the Symbolics Genera environment. CLIM will be the substrate upon which the interface to Common Music is based. Another avenue of development will be to tightly integrate the system with Common Lisp Music, a Lisp-based sound synthesis system that Bill Schottstaedt is currently implementing at CCRMA.

References

Bobrow, D. G., et al. 1988. "Common Lisp Object System Specification." *Sigplan Notices* 23.

Dannenberg, R. B. 1989. "The Canon Score Language." *Computer Music Journal* 13(1): 47–56.

McNabb, M., and D. Jaffe. 1989. "The NeXT Common Lisp Scorefile Package." Unpublished manuscript. Redwood City, California: NeXT Inc.

Pope, S. T. 1987. "A Smalltalk-80-based Music Toolkit." In *Proceedings of the International Computer Music Conference.* San Francisco: Computer Music Association.

Rahn, J. 1990. "The Lisp Kernel: A Portable Environment for Musical Composition." *Computer Music Journal* 14(4): 42–58.

Schottstaedt, B. 1983. "Pla: A Composer's Idea of a Language." *Computer Music Journal* 7(1): 11–21.

Smith, L. 1976. "SCORE: A Musician's Approach to Computer Music." *Journal of the Audio Engineering Society* 20(1).

Steele, G. 1990. *Common Lisp: The Language.* Digital Press.

Appendix 1

```
;;; A sample score file description using defscorefile,
;;; with-part and the fm part class defined in Figure 2.

(in-package 'common-music) ; Use the Common Music package

(defscorefile (pathname "cmj.score"); Output to ~/cmj.score
;; Set the global default tempo of the score to mm 140.
(in-tempo 140)
;; Sprout an fm part at score time 0. The ampFn slot is
;; set to the constant string "RAMP". The pitch,
;; amplitude, rhythm and duration slots are set dynamically
;; inside the body of the with-part.
(with-part fm (time 0 ampfn "RAMP")
  ;; Set the pitch parameter to successive notes selected
  ;; from an item stream of five chords arranged in a
  ;; heap. Kill the part after the 10th chord.
  (setf pitch (item (notes [c4 ef g][cs e a][ds fs as]
                           [e g c5][fs4 a cs5]
                           in heap for 10)
                    :kill t))
  ;; On the first note of each chord, select a new value
  ;; for amplitude, rhythm and duration. The unless-
  ;; chording macro assures that the values remain
  ;; constant for the rest of the notes in the chord.
  (unless-chording
    (setf amplitude (item (items .1 .2 .3 .4 .5
                                 in palindrome)))
    (let ((rhy (item (rhythms e s e s. s.))))
      (setf rhythm rhy)
      (setf duration (* 1.5 rhy)))))

;; A loop construct sprouts four more instances of the fm
;; part class at random time increments. Each time through
;; the loop the variable stream is set to a new item stream
;; combining an interval generator with the current value
;; of the variable offset to produce a transposition
;; template for random note generation.
(loop for begin first 0 then (+ begin (random 5))
          for offset in '(c3 c4 c5 c6)
          do
  (let ((stream (intervals 0 3 5 7 9 in random from offset
                                   returning note)))
    (with-part fm (time begin amplitude .1 duration .5
                   rhythm .15 events 18)
      (setf pitch (item stream))))))
```

Appendix 2

Item Stream Examples

Example 1

In this example the **steps** macro is used to create a random scale of half steps, whole steps and minor thirds transposed to a note stream of offsets. The period length of the step stream is set to eight and the stream is coerced to return note names instead of step positions. Four separate periods of return values are shown below.

```
(steps 1 2 3 in random from (notes c3 fs4)
     for 8 returning note)
```

which generates:

```
Period 1: C3 DS3 F3 GS3 AS3 CS4 DS4 E4
Period 2: FS4 G4 AS4 C5 DS5 F5 GS5 B5
Period 3: C3 CS3 DS3 FS3 GS3 AS3 CS4 DS4
Period 4: FS4 A4 B4 D5 DS5 F5 FS5 A5
```

Example 2.

