Studies in Movement (Set 1)

For Violoncello solo

Nigel Morgan

Symbolic Composer code annotated by Phil Legard

Studies in Movement (Set 1)

For violoncello solo

Nigel Morgan

Annotated SCOM code by Phil Legard

The sonic quality of the cello is ideally suited to slow, expressive, languorous music. The instrument is also blessed with a wide compass: a rich sonorous bass register way up into the territory of the violin, and here with a powerful, unique and incisive timbre. But the music contained in these *Studies in Movement* favours lightness, speed and agility, and that dance-like fleet-of-footness found in the best Baroque performance practice, and notably in the dance movements of the six Cello Suites of J.S.Bach.

Studies in Movement (Set 1) is one of a continuing series of works that take their starting points – harmonically at least – from the patterns given in Nicolas Slonimsky's *Thesaurus of Scales and Melodic Patterns*. An overview of Nigel Morgan's music derived from Slonimsky series can be found **here**.

The Studies presented here are a preface to *Ideograms* in the composer's series *Facts of Life*. This is an expanding collection of music for instruments and Active Notation of which *Ideograms* for cello solo, designed for on-line distribution and performance, is part. Both works have been inspired by the remarkable cellist Peter Gregson, a musician who is able to bring a unique experience of Baroque techniques together with a fascination for the latest technological innovations (he has collaborated with MIT Media Lab and Banff Centre for the Arts).

The following pages present the Symbolic Composer code that was written by Nigel Morgan to realize the *Studies in Movement*. If you wish to compile the code with your own version of Symbolic Composer, first *evaluate* the additional functions given in **Appendix A**, after which the code for each movement should execute without any issues.

```
;; Nine Studies in Movement (No.1 Continuum)

;; functions

(defun gen-palindrome-r (pat &optional seed)
; randomizes the output length of the palindrome
(if seed
(init-rnd seed))
  (prog (out len)
    (setq len (- (length pat) 1))
    (setq out
          (symbol-trim (- (+ (length pat) len)(get-random 1 len))
                       (gen-palindrome pat)))
    (return out)))


(defun symbol-interpolate-x (val start end)
(do-section :all '(gen-palindrome-r x) (symbol-interpolate val start end)))

;; material

(setq s85u '(a -c -b g e f))
(setq s85d '(m f e g -b -c))
(setq s86u '(a -d -b g d f))
(setq s86d '(m f d g -b -d))
(setq s87u '(a -e -b g c f))
(setq s87d '(m f c g -b -e))
(setq s88u '(a -f -b g b f))
(setq s88d '(m f b g -b -f))
(setq s89u '(a -d -c g d e))
(setq s89d '(m e d g -c -d))
(setq s90u '(a -e -c g c e))
(setq s90d '(m e c g -c -e))
(setq s91u '(a -f -c g b e))
(setq s91d '(m e b g -c -f))
(setq s92u '(a -e -d g c d))
(setq s92d '(m d c g -d -e))
(setq s93u '(a -f -d g b d))
(setq s93d '(m d b g -d -f))
```

This function creates a phrase to which is appended a palindrome with a random length. For example (a b c d) may yield (a b c d c), (a b c d c b) or (a b c d c b a).

The basic **symbol-interpolate** function transforms one phrase into another over a given series of steps, so (a b c) to (a g e) over four steps might look like this: (a b c) (a d d) (a e d) (a g e) This new function, **symbol-interpolate-x,** creates an interpolation that is then annexed by a palindromic version of the phrase through the use of **gen-palindrome-r**.

Slonimsky patterns 85-93 defined as sets of both ascending and descending pitch symbols.

```
(setq mat-1 (symbol-interpolate-x 8 s85u s93d)
      mat-2 (symbol-interpolate-x 6 s86u s92d)
      mat-3 (symbol-interpolate-x 4 s87u s91d)
      mat-4 (symbol-interpolate-x 2 s88u s90d)
      mat-5 (symbol-interpolate-x 4 s89u s85d)
      mat-6 (symbol-interpolate-x 2 s88u s86d)
      mat-7 (symbol-interpolate-x 3 s87u s85d)
      mat-8 (list s86d s87u)
      mat-9  (symbol-interpolate-x 5 s85d s88u)
      mat-10 (symbol-interpolate-x 5 s86d s89u)
      mat-11 (symbol-interpolate-x 5 s87d s90u)
      mat-12 (symbol-interpolate-x 5 s88d s91u)
      mat-13 (symbol-interpolate-x 5 s89d s92u)
      mat-14 (symbol-interpolate-x 5 s90d s93u))
```

Define the basic material for the movement by creating interpolations of differing length between pairs of Slonimsky patterns. Note that **mat-8** acts as a sort of bridge between the two sets of material and simply consists of pattern **s86d** followed by **s87u**.

```
(setq mat-A (append mat-1 mat-2 mat-3 mat-4 mat-5 mat-6 mat-7 mat-8)
      mat-B (append mat-9 mat-10 mat-11 mat-12 mat-13 mat-14))
```

The material is **append**ed into two sections, **mat-A** and **mat-B**. These will ultimately be brought together to constitute the pitch material of the entire piece. However they are kept separate here so that different rules can be applied to them when generating durational data.

```
(setq down-v (gen-dim 100 60 10)
      up-v (gen-cresc 60 110 10)
      mid1-v (gen-cresc-dim 50 90 12)
      mid2-v (gen-dim-cresc 96 55 12)
      cen1-v '(40 70 50)
      cen-v '(120 90 106))
```

A series of velocity gestures are created. If these are longer than the corresponding number of pitch symbols then the extra velocity values will be omitted. If they are shorter then the velocities will repeat.

```
(setq vel-pat-1 (list up-v mid2-v cen1-v cen1-v cen-v cen-v mid1-v down-v
                      up-v mid2-v cen1-v cen-v mid1-v down-v
                      up-v mid2-v mid2-v down-v
                      up-v down-v
                      up-v cen-v cen-v down-v
                      up-v down-v
                      up-v cen1-v down-v
                      up-v down-v))
```

The velocities defined above are assembled into a dynamic scheme for the first half of the movement. Each line corresponds to one of the series **mat-1** to **mat-8**.

```
(setq vel-pat-2 (gen-repeat 6 (list down-v mid1-v cen-v mid2-v up-v)))
```

In the second half of the movement the dynamic scheme repeats throughout. There are five interpolations in each of the series **mat-9** to **mat-14**, which are mapped to the dynamic scheme **down-v midl-v cen-v mid2-v up-v**.

```
(setq dur-pat-1 '((1/8)(1/8) (1/16 1/16 1/32) (1/16 1/16 1/32)(1/8 1/16 1/32)(1/8 1/16 1/32)(1/8)(1/8)
                  (1/8)(1/8)(1/16)(1/16)(1/8)(1/8)
                  (1/8)(1/8)(1/8)(1/8)
                  (1/8)(1/8)
                  (1/8)(1/16)(1/16)(1/8)
                  (1/16)(1/16)
                  (1/16)(1/32) (1/16)
                  (1/32)(1/32)
                  ))
```

In a similar manner to the dynamic content, the durations are here defined by hand for the first part of the movement and as a repeating series for the second. Note that durations are separate from note lengths and suggest articulations such as staccato. In which case we find durations such as 1/16 may correspond to note lengths of 1/8.

```
(setq dur-pat-2 (gen-repeat 6 '((1/8)(1/8) (1/8 1/16 1/32) (1/8) (1/16))))


(setq len-1 (do-section  '(25 :end) '(change-length :divide 2 x :ratio)
                        (gen-process '(symbol-repeat x y) (mapcar 'length mat-A) '(1/8) :list))

      len-2 (do-section '(= = = = x) '(change-length :divide 2 x :ratio)
                        (gen-process '(symbol-repeat x y)
                         (mapcar 'length mat-B) '(1/8) :list)))
```

To define the length material for the first part of the movement **gen-process** is used with **symbol-repeat** to create groups of 1/8 values that correspond to the length of the material. The last 25 of these groups are then divided by two to give a series of 1/16 notes.

