LISP, which was designed as early as 1960 by McCarthy (McCarthy 1960) took a long time to be accepted by the computer music community as a suitable language for expressing their problems and solutions (Boynton et al. 1986, Kornfeld 1980), but has now become the preeminent language for symbolic programming in research fields such as Computational Linguistics and Artificial Intelligence. Computer music, and in particular composition, is another field which can benefit from symbolic computation. This poses the interesting question of why the advantages LISP offers have been neglected for so long. Perhaps it is because the mainstream of computer-music research has been sound generation, which doesn’t involve many symbolic applications. In this field even present-day
technology is barely fast enough, so it is only natural to reject the extra layer of a high-level language in favor of more efficient low-level ones.

However, nowadays there is a renewed interest in computer-music composition, interactive composition systems, user interfaces for programming synthesizers and the use of AI techniques in computer music. Fundamental music research also uses algorithmic models and in all of these fields symbolic computation is pervasive and so the use of languages like FORTRAN or C is absurd. For the people entering the field of LISP programming from a background of these languages, the transition process will be painful. This is not the result of the many parentheses in LISP, but because it is difficult to come to grips with a whole new style of programming (thus a new way of thinking) and to unlearn the old stereotypical solutions. The change will often be made with a kind of "minimal effort" approach which involves using the old programming style in the new language. And indeed looking at the LISP programs emerging from the computer-music community, the imperative style can often be seen between the lines of LISP code. It is a pity to neglect the elegant ways of expressing algorithms in LISP, and doing so will often result in a disappointing performance and maintainability.

In this article I will try to make clear the functional aspects of LISP that cannot be found in the "old" languages. I hope this will result in more "LISPish" LISP programs and will give computer-music composers better techniques to express their personal constructs directly in the form of a working program.

Some Remarks as Introduction

This text comprises many examples. They were constructed for the sake of clarity, which means that they are not intended as a computer-music composition system. They rather show programming techniques that can be used in writing your own. The dialect chosen for the examples is Common LISP, but any LISP dialect with lexical scoping will do. Common LISP is likely to become a widely accepted standard although it yields less elegant programs compared to the purer LISP dialects like Scheme. The examples can be converted to Scheme by deleting the funcalls and function-quotes and adapting the function definition syntax. Readers not familiar with LISP may find the references to good introductory texts in the last paragraph useful. This article starts with a very condensed introduction to the basics of LISP in which the functions needed for the following sections are introduced as examples. Then the functional use of LISP is treated. More advanced topics such as continuations and coroutines will be treated in a subsequent paper.
LISTS AS THE MAIN DATA STRUCTURE

The primary data type of LISP is the list, which is notated as an open parenthesis followed by zero or more elements and a closing parenthesis. To give an example, the following list could be used as a representation for a MIDI note of two time-units' duration, middle C (key number 60) and full amplitude (velocity 1.0).

\((\text{note } 2 \ 60 \ 1)\)

Consider the choice of this primitive musical object here as an arbitrary one: we need just one such object in our examples. Of course the particular choice in a real program should be made on esthetic grounds—it expresses your idea of the atomic musical object and its parameters.

There is one special list, the empty list, notated as \((\ )\) or \(\text{nil}\). The elements of a list can be symbols, called atoms (like note or 60), or they can be lists. Thus a collection of notes can be represented as a list of lists.

\((\text{note } 1 \ 60 \ 1)(\text{note } 2 \ 61 \ .7)(\text{note } 1 \ 55 \ 1)\)

If we wish to express control over the timing structure in the representation, ordering a collection of notes, we could form sequential and parallel structures. This way of specifying the time relations between objects is an alternative to the use of absolute start times found in most composition systems.

\((\text{sequential } (\text{note } 1 \ 60 \ 1)(\text{note } 2 \ 61 \ .7))\)

\((\text{sequential } (\text{parallel } (\text{note } 1 \ 60 \ 1)\n\quad (\text{note } 2 \ 63 \ 1))\n\quad (\text{note } 4 \ 55 \ .7))\)

For a discussion of these time relations see Desain and Honing 1988. A piano-roll notation of the second musical object is given in Example 1. Note that the words like note, sequential, and parallel do not have any intrinsic meaning here, since they are not built-in LISP functions. They are just used as arbitrary symbols, signalling our intention with the data in that list. In Common LISP there are of course other data types available, such as strings, arrays, hash tables, and so on, which sometimes are more appropriate than lists. For the sake of clarity I will not use them here.
Abstraction and Application as Dual Mechanisms

The very heart of any functional programming language consists of a pair of dual constructs, the first of which is called application. It is the action of "calling" a function on one or more arguments. In the example below the function first is applied to a representation of a note (the argument). The syntactical form of an application is the name of the function followed by its arguments, together enclosed by parentheses (prefix notation).

