Abstract: This article presents pattern matching code which will be used for
discovering a Mozart signature based on one of the major program components of the
EMI (Experiments in Musical Intelligence) system. The code is self-sufficient. The
only requirement is that the reader use CommonLISP. For those who know LISP,
these functions are meant to demonstrate basic pattern-matching. For those who do
not know LISP but wish to, the code may be useful if used concomitantly with a
LISP text.

This article assumes that 1) pattern matching is a powerful technique to use in attempting to
discover some of the reasons why composer’s styles sound as they do; 2) patterns judged as alike
which occur in different works of a composer are valuable and constitute what the author will call
signatures; 3) re-using such signatures in music composed following standard chord protocols and
voice leading can lead to replications in the style of a composer. This latter point is not proven
here or with the works played at this conference because this code represents only a fraction of that
required for EMI’s compositional processes.

Data here will take the form of numerical representations for notes with 60 being middle C, 61
C#, etc. The figure below shows three short excerpts from three different Mozart Sonatas with the
pitch representations for each excerpt below it. Results are entered as 0’s.

```
0: This is bars 20-22 of the first movement of Sonata K 333.
(let (get ‘mozart-k.333 ‘notes) '((72 77 76 75 73 74 70 67 66 64 65))

0: This is bars 25-26 of the first movement of Sonata K 280.
(let (get ‘mozart-k.280 ‘notes) ((82 79 78 79 74 73 74 70 67 65 64))

0: This is bars 4-6 from the end of the last movement of Sonata K 333.
(let (get ‘mozart-k.333 ‘notes) ((79 77 75 74 72 70 69 75 69 70 74 77 75 72 69 70))
```

Obviously, using just pitch in the process here has significant limitations. Missing are all of the
inflections of timbre, duration, articulation, place within the bar, relation to beat, etc. However,
these can be factored into the pattern-matching process by additional code at strategic places. As
well, the examples are also small and not randomly chosen so the brief experimentation is
seriously biased. However, randomly choosing examples which will not prove the point is hardly
useful within the space limitations placed on this article.

ICMC GLASGOW 1990 PROCEEDINGS

288
By ear, one can quickly tell that the selections from the first two examples have a signature present (beginning on the sixth note of the second example and the fifth note of the first example). The pattern matches until the last note of the second example and the second to last note of the first example. The pattern-matcher described here will have to be able to find this figure. The match between the second and third examples is not so clear in that the last six notes of the third example, while resembling those of the first two examples, is not an exact match. The pattern-matcher, since the author does hear this as a variant, will also have to be able to catch this pattern.

Code is described here as it was written: top-down. All but the top-level function use recursion in traditional LISP convention. There are four program variables which are shown below.

(defvar *signatures* nil)
(defvar *range-tolerance* 1)
(defvar *error-tolerance* 0)
(defvar *pattern-size* 5)

The first is a location into which the results of the pattern-matching process will be stored. It is the place where we will find the patterns that have matched between two works given as arguments to the top-level function. The other three are what the author likes to term "tuners." They are numbers which allow alteration of the process without having to re-write code. The first, *range-tolerance*, indicates the allowance for a possible matching interval to be out of range. The default, as shown, is 1 or a minor second. This tuner will allow, for example, major and minor versions of the same interval to match. The second tuner, *error-tolerance*, indicates the number of intervals which can be completely wrong and still allow for a match. The default of zero, is a good one since it allows a lot of play with *range-tolerance* before having to turn to *error-tolerance*.

This will also be helpful in factoring rests which are problematic for matchers. Obviously, using high numbers in either or both of these tuners will cause noise so generally tuning will be in the 0 to 5 or so range. The last variable declaration, *pattern-size*, determines the size of motive that the program uses in trying to find matches. It should be noted that the default size here does not indicate any preference of the author. Setting this particular tuner is extremely important in the matching process. Amounts can range from 2 to whatever depending on the size of the note-lists being crunched. Large sizes with small note-lists will yield little if any results.

The top-level of the program is iterative. It operates the more powerful matching functions by feeding them data which equates to the works turned into intervals ("define-as-intervals") and lists of crunched patterns (according to the variable *pattern-size*). The function "signify-matches" counts the number of times a motive occurs in the resultant list and removes all of the occurrences other than the first. As will be seen, this matcher uses and returns intervals instead of note representations, since intervals are more pitch-sensitive (i.e. a minor second down followed by a major second up is the same whether its actual value be C-B-D or D-C#-E). This means that sequence and transpositions will be caught in the process of matching.

(defun pattern-match (work-1 work-2)
  (setq *signatures*
        (signify-matches (match (decide-patterns (define-as-intervals work-1)))
                         (decide-patterns (define-as-intervals work-2)))))

"Define-as-intervals" turns the data given as its arguments into a list of intervals. By subtracting the second note from the first the note numbers are translated into intervals. It causes one before the end of the list (see (null (second note-lists))) because it cannot figure an interval between the last note of the list and an empty list.

? (define-as-intervals '(-66 67 70 69 66))) = (1 2 1 -1 -2)
(delim define-as-intervals (note-lists))

ICMC GLASGOW 1990 PROCEEDINGS

219
"Decide-patterns" is a simple number cruncher which takes *pattern-size* and turns the note list into successive list of "pattern-size" sub-lists. Note that while taking *pattern-size* amounts of data from the first part of the list, it consumes one step at a time. Hence sub-lists represent all possible *pattern-size* (here set to 3) lengthed patterns in the argument. While this has the potential of creating redundancy in the process, it also causes whole patterns to emerge as will be seen in "agnify-matches."