In the second section a simple template is used to process every fifth phrase (- - - - x), dividing the default length (1/8) by two.

```
;; score-template

(def-tonality
 cello (activate-tonality (chromatic f 3))
)


(def-symbol
  cello (append mat-A mat-B)
)

(def-length
  cello (append len-1 len-2)
)
```

```
(def-duration
 cello (append dur-pat-1 dur-pat-2)
)

(def-velocity
 cello (append vel-pat-1 vel-pat-2)
)

(def-zone
  cello (z-ratio-sc (append len-1 len-2))
; (3/4 9/8 7/8 9/8 1/1 3/4 7/8 3/4 1/1 7/8 9/8 5/4 3/4 1/1 3/4 5/4 5/4 9/8 9/8 7/8 1/1 7/8 7/8 7/8
; 7/16 9/16 9/16 7/16 1/2 3/8 3/8 9/8 9/8 5/4 3/4 9/16 5/4 3/4 9/8 3/4 9/16 9/8 1/1 1/1 9/8 7/16
; 7/8 9/8 5/4 7/8 5/8 5/4 1/1 3/4 1/1 1/2 5/4 9/8 7/8 3/4 3/8)
)

(def-channel
 cello  1
)

(def-tempo 100)

(compile-instrument-p "ccl;output:" "continuum-Fi"
 cello
)

#| mat-A
((a -c -b g e f) (c -b -b g d e d g -b) (d -b a g c c c) (f a a g b b b g a)
(h b a g a -b a g) (j c a g -b -c) (k c b g -c -e -c) (m d b g -d -f) (a -d -b g d f d g)
(c -c a g c d c) (f -b a g b b b g a) (h b b g -b a -b g b b) (k c b g -c -c) (m d c g -d -e -d g)
(a -e -b g c f) (e -b a g b c b g a -b) (i b a g -b -c -b g a b) (m e b g -c -f -c g b)
(a -f -b g b f b g -b) (m e c g -c -e -c) (a -d -c g d e d g) (e a a g c c c) (i c c g a a a)
(m f e g -b -c -b) (a -f -b g b f b) (m f d g -b -d -b g d) (a -e -b g c f c g -b) (g a c g a c a)
 (m f e g -b -c -b g) (m f d g -b -d) (a -e -b g c f))

mat-B
((m f e g -b -c -b g e) (j c d g a a a g d) (g a c g a c a g c a) (d -c a g a d)
(a -f -b g b f b g -b) (m f d g -b -d -b g d f) (j d c g a -b) (g b a g b a b g a)
(d -b -b g c c) (a -d -c g d e d g -c) (m f c g -b -e -b g c) (j d b g a -c a g)
(g a a g a a a g) (d -c -b g b c b g -b) (a -e -c g c e c) (m f b g -b -f -b) (j c a g a -d a g a)
(g a a g a a a g a a) (d -c -b g a c a) (a -f -c g b e b g -c -f) (m e d g -c -d -c g d e)
 (j c c g -b -c -b g) (g a a g a a) (d -c -c g b c b g) (a -e -d g c d c g) (m e c g -c -e -c g c e)
(j c b g -b -c -b g b) (g a a g a a a) (d -d -c g a b) (a -f -d g b d))
 |#
```

```
;; Nine Studies in Movement (No.2 Arrested Movement)

(setq s85u '(a -c -b g e f))
(setq s85d '(m f e g -b -c))
(setq s86u '(a -d -b g d f))
(setq s86d '(m f d g -b -d))
(setq s87u '(a -e -b g c f))
(setq s87d '(m f c g -b -e))
(setq s88u '(a -f -b g b f))
(setq s88d '(m f b g -b -f))
(setq s89u '(a -d -c g d e))
(setq s89d '(m e d g -c -d))
(setq s90u '(a -e -c g c e))
(setq s90d '(m e c g -c -e))
(setq s91u '(a -f -c g b e))
(setq s91d '(m e b g -c -f))
(setq s92u '(a -e -d g c d))
(setq s92d '(m d c g -d -e))
(setq s93u '(a -f -d g b d))
(setq s93d '(m d b g -d -f))

(setq i-1 '(a -c)
      i-2 '(-c -b)
      i-3 '(-b g)
      i-4 '(g e)
      i-5 '(e f)
      i-6 '(f a))

(setq m-1 (gen-process '(symbol-transpose x y) (mapcar 'symbol-to-integer s85u) i-1)
; (a -c -c -e -b -d g e e c f d)
      m-2 (gen-process '(symbol-transpose x y) (mapcar 'symbol-to-integer s85u) i-2)
; (-c -b -e -d -d -c e f c d d e)
      m-3 (gen-process '(symbol-transpose x y) (mapcar 'symbol-to-integer s85u) i-3)
; (-b g -d e -c f f m d k e l)
      m-4 (gen-process '(symbol-transpose x y) (mapcar 'symbol-to-integer s85u) i-4)
; (g e e c f d m k k i l j)
      m-5 (gen-process '(symbol-transpose x y) (mapcar 'symbol-to-integer s85u) i-5)
; (e f c d d e k l i j j k)
      m-6 (gen-process '(symbol-transpose x y) (mapcar 'symbol-to-integer s85u) i-6))
; (f a d -c e -b l g j e k f)
```

Here is the selection of Slonimsky patterns used throughout the *Studies*. The second movement is based around pattern 85, defined here by the name **s85u**.

Here six pairs of pitches are defined based on consecutive pairs of symbols from the Slonimsky pattern **s85u** (a –c –b g e f). This movement uses a chromatic tonality where pitch symbol **a** corresponds to **C4**. **I-1**, the first pair of pitches shown here, symbolically describe a note moving down major second (e.g. (a –c) or (C4 Bb3)). **I-2** describes a single step upward ((-c -b) or (Bb3 B3)) and so on.

The Slonimsky series is converted into a set of numerical values using **symbol-to-integer**. This gives us the following series (0 –2 –1 6 4 5).
Each of the above symbol pairs are transposed against each of these values, so (a –c) yields:

0	-2	1	6	4	5
a –c	-c –e	-b –d	g e	e f	f d

We then have a series of phrases that extend the original phrase in a highly self-similar (fractal) manner

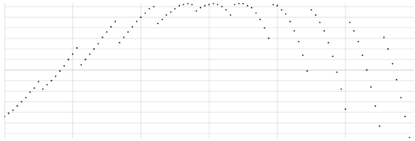
```
(setq m-all (append m-1 m-2 m-3 m-4 m-5 m-6))

(symbol-divide 2 'setq 'x m-all)

(setq x0 '(a -c))
(setq x1 '(-c -e))
(setq x2 '(-b -d))
(setq x3 '(g e))
(setq x4 '(e c))
(setq x5 '(f d))
(setq x6 '(-c -b))
(setq x7 '(-e -d))
(setq x8 '(-d -c))
(setq x9 '(e f))
(setq x10 '(c d))
(setq x11 '(d e))
(setq x12 '(-b g))
(setq x13 '(-d e))
(setq x14 '(-c f))
(setq x15 '(f m))
(setq x16 '(d k))
(setq x17 '(e l))
(setq x18 '(g e))
(setq x19 '(e c))
(setq x20 '(f d))
(setq x21 '(m k))
(setq x22 '(k i))
(setq x23 '(l j))
(setq x24 '(e f))
(setq x25 '(c d))
(setq x26 '(d e))
(setq x27 '(k l))
(setq x28 '(i j))
(setq x29 '(j k))
(setq x30 '(f a))
(setq x31 '(d -c))
(setq x32 '(e -b))
(setq x33 '(l g))
(setq x34 '(j e))
(setq x35 '(k f))
```

Join all the above self-similar phrases into one long series.

Divide the complete phrase series into pairs. Give each pair a name prefixed with **x**, as seen here in **x0-x35**.

```
(setq wave-gesture
      (cfunction
       (gen-sin 0.5 0.1 96 180 (gen-ramp 10 0.3 90))
       (list-a-scale 0 36)))
```

A sine wave is modulated with a ramp wave to create a wavelike gesture. Scale each value to between 0 and 36.