(first '(note 1 60 1)) → note

In the example the arrow (→) points to the result of the evaluation of an expression. A constant data list is preceded by a quote (') to distinguish it from an application. In the example above it prevented the LISP interpreter from recognizing (note 1 60 1) as an application of a function note to the arguments 1, 60 and 1.

There are two selector functions that take lists apart: first and rest. (Their former names, car and cdr, are considered obsolete.) There is one constructor function for building lists, called cons.

(rest '(note 1 60 1)) → (1 60 1)
(first (rest '(note 1 60 1))) → 1
(cons 'note '(1 60 1)) → (note 1 60 1)
Other functions for the construction of lists are also supplied as LISP primitives (e.g. append, which concatenates several lists, and list, which builds a list out of several elements at once), but they could be written by the programmer using only cons.

\[(\text{append } '(\text{note 1}) '(60 1)) \rightarrow (\text{note 1 60 1})\]
\[(\text{list } '\text{note 1 60 1}) \rightarrow (\text{note 1 60 1})\]
\[(\text{cons } '\text{note } (\text{cons 1 (cons 60 (cons 1 nil))})) \rightarrow (\text{note 1 60 1})\]

The second central construct is called functional abstraction and transforms an expression (a piece of program text) into a function. Our first function definition will define (by means of defun) a constant function without arguments called example that will just return, as a result, a simple constant musical structure. We will use this function a lot in the following paragraphs.

\[(\text{defun example () \{'(sequential (parallel (note 1 60 1)) \tab (note 2 63 1)) \tab (note 4 55 .7))\}}\]

\[(\text{example}) \rightarrow (\text{sequential (parallel (note 1 60 1)) \tab (note 2 63 1)) \tab (note 4 55 .7))\]

Moving on, we will define functions that have one argument and select a specific element out of a list.

\[(\text{defun second-element (list) \tab (first (rest list))})\]
\[(\text{defun third-element (list) \tab (first (rest (rest list))})\]
Now we can introduce some selector functions that take a note representation apart and a constructor function that builds one.

(defun duration-of-note (note)
  (second-element note))

(defun pitch-of-note (note)
  (third-element note))

(defun velocity-of-note (note)
  (fourth-element note))

(defun make-note (duration pitch velocity)
  (list 'note duration pitch velocity))

(duration-of-note '(note 3 60 1)) → 3
(pitch-of-note '(note 3 60 1)) → 60
(make-note 3 60 1) → (note 3 60 1)

In the first example above a function called “duration” is defined which has one parameter called note. Its body is the application of the function second-element on this parameter. In the definition the parameter note is said to be abstracted from the body. Only when duration-of-note is applied to an actual argument does the body becomes “concrete” in the sense that it “knows” what the parameter note stands for, so that its value can be calculated. In the same way we could program a set of selector and constructor functions as a data abstraction layer for sequential and parallel structures.
(defun make-parallel (structure-1 structure-2)
  (list 'parallel structure-1 structure-2))

(defun structure-1-of (complex-structure)
  (second-element complex-structure))

(defun structure-2-of (complex-structure)
  (third-element complex-structure))

(defun structural-type-of (complex-structure)
  (first complex-structure))

(structural-type-of (example)) → sequential
(structure-2-of (example)) → (note 4 55 .7)

The use of complex data structures (sequential and parallel) of two components does not make them less general in use because they can always be nested:

(sequential (sequential (note 1 60 1)
                      (note 2 61 1))
           (note 163 1))

However, Common LISP provides means of control for passing arguments to functions and we could use the so called \texttt{lambda-list keyword \&rest} to signal LISP to collect all the arguments of the function in a list.