? (decide-patterns '(14 23 22 2 3 4 5 6 7)) = (((14 23 22) (23 22 1) (2 1 2) (1 2 3) (2 3 4) (3 4 5) (4 5 6) (5 6 7))

(defun decide-patterns (work)
  (if (< (length work) *pattern-size*) nil
      (cons (butlast work (- (length work) *pattern-size*))
            (decide-patterns (rest work))))))

"Match" is the workhorse function of the program.

? (match '(1 2 3 5 6) ((2 3 4) (8 9 0))) = ((1 2 3))

(defun match (pattern-set-1 pattern-set-2)
  (cond (null pattern-set-1) nil
        (or (member (first pattern-set-1) pattern-set-2)
            (member t (negate nil (lambda (x) (match-details (first pattern-set-1) x) pattern-set-2))))
        (cons (first pattern-set-1)
              (match (rest pattern-set-1) pattern-set-2)))
    (t (match (rest pattern-set-1) pattern-set-2))))

The arguments here are two sets of patterns the size of *pattern-size*. Match moves down the first list determining whether the interval numbers of each sub-list match or not. Its first attempt is explicit and shown in the expression "(member (first pattern-set-1) pattern-set-2)." Such an attempt avoids the necessity of further evaluation if the pattern from the first list is in the second list. Further computation is necessary through the use of "match-details" in the ensuing line if such a find does not occur.

"Match-details" applies the various tuners to the matching process so that "almost" matches may occur. It takes two arguments such as which equates to a list of intervals from two different works. The conditions of acceptance are shown in the "or" argument in line four of the code. Three possibilities exist: 1) the intervals will be the same (note this is not the same as the use of "member above" - in that case all of the members of each list must be the same); 2) they will be off within the range of *range-tolerance*; 3) they will be acceptable to "tolerate-errors" which uses the program variable *error-tolerance*. The tuners for the runs shown have the default settings.

? (match-details '(2 3 4) (3 4 5)) = T
? (match-details '(2 3 4) (7 8 9)) = NIL

(defun match-details (pattern-1 pattern-2)
  (cond (null pattern-2) nil
        (for i (seq (first pattern-2) (first pattern-2))
            (= (abs (- (first pattern-1 (first pattern-2)) *range-tolerance*)
                (tolerate-errors pattern-1 pattern-2 *error-tolerance*))
               (match-details (rest pattern-1) (rest pattern-2))))
    (t nil)))

ICMC GLASGOW 1960 PROCEEDINGS 29
"Tolerate-errors" recurses through its first two arguments in parallel, reducing its third argument incrementally if no match is found. When *error-tolerance* is zero, no match is found. *error-tolerance* has the value of 1 for the runs shown.

```lisp
(symbol (tolerate-errors (1 2 3) (1 1 1) "error-tolerance") = 1
(symbol (tolerate-errors (1 2 3) (1 1 1) "error-tolerance") = 0
(defun tolerate-errors (pattern-1 pattern-2 error-tolerance)
  (cond (null pattern-1) T
      ((error-tolerance nil)
       (cons (first pattern-1) (first pattern-2)
             (error-tolerance (rest pattern-1) (rest pattern-2) error-tolerance)))
      (T (tolerate-errors (first pattern-1) (first pattern-2) error-tolerance)))
```

"Signify-matches" counts the number of times a motive occurs in the *signature* list and returns it as a list of the number plus the list. It then destructively removes the list from the match-list so that it will not repeat during recursion.

```lisp
(symbol (signify-matches '((1 1 1) (1 1 1))) ((2 (1 1 1)))
(defun signify-matches (match-list)
  (if (null match-list) nil
      ((cons (list count (first match-list) match-list test list)
            (first match-list))
       (signify-matches (remove (first match-list) match-list test list))))
```

This function is very helpful during composition in that signatures of low denomination can be struck from the process. The Fineness runs in the previous figures helps one understand the process but examples of the program's operation are required to understand how the tunes' argument variations work. The following figures show a few sample runs and the output in intervals.

```lisp
(#*range-tolerance* 1
  #*error-tolerance* 0
  #*pattern-size* 5
  (pattern-match (get 'mozart-K.533 notes) (get 'mozart-K.280 notes)) = (2 (-2 -4 -3 -2))
  (pattern-match (get 'mozart-K.533 notes) (get 'mozart-K.333 notes)) = NIL
  (#*size *range-tolerance* 2) = 2
  (pattern-match (get 'mozart-K.533 notes) (get 'mozart-K.333 notes)) = ((5 -1 -1 -1))
  (pattern-match (get 'mozart-K.280 notes) (get 'mozart-K.333 notes)) = ((2 -1 -4 -3 -2))
```

Tuning too conservatively can produce little results. Tuning too radically, however, can produce signatures when there are none. One of the major problems of this particular matcher is that it does not provide clear information as to how many matches have occurred within each of the works it analyzes. Therefore, one does not know if this is a true signature. For example, given the output of 8 as the number of times a pattern has occurred is not too useful if it equates to 7 and 1 matches in the works respectively. 4 and 4 as counts would produce more convincing results. Adding a feature to discern this and creating more tuners would certainly enhance this program's ability to recognize pitch aspects of musical style. Altering this code to produce rhythm, dynamic, harmonic, texture and other matchings could offer further refinement and develop more sophisticated tools. Utilizing the results of this code compositionally is a matter of poking signatures into computer-composed and styleless tonal music. This is described in the author's other publications listed in the bibliography as well as the work of many others. Composition of new works with signatures may be found in the author's book *Computers and Musical Style* to be released from A-R Editions of Madison, W1.