See the code for *Array* to explore a similar sine-ramp structure.

```
; (6 6 7 8 9 10 12 13 14 13 14 15 16 17 19 20 22 23 19 20 22
; 23 25 26 28 29 30 25 26 28 29 30 31 33 33 34 30 31 32 33 34
; 34 35 35 35 33 34 34 35 35 35 34 33 32 35 35 35 35 34 33 31
; 29 26 35 34 33 32 30 28 25 21 17 33 32 30 28 25 21 17 13 7
; 30 28 25 22 18 13 8 3 26 23 19 15 10 5 0)

(setq wave-gesture-r (symbol-trim 42 (reverse wave-gesture)))
```

The numbers output by wave gesture are shown above. You can see that the output beings begins in a predictable manner – simply going from one value to the next until the effects of the ramp wave assert themselves and the range of numbers become more varied. To create a sort of coda once the series has reached its most varied point a second series is formed by reversing and trimming the original.

```
(setq wave-A (eval-section-integer wave-gesture 'x 'list)
      wave-B (eval-section-integer wave-gesture-r 'x 'list))
```

For both wave outputs (the full series and the reverse & trimmed series) replace each value with the corresponding pitch pair generated above , so that (6 6 7 8 9 . . .) = (x6 x6 x7 x8 x9 . . .) = ((-c –b) (-c –b) (-e –d) (-d –c) (e f) . . .)

```
(setq ins-pauses (e-insert
                  (gen-collect nil 12 :list '(build-list '= (get-random 3 7)))
                  (sort< (pick-random-n 12 (list-a-scale 6 90))) wave-A)
      ins-pauses-r (e-insert
                  (gen-collect nil 12 :list '(build-list '= (get-random 2 5)))
                  (sort< (pick-random-n 6 (list-a-scale 6 40))) wave-B))
```

Create a series of 12 empty phrases and interpolate them into the series **wave-A** and **wave-B**.
For **wave-A**, **build list** is used in conjunction with **gen-collect** to create 12 instances of phrases with between 3 and 7 rest symbols, e.g. '(= = = =). 12 random values between 6 and 90 are chosen and sorted into order. **E-insert** then inserts a pause phrase at each of these points.

```
(setq len-1 (gen-process '(symbol-repeat x y) (mapcar 'length ins-pauses) '(1/8) :list)
      len-1r (gen-process '(symbol-repeat x y) (mapcar 'length ins-pauses-r) '(1/8) :list))
```

Use **gen-process** to execute symbol-repeat once for each phrase in our series, creating corresponding groups of 1/8 length values. So, ((-c –b)(= = =)(-e –d)) becomes ((= =)(= = =)(= =)).

```
(setq mov-orn '(x x x x x x = =  = = = =  =  = = = = = = = = x x x x x x x x = x x x x = = =
                =  = = = = = = = = = = = = = =  =
                = = = = = =  =  = = = = = = = x x = =  =  =  =  = = x x = = = = =  =  x x = = =
                =  x =  = =  x x x x = x x x x x = x x =))

(setq mov-orn-r '(= = x = x x x  =  x x =  =  x  =  =  x  =  x x = x = = = x x = = = = = x
                  x = = = = = = x x = = = =  =  = = =))

(setq len-2 (do-section mov-orn '(length-repeat 2 x) len-1)
      len-2r  (do-section mov-orn-r '(length-repeat 2 x) len-1r))
```

A template is created corresponding to the length of each series of phrases. This is used in conjunction with **do-section** to execute the **length-repeat** function on selected phrases. This divides existing lengths by a given factor in this case 2, so (1/8 1/8) becomes (1/16 1/16 1/16 1/16).

```
(def-neuron edit-1
 (in 1 '0) '0
 (otherwise '1))
```

Create a neuron that will look at a series of values and return 0 for 0 and 1 for any other number, so (0 0 2 0 5 0 1) becomes (0 0 1 0 1 0 1).

```
(setq ins-rest-p
      (e-position '1
                  (run-neuron 'edit-1  (flatten (do-section :all '(e-count '= x) ins-pauses))))
      ins-rest-r
      (e-position '1
                  (run-neuron 'edit-1  (flatten (do-section :all '(e-count '= x) ins-pauses-r)))))
```

The number of rests for each phrase are counted using **e-count** within a **do-section** function. Running the results through the neuron defined above will give us an overview of the relation of ‘sounding’ phrases to resting phrases (e.g. (0 0 1 0 1 0 1). The position of each phrase of rest is then calculated using e-position to look for instances of the number 1 in the series, e.g. (2 4 6).

```
(setq dyn-x (vector-to-list (vector-round 40 110 (gen-sin 0.5 0.1 96 108 (gen-ramp 10 0.3 108))))
      dyn  (e-insert '((0))  ins-rest-p (symbol-divide 2 nil nil (symbol-interleave dyn-x
                        (do-section :all '(car (change-length :sub 10  x))
                                    (symbol-divide 2 nil nil (symbol-repeat 2 dyn-x)))))))
```

To create a dynamic scheme another sine wave modulated to a ramp wave is created. Where rests occur a (0) value is inserted into the series using the position data obtained above. Since all other phrases are pairs the velocity for each is duplicated and then also has 10 subtracted.

```
(setq dyn-x-r (symbol-trim 42 (reverse dyn-x))
      dyn-r (e-insert '((0))  ins-rest-r (symbol-divide 2 nil nil (symbol-interleave dyn-x-r
                          (do-section :all '(car (change-length :sub 10  x))
                                      (symbol-divide 2 nil nil (symbol-repeat 2 dyn-x-r)))))))
;; score

(def-tonality
 cello (activate-tonality (chromatic c 4))
)

(def-symbol
cello (append ins-pauses (do-section :all '(reverse-pairs x) ins-pauses-r))
)

(def-length
cello (append len-2 len-2r)
)

(def-velocity
cello (append dyn dyn-r)
)

(def-zone
cello  (z-ratio-sc (append len-1 len-1r)))
#|
(1/4 1/4 1/4  1/4 1/4 1/4 3/8 3/4 1/4 1/4 1/4 1/4 3/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4
1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/2 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4
1/4 3/4 1/4 1/4 1/4 1/4 1/4 1/4 1/2 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/2 1/4 7/8 1/4 1/4 1/4
1/4 1/4 1/4 1/4 1/4 1/4 1/2 1/4 1/4 1/4 1/4 1/4 3/4 1/4 1/4 1/2 5/8 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4
 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 3/8 1/4 1/4 1/4 3/8 1/4 1/2 1/2 1/4 1/2 1/4 1/4 1/4 1/4
1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/2 1/4 1/4 1/4)
|#

(def-channel
cello  1
)

(def-tempo 120)

(compile-instrument-p "ccl;output:" "arrested-movement-4d"
 cello
)
```

Define the pitch, length and velocity content for the entire piece by appending the two series together. The reverse-pairs function is also run on the reversed & trimmed series to create a true mirror image of the first series.

```
;; Nine Studies in Movement (No.3 Dance Figures)

;; material

(setq s85 '(a -c -b g e f)
      s86 '(a -d -b g d f)
      s87 '(a -e -b g c f)
      s88 '(a -f -b g b f)
      s89 '(a -d -c g d e)
      s90 '(a -e -c g c e)
      s91 '(a -f -c g b e)
      s92 '(a -e -d g c d)
      s93 '(a -f -d g b d))


(create-tonality tonal-s85 '(c 4 e 4 f 4 f# 4 a# 4 b 4))
(create-tonality tonal-s87 '(c 4 d 4 f 4 f# 4 g# 4 b 4))
(create-tonality tonal-s89 '(c 4 d# 4 e 4 f# 4 a 4 a# 4))
(create-tonality tonal-s91 '(c 4 c# 4 e 4 f# 4 g 4 a# 4))
(create-tonality tonal-s93 '(c 4 c# 4 d# 4 f# 4 g 4 a 4))
(create-tonality tonal-s92 '(c 4 d 4 d# 4 f# 4 g# 4 a 4))
(create-tonality tonal-s90 '(c 4 d 4 e 4 f# 4 g# 4 a# 4))
(create-tonality tonal-s88 '(c 4 d& 4 f 4 f# 4 g 4 b 4))
(create-tonality tonal-s86 '(c 4 d& 4 f 4 f# 4 g 4 b 4))
```

Create a series of tonalities based on the above Slonimsky series reduced into a single octave compass.

```
(create-tonality pizz-t '(c 3 g 3 d 4 a 4))
```

A tonality that corresponds to the open strings of the cello, to produce left-hand *pizzicatos*.

```
(setq tonal-1 (gen-repeat 10 (activate-tonality (tonal-s85 c 4)))
      tonal-2  (activate-tonality '((tonal-s85 c 4)(tonal-s85 c 4)(tonal-s87 c 4)
                                    (tonal-s87 c 4)(tonal-s89 c 4)(tonal-s89 c 4)
                                    (tonal-s91 c 4)(tonal-s91 c 4)
                                    (tonal-s93 c 4)(tonal-s93 c 4)))
      tonal-3 (gen-repeat 10 (activate-tonality (tonal-s93 c 4)))
      tonal-4 (activate-tonality '((tonal-s92 c 4)(tonal-s85 c 4)(tonal-s90 c 4)
                                   (tonal-s88 c 4)(tonal-s86 c 4))))
```