The **random** pattern shown in the previous example supports a "long form" of item specification that permits various constraints to be placed on the basic random selection process. In this example the selection process has been constrained by adding probability weights and maximum selection limits to be placed on individual items in the stream. The **:weight** constraints state that half steps are four times as likely to be selected as minor thirds and that whole steps are twice as likely to be selected as minor thirds. The **:max** constraints state that no more than three half steps or one minor third may be consecutively selected. This is demonstrated below.

```
(steps (1 :weight 4 :max 3)
     (2 :weight 2) (3 :max 1)
     in random from (notes c3 fs4)
     for 8 returning note)
```

which generates:

```
Period 1: C3 D3 DS3 F3 FS3 G3 GS3 AS3
Period 2: FS4 GS4 B4 CS5 D5 DS5 F5 FS5
Period 3: C3 D3 DS3 E3 F3 G3 A3 AS3
Period 4: FS4 G4 AS4 C5 CS5 DS5 E5 F5
```

Example 3.

The **rhythms** macro creates an item stream that generates either real or integer rhythmic values. In addition to supporting a basic set of rhythmic symbols, it allows more complicated rhythmic expressions to be notated by the + and − operators. This example shows a simple cycle of representative rhythmic symbols and expressions in a metronome tempo of 120. The rhythm values shown below are: quarter, whole, triple-dotted eight, double, triplet thirty-second, long, a whole note tied to a double-dotted sixteenth, and a triplet double minus a sixteenth note.

```
(rhythms q w e... d t32 l w+s.. td-16 tempo 120)
```

which generates:

```
Period 1: 0.5 2.0 0.468 8.0 0.041 4.0 2.218 5.208
```

Example 4.

The **series** macro creates item streams that support serial interval generation. In this example, each period of the interval series will select a new transposition note and row form based on the respective offset and row form streams. The row form stream specifies that the series be generated each period according to a cycle of prime, inversion, retrograde and retrograde-inversion forms. The result is shown below the code example.

```
(series 0 11 1 10 2 9 3 8 4 7 5 6
   from (notes c3 g d4 a in heap)
   forming (items p i r ri) returning note)
```

which results in:

```
Period 1: C3 B3 CS3 AS3 D3 A3 DS3 GS3 E3 G3 F3 FS3
Period 2: A4 AS3 GS4 B3 G4 C4 FS4 CS4 F4 D4 E4 DS4
Period 3: GS4 G4 A4 FS4 AS4 F4 B4 E4 C5 DS4 CS5 D4
Period 4: CS3 D3 C3 DS3 B2 E3 AS2 F3 A2 FS3 GS2 G3
```

Example 5.

Item streams support recursive definition. In this example, three small heaps of notes are arranged in a larger palindrome pattern. This arrangement causes the overall pattern to be predicable while allowing the articulation of the smaller subpatterns to be unpredictable as shown below.

```
(notes (notes c3 d e in heap)
       (notes c4 d e in heap)
       (notes c5 d e in heap)
   in palindrome)
```

which produces:

```
Period 1: E3 C3 D3 E4 D4 C4 D5 E5 C5 D5 C5 E5 E4 D4
          C4 E3 D3 C3
Period 2: D3 E3 C3 C4 D4 E4 D5 C5 E5 D5 E5 C5 C4 D4
          E4 E3 C3 D3
```

Example 6.

Motives may be defined by naming a subpattern stream and then referencing that stream by name using the **#@** accessing form. In this example a cycle references a heap motive named X. The result can be seen below the example.

```
(notes c4 (notes fs2 g af in heap named x)
     d #@x f #@x #@x b)
```

generates:

```
Period 1: C4 FS2 GS2 G2 D4 GS2 G2 FS2 F4 GS2 G2 FS2
          FS2 GS2 G2 B4
Period 2: C4 G2 GS2 FS2 D4 G2 GS2 FS2 F4 G2 FS2 GS2
          FS2 GS2 G2 B4
```