(defun make-sequential (&rest elements)
  (cons 'sequential elements))

(defun make-parallel (&rest elements)
  (cons 'parallel elements))

(make-sequential (make-note 1 60 1)
                  (make-note 1 62 1)
                  (make-note 163 1)) →
(sequential (note 1 60 1)(note 1 62 1)(note 163 1))
Also, optional parameters to a function, that will be assigned a default value when missing in the function call, can be defined by means of the &optional lambda-list keyword.

```
(defun make-note (&optional (duration 1) (pitch 60) (loudness 1))
  (list 'note duration pitch loudness))

(make-note 2 61) → (note 2 61 1)
(make-note) → (note 160 1)
```

For the sake of clarity we will use only the make-sequential and make-parallel functions which have two arguments and the make-note function which has three arguments. Using the data abstraction layer provided by the selector and constructor functions for notes to implement transformations on notes produces very readable LISP code.

```
(defun transpose-pitch (pitch interval)
  (+ pitch interval))

(defun mirror-pitch (pitch center)
  (- center (- pitch center)))

(defun transpose-note (note interval)
  (make-note (duration-of-note note)
    (transpose-pitch (pitch-of-note note) interval)
    (velocity-of-note note)))

(defun mirror-note (note center)
  (make-note (duration-of-note note)
    (mirror-pitch (pitch-of-note note) center)
    (velocity-of-note note)))

(defun transpose-note-semitone (note)
  (transpose-note note 1))

(defun mirror-note-around-middle-c (note)
  (mirror-note note 60))
```
Note that we first defined some general note-transforming functions and then used these in turn to define the dedicated ones required in the following chapter. The utility functions for the pitch arithmetic isolate the calculation of pitches from the note-transforming functions.

Application should be used as the only means to pass information to a function. This ensures the behavior of functions is not dependent upon the context of its call. This obviates the use of global variables. They are to be used only in cases where they represent truly constant values.

Recursion as Main Control Structure

From the beginning the use of recursion in LISP programs was pervasive. Consider the transposition of a collection, i.e. a list, of notes. When the list is empty the task is simple: the empty list has to be returned. Otherwise we cons the transposition of the first note in the result of transposing the rest of the list. The function required for transposing this smaller list is precisely the function we are writing at this moment, so it only has to call itself. This process of self-reference is called recursion.

(defun transpose-note-list-semitone (notes)
  (when notes
    (cons (transpose-note-semitone (first notes))
      (transpose-note-list-semitone (rest notes)))))

(transpose-note-list-semitone '((note 160 1) (note 2 59 .7)(note 165 .7))) →
  ((note 161 1) (note 2 60 .7)(note 166 .7))

This simple form of recursion (called tail recursion) is recognized by any reasonable compiler, and internally transformed into plain iteration, thus overcoming the overhead usually associated with recursion, i.e. extra function calls and increased stack space. In LISP the empty list: () and the truth value false are defined equivalent and called nil. Conversely, everything
that is not nil is considered as true: t. This is used here in two ways. Firstly, the condition in the when clause will only be considered true when the note list is not empty, when there are notes left. Secondly, the result of the when clause when the condition is false will be nil, and this will function as the empty starting list for consing in transposed notes.

While newly designed languages accepted recursion as a control structure, LISP was augmented with "down-to-earth" and well-known iterative control structures, since it was recognized that in some cases these are simpler for humans to use than recursion. For complex cases however, the recursive form is often the more elegant and easier to read. For example, if we wish to define a transposition on a complex musical structure (built from parallel and sequential) we must first dispatch on the type of structure (using a case construct) and then apply the transformation recursively on the component structures, and finally reassemble the transposed parts into their parallel or sequential order. The resultant program would look very messy when written iteratively.

```
(defun transpose-semitone (structure)
  (case (structural-type-of structure)
    (note (transpose-note-semitone structure))
    (sequential (make-sequential
                   (transpose-semitone (structure-1-of structure))
                   (transpose-semitone (structure-2-of structure))))
    (parallel (make-parallel
               (transpose-semitone (structure-1-of structure))
               (transpose-semitone (structure-2-of structure)))))

(transpose-semitone (example))→
(sequential (parallel (note 1611)
                  (note 2 64 1))
               (note 4 56 .7)))
```

case will evaluate its first argument, then select a subsequent clause starting with that value, followed by an evaluation of the second part of that clause. In general it can be said that recursion is the natural control structure for hierarchical data. And hierarchical structures are common in music.
FUNCTIONS AS FIRST-CLASS OBJECTS

In any good programming language all possible objects are allowed to appear in all possible constructs: they are all first-class citizens. However, in many programming languages this rule is often violated. In LISP even exotic objects such as functions can be passed as an argument to a function (in an application construct) or yielded as a result from a function. At first sight this may not seem unusual. PASCAL, for example, allows the name of a procedure to be passed to another one using an ad hoc construction. And in C, pointers to functions can be passed around. However, in LISP all functions are uniformly considered as data objects in their own right, and functions operating on them can be used. This provides an abstraction level that is really a necessity but that is lacking in many other languages. For composers of computer music it is quite natural to think in terms of abstract transformations on objects like time-mappings, which are functions themselves.