Define a tonal plan for the movement. **Tonal-1** and **Tonal-4** consist of repetitions of a single tonality, whereas the others suggest tonal movement from zone to zone (e.g. phrase to phrase).

```
(setq symbol-pattern (but-last (gen-palindrome '(a b c d e f)))
      ; (a b c d e f e d c b)
     sym-pat (but-last (gen-palindrome (reverse '(a b c d e f)))))
      ; (f e d c b a b c d e)
```

Create a pair of palindromic sequences. Note the use of **but-last** to remove the final value, which would otherwise duplicate the first note when the series was repeated.

```
(setq temp-1
      (gen-collect 0.13 10 :list
                   '(gen-template nil 1 1 (length symbol-pattern))))
; ((x = = = = x x x = =) (= x x x x = x x x =) (= x x x = x x = x x) (x x = = = = = x = =)
; (x x = x = = x = x x) (x x = = = = x x x =) (x x x = = x x = = x) (= = = = = x x x x x)
; (= x = x x = = = = x) (x = = = = = x x = =))

(setq temp-2
      (gen-collect 0.15 10 :list
                   '(gen-template nil 2 1 (length symbol-pattern))))

(setq temp-3
      (gen-collect 0.17 10 :list
                   '(gen-template nil 1 1 (length symbol-pattern))))

(setq temp-4
      (gen-collect 0.14 5 :list
                   '(gen-template nil 3 1 (length sym-pat))))
```

Create a series of templates in which each section is the same length as the palindromes generated above. The first three are based on 10 repetitions of the first palindrome (**symbol-pattern**), the final based on five repetitions of the second (**sym-pat**).

```
(setq s-pat-list (gen-repeat 10 (list symbol-pattern))
     s-patx-list (gen-repeat 5 (list sym-pat)))
```

Create a series of repeats of each palindrome. These can then be used in conjunction with the templates above.

```
(setq matx-1 (gen-process-list '(fill-template-swallow x y)  temp-1 s-pat-list))
; ((a = = = = f e d = =) (= b c d e = e d c =) (= b c d = f e = c b) (a b = = = = = d = =)
; (a b = d = = e = c b) (a b = = = = e d c =) (a b c = = f e = = b) (= = = = = f e d c b)
; (= b = d e = = = = b) (a = = = = = e d = =))


(setq matx-2 (gen-process-list '(fill-template-swallow x y)  temp-2 s-pat-list)
     matx-3 (gen-process-list '(fill-template-swallow x y)  temp-3 s-pat-list)
     matx-4 (gen-process-list '(fill-template-swallow x y)  temp-4 s-patx-list))
```

Filter the first palindrome using **fill-template-swallow**. Here each (x) value in the template will be filled with the corresponding value in the phrase.

```
(setq len-1 (gen-process '(symbol-repeat x y) (mapcar 'length matx-1) '(1/8) :list)
      len-1C (gen-process '(symbol-repeat x y) (mapcar 'length matx-4) '(1/8) :list))

;;;------------------

(setq len-1x
      (gen-process-list '(length-condense (align-to-symbol x y)) matx-1 len-1))
; ((-1/2 3/8 -1/4 1/8) (-1/8 3/4 -3/8) (-3/8 1/8 -3/8 3/8) . . . ..


(setq matx-1x (do-section :all '(delete '= (find-beat x)) matx-1))
; ((e b) (b) (d d) (b e e) (c c) . . .


(setq len-2 (p-replace-sections '(= x x = x x = = x x) len-1x len-1)
      mat-2 (p-replace-sections '(= x x = x x = = x x) matx-1x matx-1))
;((1/8 1/8 1/8 1/8 1/8 1/8 1/8 1/8 1/8 1/8) (-1/8 3/4 -3/8) (-3/8 1/8 -3/8 3/8)
;((= = = = e f e = = b) (b) (d d) (= b c = e = e d c =) (c c) . . .

;; -----------------
(setq len-2x
      (gen-process-list '(length-condense (align-to-symbol x y)) matx-2 len-1))

(setq matx-2x (do-section :all '(delete '= (find-beat x)) matx-2))

(setq len-3 (p-replace-sections '(x x x = = = x x = = =) len-2x len-1)
      mat-3 (p-replace-sections '(x x x = = = x x = =) matx-2x matx-2))

;; -----------------

(setq len-3x
      (gen-process-list '(length-condense (align-to-symbol x y)) matx-3 len-1))

(setq matx-3x (do-section :all '(delete '= (find-beat x)) matx-3))

(setq len-4 (p-replace-sections '(= = x x = x = = x =) len-3x len-1)
      mat-4 (p-replace-sections '(= = x x = x = = x =) matx-3x matx-3))

;; --------------------
```

For each value in the generated materials make a series of (1/8) lengths. **Matx-2** and **matx-3** are omitted since they are the same size as **matx-1** (10 phrases length).

Condense the length content of each phrase so that all consecutive rest symbols are grouped together as a single negative value, while all pitch symbols are grouped together under positive values. So (= = a = a b) becomes (-1/4 1/8 -1/8 1/4).

For each phrase delete all the rest symbols and keep only the first pitch symbol that falls immediately after a rest or at the start of a phrase. So (a = = b c d = a =) becomes (a b a).

Replace selected sections of **len-1** with those from **len-1x**. Note that when a phrase in **len-1x** such as (b) corresponds with (-1/8 3/4 –3/8) in matx-1x the compiler will understand this to mean that (b) has a duration of (3/4). The negative values are ignored in the absence or corresponding rest symbols.

```
(setq pizz-m (gen-process-list '(fill-template x y)
                               (mapcar 'find-beat (mapcar 'template-invert temp-1))
                               (gen-repeat 10 (list '(a b c d))))
      pizz-n (gen-process-list '(fill-template x y)
                               (mapcar 'find-beat (mapcar 'template-invert temp-2))
                               (gen-repeat 10 (list '(a b c d))))
      pizz-o (gen-process-list '(fill-template x y)
                               (mapcar 'find-beat (mapcar 'template-invert temp-3))
                               (gen-repeat 10 (list '(a b c d))))
      pizz-p (gen-process-list '(fill-template x y)
                               (mapcar 'find-beat (mapcar 'template-invert temp-4))
                               (gen-repeat 5 (list (reverse '(a b ))))))
;; ----------------------

(setq pp '(35 30)
      p '(45 40)
      mp '(60 55)
      mf '(75 70)
      f  '(96 90)
      ff '(115 110))

(setq dyn-1 (list mf p ff mf p ff f f mp f
                  mp mp p pp p mp mf f mp p
                  mf f mp f mf f mp mf ff mp
                  f mf mp p mp))

;; score-template

(def-tonality
cello  (append tonal-1 tonal-2 tonal-3 tonal-4)
pizz   (gen-repeat 35 (activate-tonality (pizz-t c 3)))
)


(def-symbol
 cello  (append (do-section '(= = = = = = x x = =) '(ornament-higher 5 (symbol-repeat 2 x)) mat-2)
                (do-section '(= = = = = x = = x x) '(ornament-higher 4 (symbol-repeat 2 x)) mat-3)
                (do-section '(x x = = x = x x = =) '(ornament-lower 3 (symbol-repeat 2 x)) mat-4)
                (do-section '(= = = = x) '(ornament-lower 9 (symbol-repeat 2 x)) matx-4))
 pizz   (append pizz-m pizz-n pizz-o pizz-p)
)
```

Invert the previously defined templates and then execute **find-beat** so that only the first symbol after a rest is included, so (x x x x = = = x x =) becomes (x = = = = = = x = =). These values are then filled with an open string (a b c d) as an *ostinato*-style *pizzicato*.

Define a dynamic scheme for each phrase in each section of the whole piece.

Begin to bring all the sections together, starting with tonality. Note that the *pizz* tonality must remain the same throughout!

Bring together all the symbolic material with **append**. Also a template is used with ornament functions to avoid repeated notes and create a recurrent series of dance-like ornamentations.

```
(def-length
 cello (append (do-section '(= = = = = = x x = =) '(length-repeat 2 x)  len-2)
               (do-section '(= = = = = x = = x x) '(length-repeat 2 x) len-3)
               (do-section '(x x = = x = x x = =) '(length-repeat 2 x) len-4)
               (do-section '(= = = = x) '(length-repeat 2 x) len-1C))

 pizz  (append len-1 len-1 len-1 len-1C )
)
```

Since the ornament functions add additional notes to the phrase any corresponding note-length symbols need to be divided into two with **length-repeat**.

```
(def-velocity
 cello  dyn-1
 pizz  '(60)
)

(setq repeats '(2 2 1 2 2 1 1 1 2 1
                1 1 1 2 2 2 1 1 2 2
                1 1 2 2 1 2 1 1 2 3
                1 2 3 2 1))

(def-zone
cello  (zone-expand repeats (z-ratio-sc (length-of cello)))
pizz   (zone-expand repeats (z-ratio-sc (length-of cello)))
)
```

Create repeating 'zones' of pitch, symbol and velocity, once more for a dance-like effect.

```
(def-channel
cello 1
pizz 2
)


(def-tempo 140)

(compile-instrument-p "ccl;output:" "dance-figures-3"
 cello
 pizz
)
```

```
;; Nine Studies in Movement (No.4 Ornamentation)
;; NB: This code generates the basic tonal structure that was
;; later ornamented by hand.

;; material

(create-tonality tonal-s85 '(c 4 e 4 f 4 f# 4 a# 4 b 4))
(create-tonality tonal-s87 '(c 4 d 4 f 4 f# 4 g# 4 b 4))
(create-tonality tonal-s89 '(c 4 d# 4 e 4 f# 4 a 4 a# 4))
(create-tonality tonal-s91 '(c 4 c# 4 e 4 f# 4 g 4 a# 4))
(create-tonality tonal-s93 '(c 4 c# 4 d# 4 f# 4 g 4 a 4))

(create-tonality tonal-s92 '(c 4 d 4 d# 4 f# 4 g# 4 a 4))
(create-tonality tonal-s90 '(c 4 d 4 e 4 f# 4 g# 4 a# 4))
(create-tonality tonal-s88 '(c 4 d& 4 f 4 f# 4 g 4 b 4))
(create-tonality tonal-s86 '(c 4 d& 4 f 4 f# 4 g 4 b 4))

(setq s90-1 (c-pitch-to-symbol (symbol-bundle 2 '(c 4 d 4 e 4 f# 4 g# 4 a# 4))))
; (a c e g i k)


(setq mat-90 (gen-intro 5 s90-1))
; (-k -i -g -e -c a c e g i k)

(setq 9-list
(gen-collect 0.12 9 :list '(gen-random nil 9 mat-90)))
; ((g -c -c -i i i k g a) (-i -k g -g -k -i -k c a) (-g -k i -k -c -c c g -k)
; (g k g e -i -k -c -c k) (a -c e -i -e g c -i k) (c c -e -c -c -e g a i)
; (-e -k -k c -e -g -k c a) (i e -g g a -c -g a c) (k -e -i -g -c -e e k e))


(setq 9-list-length (gen-process '(symbol-repeat x y)
                                 (mapcar 'length 9-list) '(1/4) :list))

 (setq len-list
       (do-section :all '(length-variate nil 4 2 x) 9-list-length ))

;; score

(def-tonality
 cello (activate-tonality (chromatic d 4))
)
```

Convert Slonimsky pattern 9 into a series of symbols, where C4 = a and so on.

Generate a five-note introduction to the phrase.

Create nine lists of nine pitches. For each list, **gen-random** is used to make nine selections from mat-90. **Gen-collect** enables this selection process to be executed nine times.

For each pitch create a corresponding 1/4 length note.

Create 1/8 notes and rests using **length-variate**, for example: (1/4 1/4 1/4) = (1/8 1/8 –1/4 1/4).

```
(def-symbol
cello 9-list
)

(def-length
cello len-list
)

(def-velocity
cello '(64)
)

(def-zone
cello (z-ratio-sc len-list)
)

(def-channel
 cello 1
)


(def-tempo 50)

(compile-instrument-p "ccl;output:" "ornament-1"
 cello
)
```

```
;; Nine Studies in Movement (No.5 Phrases)

; Functions

; See appendix A for all the bespoke functions used in this movement

; material

(setq s85 '(a -c -b g e f)
      s86 '(a -d -b g d f)
      s87 '(a -e -b g c f)
      s88 '(a -f -b g b f)
      s89 '(a -d -c g d e)
      s90 '(a -e -c g c e)
      s91 '(a -f -c g b e)
      s92 '(a -e -d g c d)
      s93 '(a -f -d g b d))

(setq s85-x
      (c-list-rotate 0.1 (delete-lists 1 (create-lists (find-change
                                                         (cf-noise-white 256 1.0 .37 s85)))))
      s86-x
      (c-list-rotate 0.1 (delete-lists 1 (create-lists (find-change
                                                         (cf-noise-white 256 1.0 .37 s86)))))
      s87-x
      (c-list-rotate 0.1 (delete-lists 1 (create-lists (find-change
                                                         (cf-noise-white 256 1.0 .37 s87)))))
      s88-x
      (c-list-rotate 0.1 (delete-lists 1 (create-lists (find-change
                                                         (cf-noise-white 256 1.0 .37 s88)))))
      s89-x
      (c-list-rotate 0.1 (delete-lists 1 (create-lists (find-change
                                                         (cf-noise-white 256 1.0 .37 s89)))))
      s90-x
      (c-list-rotate 0.1 (delete-lists 1 (create-lists (find-change
                                                         (cf-noise-white 256 1.0 .37 s90)))))
      s91-x
      (c-list-rotate 0.1 (delete-lists 1 (create-lists (find-change
                                                         (cf-noise-white 256 1.0 .37 s91)))))
      s92-x
      (c-list-rotate 0.1 (delete-lists 1 (create-lists (find-change
                                                         (cf-noise-white 256 1.0 .37 s92)))))
```

Cf-noise-white is used to create 256 samples of white noise (e.g. random numbers between –1.0 and 1.0). These are then associated with symbols from each tonality – in this case **s85** – essentially giving us 256 random pitches drawn from **s85**. **Find-change** looks for repeated notes and changes the second into a rest (e.g. (a c c b) becomes (a c = b)). **Create-lists** defines phrases beginning at a rest symbol (so (a c = b) becomes ((a c) (= b)), **Delete-lists** then removes any lists with a length of only 1 character (e.g. single rests). Finally, **c-list-rotate** will 'rotate' the content of random lists by one character (based on a random seed of 0.1). Therefore (a b =) may become (= a b).

```
s93-x
(c-list-rotate 0.1 (delete-lists 1 (create-lists (find-change
                                                   (cf-noise-white 256 1.0 .37 s93))))))


(setq s-list (gen-random 0.1 41 (list-a-scale 0 8))) ;; order of slonimksy patterns
; (2 6 4 0 7 6 7 6 5 1 0 3 3 7 1 0 6 0 3 3 4 5 0 1 0 6 1 0 1 0 4 4 6 3 3 1 6 6 7 6 3)


(setq s-mix-list
      (flat-them (gen-process-list
                  '(p-select x y) (list-a-scale 0 41) ; sequence slots
                  (e-substitute
                   (list s85-x s86-x s87-x s88-x s89-x
                         s90-x s91-x s92-x s93-x)
                   '(0 1 2 3 4 5 6 7 8) s-list))))

#| ;; s85-x
((g e -b -c) (= g -c g) (= f e) (= a) (= g) (e =) (f -c g =) (e a e f g -c g -c =)
(= g -b -c) (= -b) (= g -c e g) (e =) (-c =) (f -b -c e -b e g -b -c e -c g e -b g -c =)
(g -c g =) (= -b f e -b) (-c f -b e f g -b a g f g =) (e -c g e g =)
(f -b g -b g e -b e -c -b e -b a -c e g f e =) (= g a) (= e -c f) (-b -c g =) (e =)
(a -c -b =) (= -c) (a -b f e -b a g -b -c -b -c g -b g a g -b =) (g f a g e -b f -c =)
(= a f -b f e -c) (e =) (-b f =) (g a -b -c =) (g a -c a -c e =) (= g -b) (= g) (= -b)
(= g) (= -b e a f g -b e g f e -c f -c a) (-b e g a e =) (= -c g e) (= g -b a -c g -b e f -c)
(f e f g e -c -b e -b f a -c g -b a e -c e a g e -c a =))

;; s-mix-list
((g c -b -e) (= g -f g) (= e d) (= a) (= g) (b =) (d -e g =) (b a b e g -f g -f =)
(= g -c -e) (= -b) (= g -c e g) (b =) (-f =) (d -d -e c -d c g -d -e c -e g c -d g -e =)
(g -d g =) (= -b f e -b) (-f e -c b e g -c a g e g =) (e -c g e g =)
(f -b g -b g b -b b -f -b b -b a -f b g f b =) (= g a) (= d -d e) (-c -e g =) (e =)
(a -d -b =) (= -c) (a -c e b -c a g -c -f -c -f g -c g a g -c =) (g f a g d -b f -d =)
(= a f -b f e -c) (d =) (-b f =) (g a -c -d =) (g a -d a -d d =) (= g -c) (= g) (= -b) (= g)
(= -c b a e g -c b g e b -f e -f a) (-c b g a b =) (= -e g c) (= g -c a -f g -c b e -f)
(f b f g b -f -b b -b f a -f g -b a b -f b a g b -f a =))

|#
```

The material from this movement will be derived from picking combinations of the nine sequences defined above. First a series of random numbers between 0 and 8 are chosen.