FUNCTIONS AS ARGUMENTS

Suppose we want to write a function mirror-around-middle-c which would look similar to transpose-semitone defined above but only uses mirror-note-around-middle-c instead of transpose-note-semitone as the bottom level transformation. Instead of just writing a new function for that purpose, it is better to abstract from the bottom transformation and write a general transform function. This function is now given the note transformation as an extra functional argument, which enables it to deal with all kinds of note transformations. Wherever it needs the result of the application of the note transformation function to a specific note, it calculates that with the LISP funcall construct.
(defun transform (structure transform-note)
  (case (structural-type-of structure)
    (note       (funcall transform-note structure))
    (sequential (make-sequential
                  (transform (structure-1-of structure)
                              transform-note)
                  (transform (structure-2-of structure)
                              transform-note)))
    (parallel   (make-parallel
                  (transform (structure-1-of structure)
                              transform-note)
                  (transform (structure-2-of structure)
                              transform-note))))

(transform '(sequential (note 160 1) (note 2 63 1))
            #'transpose-note-semitone)
  (sequential (note 1611) (note 2 64 1))

(transform '(sequential (note 160 1) (note 2 63 1))
            #'mirror-note-around-middle-c)
  (sequential (note 160 1) (note 2 57 1))

Note the use of the #' construct (called the function quote), which is used to signal to LISP that the following expression is to be considered as a function.

The possibilities of the chosen representation of musical objects and transformations on it are illustrated by the following example. Here each note is transformed into a sequence of two notes with half the original duration. This transformation, called double-note, is built from two other transformations. The first one, half-note, divides the duration of its argument by two. The second, twice-note, makes a sequence of two identical copies of its argument.

(defun twice-note (note)
  (make-sequential note note))
(defun half-note (note)
  (make-note (/ (duration-of-note note) 2.0)
    (pitch-of-note note)
    (velocity-of-note note)))

(defun double-note (note)
  (twice-note (half-note note)))

(transform (example) #'double-note)

(sequential
  (parallel (sequential (note .5 601) (note .5 601))
    (sequential (note 163 1) (note 163 1))
    (sequential (note 2 55 .7) (note 2 55 .7))))

The use of functions as arguments (so-called **downward funargs**) seems to
give so much extra power that we might begin to wonder what good the
passing of functions as results (so-called **upward funargs**) could give us.

**FUNCTIONS AS RESULTS**

If we wanted to apply an octave transposition to a structure we would have
to write a new function, `transpose-note-octave`, and use it as an argu-
ment for `transform`.

(defun transpose-note-octave (note)
  (transpose-note note 12))

(transform (example) #'transpose-note-octave)

(sequential (parallel (note 172 1) (note 2 75 1))
  (note 4 67 .7)))

This means we always have to define the possible transformations in
advance. This is not satisfactory and instead we could use **anonymous**
functions as an argument to the `transform` function. Anonymous functions
are not as bad as they look. They are merely a consequence of the rule of
first class citizens. For example, it is perfectly normal for objects like
numbers, lists, and strings to have a notation for the constant values.
Functions should also have this property. The anonymous function of one
argument that will transpose a note by an octave is notated like this:
(lambda (note)
  (transpose-note note 12))

This kind of function can be used as argument to the transpose function (remember the function-quote).

(transform (example)
  #'(lambda(note)
      (transpose-note note 12))→
  (sequential (parallel (note 172 1)
                       (note 275 1))
      (note 467 .7))

(transform (example)
  #'(lambda(note)
      (mirror-note note 72))→
  (sequential (parallel (note 184 1)
                       (note 2811)
      (note 489 .7)))

Still, this is a little tedious to do, and we define a function transpose-note-transform that will calculate these transposition functions when given the correct number of semitones.

(defun transpose-note-transform (interval)
  #'(lambda (note) (transpose-note note interval)))

(defun mirror-note-transform (center)
  #'(lambda (note) (mirror-note note center)))

(transform (example) (transpose-note-transform 2))→
  (sequential (parallel (note 162 1)
                       (note 265 1))
      (note 457 .7)))
(transform (example) (mirror-note-transform 67)) →
(sequential (parallel (note 174 1)
  (note 2 711))
  (note 4 79 .7)))

GENERALITY AS AIM

Functional programming makes it possible to construct very general programs that are customized for specific purposes. These are the tools that are badly needed in software design. They deserve to be supplied in software libraries, so that programmers can stop reinventing the wheel each time. As a tool for composers these programs may aim to be as "empty" as possible, only expressing general abstract knowledge about musical structure and leaving open details about the specific material and relations used.

The transformations we have so far designed are not yet that general. They were structure preserving, and thus a transformation of a sequence would always yield a sequence. Only at the note level could the structure expand into a bigger one (e.g. when using double-note). Bearing this in mind, we are going to develop a more general transformation device. Our new transformation is like the old one, except that it takes two more arguments to calculate what a sequential and what a parallel structure transforms into.

(defun transform
  (structure sequential-transform parallel-transform note-transform)
  (case (structural-type-of structure)
    (note (funcall note-transform structure))
    (sequential (funcall sequential-transform
      (transform (structure-1-of structure)
        sequential-transform
        parallel-transform
        note-transform)
      (transform (structure-2-of structure)
        sequential-transform
        parallel-transform
        note-transform))))
After this rather tedious definition we have available a very powerful transformational device. Let me first give a rather stupid example of its use: the no-transform transformation function which just rebuilds its argument.

```lisp
(defun no-note-transform (note) note)
(defun no-transform (structure)
  (transform structure
    #'make-sequential
    #'make-parallel
    #'no-note-transform))
(no-transform (example))→
(sequential (parallel (note 160 1)
    (note 2 63 1))
  (note 4 55 .7))
```

The results passed upward by no-note-transform, make-parallel, and make-sequential are musical structures (see Example 2). Using the same idea, but substituting the identity note transformation by another one, gives us the second example. It supports all structure-preserving transformations.
(defun our-old-transform (structure transform-note)
  (transform structure
    #'make-sequential
    #'make-parallel
    transform-note))

Now some more useful transformations can be constructed. The results passed upward will be numbers. The first transformation calculates the duration of a complex musical structure by adding or maximizing the duration of substructures. Similarly it is possible to calculate the duration of the longest note, the number of notes in a piece, and the maximum number of parallel voices of a complex structure. Note how easily the transform function is adapted to these different purposes by "plugging in" different functional arguments (see Example 2).

(defun duration-of (structure)
  (transform structure #'+ #'max #'duration-of-note))

(defun longest-note-duration (structure)
  (transform structure #'max #'max #'duration-of-note))

(defun count-one (note) 1)

(defun number-of-notes (structure)
  (transform structure #'+ #'+ #'count-one))

(defun max-number-of-parallel-voices (structure)
  (transform structure #'max #'+ #'count-one))

(duration-of (example)) → 6
(longest-note-duration (example)) → 4
(number-of-notes (example)) → 3
(max-number-of-parallel-voices (example)) → 2

To demonstrate again the generality of the transform, we will now write a program to draw a piano-roll notation of a musical structure as shown in Example 1. This program was inspired by a tree-drawing program of Joop Ringelberg. To draw a note at the correct place we need to know the
EXAMPLE 2: TRANSFORMATIONS OF THE EXAMPLE STRUCTURE

absolute start time of a musical object, information that the transform function itself does not supply. When context information is missing, it is a well-known trick in AI programming to calculate a function of the (not yet known) context as temporary result. Indeed such a solution is possible here. The draw-note function can return a function that will draw a graphical representation of a note when given a start time. As the drawing is done as a side effect, this function can then return the end time of the note as context to use in further drawing. The draw-sequential function just receives two such draw functions as arguments and constructs the draw function that will pass its start time to the first and pass the end time returned by the first to the second, returning its end time as the result. The function draw-parallel will pass its start time to both substructure draw functions returning the maximum end time they return. Thus neither numbers nor musical structures are passed upward as result of the transformation on substructures, but functions that can draw the substructure
when given a start time. At the top level we will just have to apply the draw function resulting from the call to transform to time 0. An alternative draw-note function is given for a quick test of the code without implementing the graphic procedure draw-horizontal-block.

```
(defun draw (structure)
  (funcall (transform structure
             #'draw-sequential
             #'draw-parallel
             #'draw-note)
           0))

(defun draw-note (note)
  #'(lambda (time)
      (draw-horizontal-block time ; assumed graphical primitive
       (pitch-of-note note) ; left x-position
       (duration-of-note note) ; y-position
       (velocity-of-note note)); width
       (+ time (duration-of-note note)))) ; end time

(defun draw-note (note) ; alternative for testing without
  ; graphics
  #'(lambda (time)
      (print (list 'time time
                   'pitch (pitch-of-note note)
                   'duration (duration-of-note note)
                   'velocity (velocity-of-note note)))
      (+ time (duration-of-note note))))

(defun draw-sequential (a b)
  #'(lambda (time)
      (funcall b (funcall a time))))

(defun draw-parallel (a b)
  #'(lambda (time)
      (max (funcall b time)
           (funcall a time))))

(draw (example)) → '''example 1''
```