The selection list (**s-list**) is now used to create a sequence in which each phrase in the series is drawn from one of the nine phrase collections **s85-x** to **s93-x**. To the left you can see how the output for **s85-x** looks, compared with the aggregate version **s-mix-list**.

```
(symbol-divide (mapcar 'length s-mix-list) 'setq 'x (flatten s-mix-list))

(setq x0 '(g c -b -e))
(setq x1 '(= g -f g))
(setq x2 '(= e d))
(setq x3 '(= a))
(setq x4 '(= g))
(setq x5 '(b =))
(setq x6 '(d -e g =))
(setq x7 '(b a b e g -f g -f =))
(setq x8 '(= g -c -e))
(setq x9 '(= -b))
(setq x10 '(= g -c e g))
(setq x11 '(b =))
(setq x12 '(-f =))
(setq x13 '(d -d -e c -d c g -d -e c -e g c -d g -e =))
(setq x14 '(g -d g =))
(setq x15 '(= -b f e -b))
(setq x16 '(-f e -c b e g -c a g e g =))
(setq x17 '(e -c g e g =))
(setq x18 '(f -b g -b g b -b b -f -b b -b a -f b g f b =))
(setq x19 '(= g a))
(setq x20 '(= d -d e))
(setq x21 '(-c -e g =))
(setq x22 '(e =))
(setq x23 '(a -d -b =))
(setq x24 '(= -c))
(setq x25 '(a -c e b -c a g -c -f -c -f g -c g a g -c =))
(setq x26 '(g f a g d -b f -d =))
(setq x27 '(= a f -b f e -c))
(setq x28 '(d =))
(setq x29 '(-b f =))
(setq x30 '(g a -c -d =))
(setq x31 '(g a -d a -d d =))
(setq x32 '(= g -c))
(setq x33 '(= g))
(setq x34 '(= -b))
(setq x35 '(= g))
(setq x36 '(= -c b a e g -c b g e b -f e -f a))
(setq x37 '(-c b g a b =))
(setq x38 '(= -e g c))
(setq x39 '(= g -c a -f g -c b e -f))
(setq x40 '(f b f g b -f -b b -b f a -f g -b a b -f b a g b -f a =))
```

Symbol-divide divides **s-mix-list** into a series of variables prefixed with x. This means that we can easily manipulate the entire sequence later in the code.

```
(setq n-list (gen-random 0.1 34 (list-a-scale 0 41)))

(setq r-sym (eval-section-integer n-list 'x 'list)) ; randomized list
```

Create 34 random values between 0 and 40 and then assemble a series of phrases by picking the corresponding phrases from **x0** to **x40**.

```
#|
((g a -c -d =) (= a f -b f e -c) (b a b e g -f g -f =) (= g -f g) ; 4
(f -b g -b g b -b b -f -b b -b a -f b g f b =) (e -c g e g =)  ; 2
(f b f g b -f -b b -b f a -f g -b a b -f b a g b -f a =) (= -b) (= e d) ; 3
(= -b) (= g -f g) (-f e -c b e g -c a g e g =) (= -b f e -b) (= -c) ; 5
(g a -c -d =) (g c -b -e) (b =) (g c -b -e) (g a -c -d =) (= g -c -e) ; 6
(= e d) (d -e g =) (g c -b -e) (a -d -b =) (-c -e g =) (= g) (e -c g e g =) ; 7
(f -b g -b g b -b b -f -b b -b a -f b g f b =) (d -e g =) (= -b) (= -b) ; 4
(f b f g b -f -b b -b f a -f g -b a b -f b a g b -f a =) (= g) ; 2
(f -b g -b g b -b b -f -b b -b a -f b g f b =)) ; 1

|#

(init-rnd 0.127)

(setq output-all/model
'((rv)(rv)(rv)()
(fd)(se)
(rv)()()
()()()()(rv)
(rv)()()()()()
()()()()()()(se)
()(rv)()()
()()
(fd)))
```

This code is part of Nigel Morgan's 'scoresheet' style of programming, which is also used throughout *Six Concertos* and *Piece d'Orgue*. Each phrase in the movement is represented as a pair of brackets. Any transformation to that phrase is indicated by a two-letter 'I function'. Here, **rv** directs that symbol order should be reversed; **se** denotes symbol expansion and **fd** the deletion of all instances of a random pitch symbol (**filter-delete**).

```
(setq r-symx ; with expansions, cutting up of long phrases, retrogrades
      (do-section
       (mtypes-to-template 'fd output-all/model)
       '(filter-delete (pickn 1 x) x)
       (do-section
        (mtypes-to-template 'se output-all/model)
        '(symbol-list-expand x)
        (do-section
         (mtypes-to-template 'rv output-all/model)
         '(symbol-retrograde x) r-sym))))
```

This code processes the above defined scoresheet against **r-sym** making reversals, symbol expansions and deletions as indicated by the composer.

```
(setq r-symwx
      (do-section (mtypes-to-template 'fd output-all/model)
            '(delete-lists 1 (c-list-rotate 0.1 (create-lists x)))  r-symx))

(setq r-symxy (flat-them r-symwx))

#| ;; final phrase list

((= -d -c a g) (-c e f -b f a =) (= -f g -f g e b a b) (= g -f g) (f -b) (= -b) ;6
 (= b -b b -f -b b -b a -f b) (= f b) (i -b e -c g e g =) ; 3
(= a -f b g a b -f b a -b g -f a f -b b -b -f b g f b f) (= -b) (= e d) (= -b) ; 4
(= g -f g) (-f e -c b e g -c a g e g =) (= -b f e -b) (-c =) (= -d -c a g) (g c -b -e) ; 6
(b =) (g c -b -e) (g a -c -d =) (= g -c -e) (= e d) (d -e g =) (g c -b -e) (a -d -b =) ; 8
 (-c -e g =) (= g) (i -b e -c g e g =) (f -b g -b g b -b b -f -b b -b a -f b g f b =) ; 4
(= g -e d) (= -b) (= -b) (f b f g b -f -b b -b f a -f g -b a b -f b a g b -f a =) (= g) ; 5
(f -b g -b g) (= -b) (= -f -b) (= -b a -f) (= g f)) ; 5
|#

(setq r-len (gen-process '(symbol-repeat x y)  (mapcar 'length r-symxy) '(1/8) :list))
```

The symbol expand function results in nested lists, such as ((= - b a f)(= g f)). This code processes the lists removing any nesting with **flat-them** as well as getting rid of any redundant rest symbols created by a further execution of **create-lists.**

Define the default length for each symbol as being 1/8.

```
(setq output-all/vc
'((gi )(cl)()(gt xl)(gt lr)(cl)
  (sr xl)(xl gi)(dn xl)
  (lv)(cl)(cl)()
  (gt xl)(fg xl)(gi xl)(cl)(gi xl)(gc xl)
  (cl)(sr xl)(sr xl)(gi xl)(gi xl)(gc xl)(sr xl)(sr xl)
  (dn xl)()(up xl)(dl lv)
  (gi)(cl)()(lv)(xl sr)
  (gt xl)(xl sr)(xl sr)(gi xl)(gc xl)))
```

This is a second scoresheet for processing both the length and pitch symbols. Some of the I Functions indicated here relate to pitch and some to length. Sometimes pairs are necessary, for example **fg** indicates **symbol-figurate**, which extends the number of symbols in a phrase. It is therefore paired with **xl** – **extend lengths** – which will ensure that there are corresponding note lengths to indicate the ornamental figuration of the phrase.

```
(setq pp '(35 30)
      p '(45 40)
      mp '(60 55)
      mf '(75 70)
      f  '(96 90)
      ff '(115 110))
```

```
(setq dyn-1 (list p mp mf p mp p
                  mp p mf
                  f p mp p
                  mf f mp mf  mp mf
                  f mp mf p mp mf mp mf
                  f mp mf f
                  p mp mf f mf
                  f mp p mp mf))

(setq zone-ex
'(1 1 1 1 1 1
  1 1 1
  1 1 1 1
  2 1 1 1 1 1
  1 1 1 1 1 1 1 1
  2 1 1 1
  1 1 1 1 1
  1 1 1 1 1
))
```