Having shown this general solution for dealing with context information, it is clear that this will not be always the best solution. When information
like start time is used a lot, it may be simpler to adapt the transform function itself so that it passes this information as well to the note transformation function:

```lisp
(defun time-transform
  (structure
    sequential-transform parallel-transform note-transform
    &optional (time 0))
  (case ( structural-type-of structure)
    (note (funcall note-transform structure time))
    (sequential
      (funcall sequential-transform
        (time-transform (structure-l-of structure)
          sequential-transform parallel-transform note-transform time)
        (time-transform (structure-2-of structure)
          sequential-transform parallel-transform note-transform
          (+ time (duration-of (structure-1-of structure))))))
    (parallel
      (funcall parallel-transform
        (time-transform (structure-1-of structure)
          sequential-transform parallel-transform note-transform time)
        (time-transform (structure-2-of structure)
          sequential-transform parallel-transform note-transform time))))

The time is made into an optional parameter to facilitate the omission of the start time zero in the call of time-transform at the top level. We now can build transformations, such as a fade-out (decrescendo), that are time dependent.
(defun fade-out-transform (begin end)
  #'(lambda (note time)
      (if (< begin time end)
          (make-note (duration-of-note note)
                      (pitch-of-note note)
                      (* (velocity-of-note note)
                         (- 1.0 (/ (- time begin)
                                     (- end begin))))
          note)))

(time-transform (make-sequential (example) (example))
  #'make-sequential
  #'make-parallel
  (fade-out-transform 0 10)) →
(sequential (sequential (parallel (note 160 1) (note 2 63 1))
             (note 4 55 .56))
  (sequential (parallel (note 1 60 .4) (note 2 63 .4))
             (note 4 55 .14)))

Sometimes we wish to transform our musical objects to note lists where each note has as an extra first parameter an absolute start time, e.g. to play them using a system like John Rahn’s LISP Kernel (Rahn 1988, 1990). We can do that now easily. All that is required is a function to transform a note to a list of one note in the new format, and a function that will merge two sorted parallel note lists. An alternative for this function using the Common LISP merge primitive is given as well.

(defun add-absolute-start-time (note time)
  (list (list 'note
              time
              (duration-of-note note)
              (pitch-of-note note)
              (velocity-of-note note))))
(defun merge-note-lists (list-1 list-2)
  (cond ((null list-1) list-2)
        ((null list-2) list-1)
        ((< (second-element (first list-1))
            (second-element (first list-2)))
            (cons (first list-1)
                  (merge-note-lists (rest list-1) list-2)))
        (t (cons (first list-2)
                 (merge-note-lists list-1 (rest list-2)))))))

(defun merge-note-lists (list-1 list-2) ; alternative
  (merge 'list list-1 list-2 #'< :key #'second-element))

(defun musical-object-to-note-list (object)
  (time-transform object
                  #'append
                  #'merge-note-lists
                  #'add-absolute-start-time))

(musical-object-to-note-list (example)) →
'((note 0 1 60 1)(note 0 2 63 1)(note 2 4 55 .7))

COMBINATORS AS FUNCTION BUILDERS

Since it turned out to be so useful to be able to talk about functions as objects which are passed to and from other functions, we are now going to examine the possibilities of a special kind of these "other" functions, called **combinators.** A combinator is a higher order function that has only functions as arguments and returns a function as a result. The first one we will show is the combinator called twice. It can double the action of any function, and therefore (twice #'rest) will be a function that removes the first two elements from a list, (twice #'double-note) will yield a four-fold splitting of notes, and (twice #'mirror-note-around-middle-c) will be an absolutely useless transformation (the identical transformation).

(funcall (twice #'rest) '(1 2 3 4 5)) → (3 4 5)
(our-old-transform (example) (twice #'transpose-note-octave)) →
(sequential (parallel (note 1 84 1)
    (note 2 87 1))
    (note 4 79 .7)))

Here is the definition of twice.

(defun twice (transform)
  #'(lambda(object) (funcall transform
    (funcall transform object)))))

The second combinator is the "function-composition" combinator. It can combine the actions of two transformations into a new one. It is important not to confuse function composition and musical composition.

(defun compose (transform-1 transform-2)
  #'(lambda(object)
    (funcall transform-1 (funcall transform-2 object)))))

(funcall (compose #'first #'rest) '(c d e f g a b)) → d

To construct a transformation that is a doubling applied to the result of an octave transposition we could use this combinator to build it.

(our-old-transform
  (example)
  (compose #'double-note #'transpose-note-octave)) →
(sequential
  (parallel (sequential (note .5 72 1)(note .5 72 1))
    (sequential (note 1 75 1)(note 1 75 1))
    (sequential (note 2 67 .7)(note 2 67 .7)))

To show the usefulness of these constructions we will write a function that calculates a complex melody from a simple one by adding a parallel melody that is the doubling of the original one transposed one octave. This is a transformation often used in Javanese Gamelan music. The score of the add-doubled transformation on the example object is shown in Example 3.
(defun add-doubled (structure)
  (make-parallel
   structure
   (our-old-transform structure
     (compose #'double-note
               #'transpose-note-octave)))))

EXAMPLE 3: RESULT OF THE ADD-DOUBLED TRANSFORMATION

Note that we could have defined twice as a composition of a transform with itself.

(defun twice (transform)
  (compose transform transform))

PARAMETERS AS SUPERFLUOUS

When defun defines a function, it creates an anonymous function using the argument list and the body, and stores it in the function cell of the symbol that is the name of the function. This means that the next two expressions are equivalent.
(defun duration-of-note (note)
  (second-element note))

(setf ; assign to the function slot
  (symbol-function 'duration-of-note); of the symbol
  #'(lambda(note) ; duration-of-note
    (second-element note))) ; this anonymous function

defun can be considered as a device that makes the definition of a function
casier to read, but it assumes the name of the function and the function
body itself to be constants. Let us first make a similar construct that gives a
little bit more power than defun, and then use it to define duration-of
again.

(defun define-function (name function)
  (setf (symbol-function name) function))

(define-function 'duration-of-note
  #'(lambda(note)
    (second-element note)))

Now we can calculate a function body instead of using a constant
anonymous function. In the first example below the calculation merely
finds the function definition of a symbol, to create a synonymous function.
In the second it is really calculated by one of the transformations we defined
above.

(define-function 'premier (symbol-function 'first))

(premier '(1 2 3)) → 1

(define-function 'transpose-note-octave
  (transpose-note-transform 12))

(transpose-note-octave '((note 1 60 1)) → (note 1 72 1)

Note that in the definitions above, the formal parameters which
appeared in the argument list of the defun form are no longer needed. If we
have access to enough combinators like twice and compose, we can even do
completely without parameters.
(define-function 'double-note (compose #'twice-note #'half-note))

(define-function 'double-and-raise-note
    (compose #'double-note #'transpose-note-octave))

(define-function 'four-times (twice #'twice))

(define-function 'but-first-four (four-times #'rest))

(but-first-four '(c d e f g a b)) → (g a b)

Note how twice is applied to itself to yield the four-times function.

Languages built on combinators using parameter-free programming are very useful in domains centered around one type of object and many transformations on this object (like the musical structures in our examples). In these domains they facilitate the definition of higher levels of abstraction whereby transformations are considered objects in their own right, so that they can be manipulated, combined, and modified. However, when dealing with functions of many arguments we need a lot of combinators for juggling with the order of arguments, leading to programs that are difficult to read. For humans, an extra hook into our memory, by means of a mnemonic parameter name, is often indispensable. In addition, more heterogeneous domains consisting of different sorts of objects, all subjected to transformations that are conceptually more or less the same, can be modelled better using another style of programming in LISP. In this style, named object-oriented programming, it is straightforward to express, e.g., how both a melody and a synthesizer can have their own definition of "transposition."

Some Words as Conclusion

Exaggerating my standpoint, I will give a summary of a "good" style of LISP programming.

1. No function is longer than five lines.
2. Programs are written functionally.
3. No global variables are used.
4. Names of functions and parameters are long and descriptive.
5. Flow of control is done with recursion.
Such a clean and elegant style will result in programs that are easy to construct and maintain. One advantage that languages like FORTRAN and C still have when compared to LISP is their speed. Luckily good industrial Common LISP compilers make the difference quite small.

FRIENDS AS GUINEA PIGS

I am very grateful that the following friends volunteered to read and comment on the first draft of this paper: Paul Berg, Edwin Bos, Wim Claassen, Jim Grant, Margriet Hoenderdos, Koen De Smedt, John Rahn, Joop Ringelberg, Dick Rijken, Huub van Thienen and Theo Vosse, and especially Henkjan Honing because of his stimulating attitude and good ideas.

I would like to thank the colleagues and students of the Language Technology group of the University of Nijmegen and the Centre for Art and Media Technology in Utrecht and for the stimulating environment they provided while I was working on this article.

LISP AS A BOOK TOPIC

Since there are so many books on LISP it is difficult to select the one that will be of most use. A quick rule of thumb that can be used is: if a book on LISP starts by explaining SETQ (or SETF) it is unsuitable.