Throughout the *Studies*, zone lengths are analogous to the lengths of phrases. Zones can also be expanded to create repetitions of the material they contain. Here material for zone expansion is defined.

```
;; symbol processing

(init-rnd 0.127)

(setq vc-sym
(do-section
(mtypes-to-template 'st output-all/vc)
'(sequence-transpose x)
(do-section
(mtypes-to-template 'se output-all/vc)
'(symbol-list-expand x)
(do-section
(mtypes-to-template 'd1 output-all/vc)
'(distort-transpose 1 x)
(do-section
(mtypes-to-template 'gp output-all/vc)
'(gen-palindrome  x)
(do-section
(mtypes-to-template 'gc output-all/vc)
'(g-coda (length x) x)
(do-section
(mtypes-to-template 'gi output-all/vc)
'(g-intro (length x) x)
```

The following code processes the pitch material in accordance to the I Functions indicated on the scoresheet. So in the highlighted example an instance of **(st)** in the scoresheet will cause the sequence-transpose function to be executed on the corresponding phrase.

```
(do-section
(mtypes-to-template 'gt output-all/vc)
'(g-tremelo  x)
(do-section
(mtypes-to-template 'sr output-all/vc)
'(symbol-repeat 2  x)
(do-section
(mtypes-to-template 'd2 output-all/vc)
'(distort-transpose -1 x)
(do-section
 (mtypes-to-template 'su output-all/vc)
 '(symbol-upward x)
(do-section
 (mtypes-to-template 'sd output-all/vc)
 '(symbol-downward x)
(do-section
(mtypes-to-template 'sh output-all/vc)
'(symbol-harmonize nil 'mix -7 5 x)
(do-section
 (mtypes-to-template 'mr output-all/vc)
 '(make-rest x)
(do-section
 (mtypes-to-template 'ts output-all/vc)
 '(symbol-thin 2 5 x nil)
(do-section
 (mtypes-to-template 'fg output-all/vc)
 '(symbol-figurate x)
(do-section
(mtypes-to-template 'up output-all/vc)
'(upward x)
(do-section
(mtypes-to-template 'dn output-all/vc)
'(downward x)
(do-section
(mtypes-to-template 'ud output-all/vc)
'(up-down x)
(do-section
(mtypes-to-template 'du output-all/vc)
'(down-up x)
(do-section
(mtypes-to-template 'fl output-all/vc)
'(floating x)
r-symxy)))))))))))))))))))))
```

```
;; length processing here

(setq r-len
 (gen-process '(symbol-repeat x y)  (mapcar 'length vc-sym) '(1/8) :list))

(init-rnd 0.127)

(setq vc-len
(do-section (mtypes-to-template 'rr output-all/vc)
'(l-rest-revert x)
(do-section (mtypes-to-template 'lv output-all/vc)
'(length-variate nil (get-random 2 4) 2 x)
(do-section (mtypes-to-template 'xl output-all/vc)
'(change-length :divide 2 x :ratio)
(do-section (mtypes-to-template 'cl output-all/vc)
'(change-length :times 2 x :ratio)
 (do-section (mtypes-to-template 'zl output-all/vc)
'(change-length :times 4 x :ratio)
(do-section (mtypes-to-template 'lr output-all/vc)
'(length-repeat 2 x)
 r-len )))))))

;; score

(def-tonality
cello (activate-tonality (chromatic e& 4))
)

(def-symbol
cello vc-sym
)

(def-length
cello vc-len
)

(def-velocity
cello dyn-1
)
```

```
(def-zone
cello (zone-expand zone-ex (z-ratio-sc vc-len))
;(5/4 7/4 9/8 1/2 1/2 1/2 11/8 3/8 27/16 3/1 1/2 3/4 1/4 1/1 3/2 5/8 1/2 5/8 1/2 1/2
; 1/2 5/8 1/2 3/8 1/2 1/2 1/2 1/1 1/4 21/16 19/8 1/1 1/2 1/4 3/1 1/4 5/8 1/4 3/8 1/2 3/8)
)

(def-channel
cello 1
)

(def-tempo 90)

(compile-instrument-p "ccl;output:" "phrases-1"
cello
)
```

Appendix A - Additional functions

```
;; These will need to be evaluated before running the code for movement five.

;; zone expand

(defun zone-expand (x-by zne-lis) ; adjusted 15.3.04
"expanding values of chosen zone-lengths to create repeats"
  (prog ( out)
    loop
    (cond ((null zne-lis) (return (get-ratio-sc out))))
    (setq out (append out (list (* (car x-by) (get-ratio-cl (car zne-lis))))))
    (setq x-by (cdr x-by))
    (setq zne-lis (cdr  zne-lis))
    (go loop)))


;; length variate

(defun length-variate (seed count divide l-lengths)
"produces rhythmic variants whilst keeping symbol pattern intact provided division is by 2"
 (diagnostic2 "length-variate" $cr$)
(if (eq count 0)
  (l-rest-revert l-lengths)
  (l-divide seed count divide nil nil
            (symbol-shuffle (length-masking count l-lengths seed)))))


;; gen coda

(defun gen-coda (n lis &optional t-length)
"enables generation of further n symbols in a sequence
 defaults to chromatic"
(if t-length
  (append lis (symbol-trim n (symbol-transpose t-length lis)))
  (append lis (symbol-trim n (symbol-transpose 12 lis)))))
```

```
;; gen intro

(defun gen-intro (n lis &optional t-length)
"enables generation of further n symbols in a sequence
 defaults to chromatic"
(if t-length
  (append (symbol-trim-r n (symbol-transpose t-length lis)) lis)
  (append (symbol-trim-r n (symbol-transpose -12 lis)) lis)))



;; eval section integer
(defun eval-section-integer (section-list symbol-affix how)
"variant of eval-section enables use of integer lists - x0 x1 x2 etc"
(diagnostic2 "eval-section-r" $cr$)
  (prog (out)
    loop
    (cond ((null section-list)
           (return (cond ((equal how 'append) (eval-list out))
                         ((equal how 'list) (mapcar 'eval out))))))
    (setq out (append out
                      (list (compress (list
                             symbol-affix (car section-list))))))
    (setq section-list (cdr section-list))
    (go loop)))

;; gen process list
(defun gen-process-list (f-expr values patterns)
"processes a list of lists with a list of differing values
- gen-process only allows a single list to be processed "
(diagnostic2 "gen-process-list" $cr$)
(setq f-expr (eval (list 'function
                           (append '(lambda) (list '(x y) f-expr)))))
  (prog (out)
    (let* ((initial diagnose-verbose)
           (diagnose-verbose nil))
      (setq out (mapcar f-expr values
                         patterns ))
      (setq diagnose-verbose initial))
           (return out)))
```

```
;; create lists

(defun create-lists (lisx)
"creates lists using rest symbols to mark divisions"
  (symbol-divide
   (reverse (mapcar 'abs
               (do-section :all '(apply '- x)
                  (symbol-divide '(2 (-1)) nil nil
               (reverse (append (e-position '= lisx)
                   (list (length lisx))))))))
                         nil nil lisx))


;; delete lists

(defun delete-lists (value lisy)
  "delete lists equal to or below a length value - adjunct to create-lists"
  (prog (out el)
    loop
    (cond ((null lisy) (return (delete 'nil out))))
    (setq el (car lisy))
    (setq out (append out
                      (list (cond ((>= value (length el)) ())
                                  (t el)))))
    (setq lisy (cdr lisy))
    (go loop)))

;; c-list-rotate

(defun c-list-rotate (seed lisz)
"rotates contents of lists at random except for the first list - necessary part of create-lists"
(do-section (p-replace nil 'first '= (gen-template seed 1 1 (length lisz)))
            '(symbol-scroll -1 x) lisz))
```

```lisp
;; Flat them

(defparameter *flat-them-stack* nil)

(defun flat-them-sup (l)
  (cond ((null l) nil)
        ((is-flat l)
           (push l *flat-them-stack*))
        (t
         (flat-them-sup (car l))
         (flat-them-sup (cdr l)))))

(defun flat-them (l)
  (setq *flat-them-stack* nil)
  (flat-them-sup l)
  (nreverse *flat-them-stack*))

;; Mtypes to template

(defun mtypes-to-template (mtypes array-output)
"produces a template list from array-output equivalent to instrument-to-string"
  (do-quietly
(e-substitute '(x) '(b)
     (e-substitute '(=) '(a)
                   (mapcar 'integer-to-symbol
                           (flatten (mapcar
                                     (function (lambda (x)
                                                 (e-count mtypes x))) array-output)))))))
```

```
;; Symbol list expand

(defun symbol-list-expand (lis &optional seed)
"expands symbol-list in using by selection of content range
 and randomized repetition"
(if seed
(init-rnd seed))
(if (is-rest (car lis))
 (let ((selection (pick-random '(:this :that :other))))
         (case selection
           (:this
    (append lis (symbol-trim-r  (get-random 1 (- (length lis) 1)) lis )))
           (:that
 (append (symbol-trim  (get-random 1 (- (length lis) 1)) lis )
                      (cdr lis )))
(:other
 (append  lis (distort-transpose 1
                (symbol-trim-r  (get-random 1 (- (length lis) 1)) lis ))))))
(let ((selection (pick-random '(:this :that :other))))
         (case selection
           (:this
    (append (symbol-trim  (get-random 1 (- (length lis) 1)) lis) lis ))
           (:that
 (append (symbol-trim-r  (get-random 1 (- (length lis) 1)) (reverse lis))
                      (cdr lis )))
           (:other
           (append (but-last (reverse (distort-transpose 1
              (symbol-trim  (get-random 1 (- (length lis) 1)) lis))))
                 lis ))))))
```

```
;; Distort transpose

(defun distort-transpose (value pattern &optional rnd)
"distorts a phrase incrementally by transposition"
(let ((out) (element rnd))
(dotimes (i (length pattern))
(if rnd
(setq element (transpose-symbol
                    (nth i pattern)
                    (pick-random (g-integer 0 (length pattern)))))
(if  (< 1 value)
(setq element (transpose-symbol
                    (nth i pattern) (+ i value)))
(if  (minusp value)
(setq element (transpose-symbol
                    (nth i pattern) (* i value)))
(setq element (transpose-symbol
                    (nth i pattern) i)))))

(push element out))
(nreverse out)))

;; Pickn

(defun pickn (n lis )
  (prog (out)
    (dotimes (i n)
      (setq out (append out
                        (list (pick-random lis)))))
    (return out)))

;; Downward

(defun downward (lis)
 (flatten
(do-section :all '(symbol-inversion 'a x)
            (mapcar 'symbol-upward (gen-variants-tx nil (get-random 2 4) nil lis)))))
```

```
;; Gen variants tx

(defun gen-variants-tx (seed n p pat)
  (diagnostic2 "gen-variants-tx" $cr$)
  (prog (out temp)
    (if (null pat) (return nil))
    (setq temp diagnose-verbose)
    (setq diagnose-verbose nil)
    (if (null p) (setq p 1))
    (if seed (init-rnd seed))
    (for i p 1 n nil
      (setq out
            (append out (list (symbol-transpose i
                                                (gen-random-successive (rnd)
                                                            (get-random 2 (length pat)) pat))))))
    (setq diagnose-verbose temp)
    (return (append (list pat) out))))

;; Symbol upward

(defun symbol-upward (lis)
(fill-template lis
(ornament-higher 12 (sort-ascending (filter-delete '= lis)))))

;; Upward

(defun upward (lis)
  (flatten (mapcar 'symbol-upward (gen-variants-tx nil (get-random 2 4) nil lis))))

;; Symbol figurate

(defun symbol-figurate (lis)
"figurates from within the symbols of the source pattern"
(ornament-lower 12 ; adds chromatic inflections
(symbol-interleave lis  (find-change (nthcdr (length lis) (gen-variants nil 1 lis))))))

;; G-tremelo
(defun g-tremelo (lis)
"adds tremelo to a phrase doubling its length -
takes the tremelo notes from first or last symbols in the list"
(if (is-rest (car lis))
  (add-tremelo lis (list (last lis)))
 (append (but-last (add-tremelo lis (list (first lis)))) '(=))))
```