Abelson & Sussman 1985 is an almost perfect introduction to all the wonders of LISP programming and programming in general. There are extensive examples, but it is a pity they are all of the numerical and engineering type. It uses the Scheme dialect. Anderson et al. 1987 is a introductory textbook with lots of exercises. Friedman & Felleisen 1986 is a very funny (and good) programmed instruction course. If you want to learn to write recursive list programs in LISP without a teacher and without prerequisites, try this. Watch out for the differences between editions. There is an MIT Press version as well. Henderson 1980 is a good and clear introduction to functional programming, with many examples. Steele 1984 is the defining report on Common LISP, not intended for learning LISP but indispensable as a reference. If you have the opportunity, take a look into the Symbolics manuals to gain a feeling for the size and power of the programming environment of a LISP machine. Winston and Horn 1981 is a general introduction to LISP.
Glossary

Atom. In LISP: any symbol, number or other non-list.

Access function. A function which is part of a data abstraction layer (a selector or constructor function).

Anonymous function. A function whose "pure" definition is given, not assigning it a name at the same time.

Application. Obtaining a result by supplying a function with suitable arguments.

Combinator. A function that has only functions as arguments and returns a function as result.

Cons. A LISP primitive that builds lists. Sometimes used as verb: to add an element to the beginning of a list.

Constant function. A function that always returns the same value.

Constructor function. A function that, as part of the data-abstraction layer, provides a way of building a data structure from its components.

Continuation. A way of specifying what a function should do with its result.

Coroutines. Parts of the program that run in alternation, but remember their own state of computation in between switches.

Data abstraction. A way of restricting access and hiding detail of data structures.

Data type. A class of similar data objects, together with their access functions.

Dialect. A programming language can be split up into dialects that only differ (one hopes) in minor details. LISP dialects are abundant and may differ a lot from each other even in essential constructs.

First-class citizens. Rule by which any type of object is allowed in any type of programming construct.

Function. A program or procedure that has no side effects.

Function composition. The process of applying one function after another.

Functional abstraction (or procedural abstraction). A way of making a piece of code more general by turning part of it into a
parameter, creating a function that can be called with a variety of values for this parameter.

**FUNCTIONAL ARGUMENT (FUNARG).** A function that is passed as argument to another one (downward funarg) or returned as result from other one (upward funarg).

**FUNCTION QUOTE.** A construct to capture the correct intended meaning (with respect to the current lexical environment) of an anonymous function so it can be applied later in another environment; a lexical closure. It is considered good programming style to use function quotes as well when quoting just the name of a function.

**GLOBAL VARIABLES.** Objects that can be referred to (inspected, changed) from any part of the program.

**HIGHER-ORDER FUNCTION.** A function that has functions as arguments.

**IMPERATIVE STYLE.** A programming style in which assignment and iteration are the main constructs.

**ITERATION.** Repeating a certain segment of the program.

**LAMBDA-LIST KEYWORD.** A keyword that may occur in the list of parameter names in a function definition. It signals how this function expects its parameters to be passed, whether they may be omitted in the call, and so forth.

**LEXICAL SCOPING.** A rule that limits the “visibility” of a variable to a textual chunk of the program. Much confusion can result from the older—so-called dynamic scoping—rules.

**OBJECT-ORIENTED PROGRAMMING.** A style of programming whereby each data type is grouped with its own access function definitions, possibly inheriting them from other types.

**PARAMETER-FREE PROGRAMMING.** A style of programming whereby only combinators are used to build complex functions from simple ones.

**PREFIX NOTATION.** A way of notating function application by prefixing the arguments with the function.

**QUOTE.** A construct to prevent the LISP interpreter from evaluating an expression.

**RECURSION.** A method by which a function is allowed to use its own definition.

**SELECTOR FUNCTION.** A function that as part of the data abstraction layer provides access to a data structure by returning part of it.
SIDE EFFECT. Any actions of a program that may change the environment and so change the behavior of other programs.

STACK. A list of function calls that are initiated but have not yet returned a value.

STRUCTURE PRESERVING. A way of modifying data that keeps the internal construction intact but may change attributes attached to the structure.

TAIL RECURSION. A type of recursion in which the recursive call is the "last" thing the program does.

REFERENCES