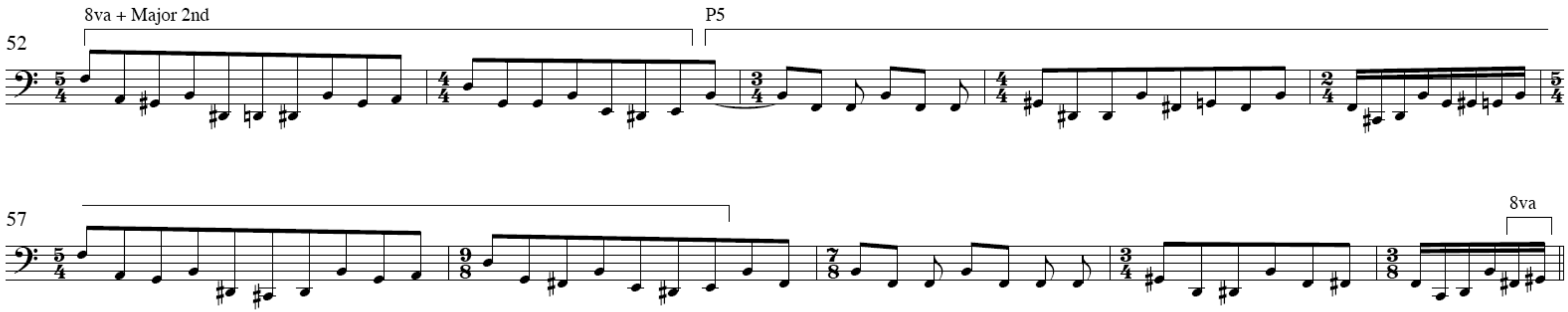
```
;; g-intro

(defun g-intro (value lis)
"use with length-repeat"
  (prog (out el)
    (setq el (gen-intro value lis))
    (setq out
          (cond ((is-pause-symbol
                  (car el))
                  (append '(=) (p-remove value el)))
        (t (append (p-remove (- value 1) el) '(=)))))
    (return out)))

;; g-coda


(defun g-coda (value lis)
"use with length-repeat"
  (prog (out el)
    (setq el (gen-coda value lis))
    (setq out
          (cond ((is-pause-symbol
                  (car el))
                 (append '(=) (p-remove value el)))
         (t (append (p-remove (- value 1) el) '(=)))))
    (return out)))
```

Appendix B

In the composition of *Studies in Movement* the code has generally been written to create finished compositions with complete pitch, length and dynamic schemes in place. The exceptions to this being:

1) Continuum – transposed bars

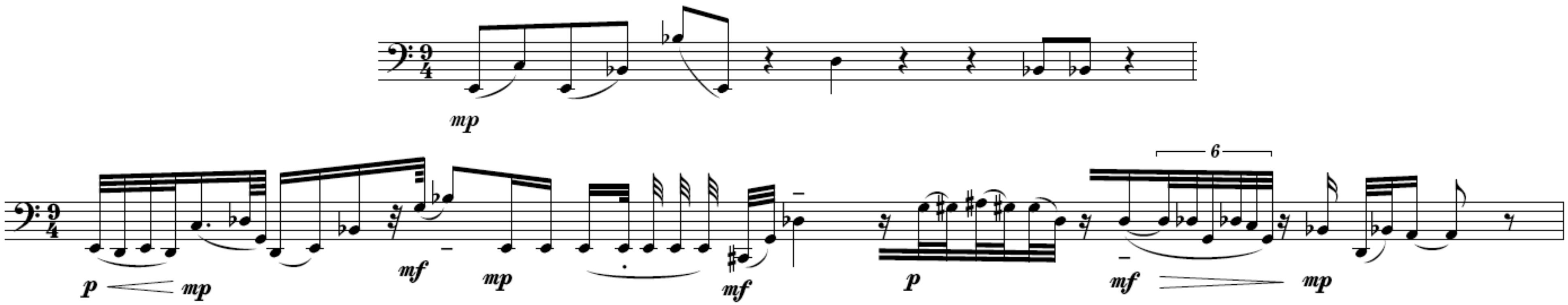
In the editing of the work it was decided that transpositions would add more life to a movement that was generally centred around a single octave. While these transpositions could have easily been incorporated into the code the decision was made at a later stage in the editing process of the piece.



Original output from the code for *Continuum*, showing passages transposed in the final score.

2) Ornamentation

The code for ornamentation provides the skeletal structure for a piece which a player may ornament as s/he sees fit. In the score for *Studies in Movement*, Nigel Morgan has provided his own ornamented version based on figures common to Turkish *Maqâm* forms of improvisation on the *ud*.



Initial phrase of the movement as generated by SCOM, and consequent ornamental interpretation.