Quatuor des Timbres

For an ensemble of mixed timbres

Symbolic Composer score annotations

Nigel Morgan

Quatuor des Timbres

For an ensemble of mixed timbres

Nigel Morgan

This four-movement work for an ensemble of mixed timbres was created entirely from algorithmic constructs within the Lisp software Symbolic Composer. It is the foundation of the *Instrumentarium Novum* series that resulted in the composition of *Six Concertos* for (self-directed) orchestra. The annotation presented here is particularly significant for two reasons: it is one of the only annotated scores in the present archive which involves the use of SCOM's Class System; it contains in the code for the fourth movement, an instrumental setting of Psalm 19, which explains how a vocal score might be composed using algorithmic techniques.

The Class System (CS) within SCOM was created by Pekka Tolonen in 1997 and was subsequently documented and annotated in a Guide to the Class System by Nigel Morgan in 1998. Although Nigel Morgan uses the System from time to time in his compositions, it has never proved a robust replacement for the Zone-Control technique he developed in 1991. CS requires the composer to start from an exact definition of zone length, rather as an artist might work by deciding how large a canvas might be and then defining the relative size of each structural part of the image. Zone-Control by contrast is quite the reverse: as phrases of pitch and note-length are generated they are collected together into zones, and in such a way that any subsequent alteration to phrase size immediately changes the size and partitioning of the zone definition needed to compile the music.

One of the major influences on *Quatuor* was the use of graphic scores by Milton Babbitt and Morton Feldman. In *Composition for Four Instruments* Babbitt devised a simple beat/space system that enabled him to sketch out a display of sonic activity by the four instruments. This system indicates whether an instrument is busy or inactive, making sounds or keeping silent. Feldman's box/grid system for *IXION* is a little more sophisticated, but designed to enable the composer to create a quick visual sketch showing the density and register of sound organisation with a defined temporal 'moment'. Both these approaches are recreated in an algorithmic format within *Quatuor des Timbres*. Just as Babbitt and Feldman used their graphic methods to make rough prototypes of musical action for ensemble performance, leaving the musical detail to improvisation or imagination, Nigel Morgan is able to sketch entire prototype movements of a composition to explore the effects of applying different rules and constraints, and filling these temporary structures with algorithmically generated musical detail.

In Movement I *Four Necessary Chorales* and Movement IV – *In Finem Psalmus David* the focus has been on adopting techniques and musical practices from the past, but with a contemporary algorithmic slant. Of these two, Movement IV is the most groundbreaking in terms of algorithmic practice as the rhythm of the text of Psalm 19 generates the pitch and phrase structure. This algorithmic approach was originally developed in the BBC commission *Schizophonia* (2001) and then in an unaccompanied vocal setting of Psalm 19.

```
;; Quatour des Timbres movement 1 i: Four Necessary Chorales – Chorale 1

;; functions

(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))
```

This code creates a new function that divides a set of symbols into lists based on the presence of rest symbols (=). For example:

`(create-lists  '(a b c = = e f g = f g a = c e))`
yields: `((a b c) (= e f g) (= f g a) (= c e))`

```
;; material

(setq mat (vector-to-symbol a l (gen-noise-white 128 1.0 0.5)))
```

Use `gen-noise-white` to create 128 ‘samples’ of white noise in the range –1.0 to 1.0. The value 0.5 is used as the random seed. These samples are then mapped to a series of symbols in the range ‘a to ‘l. This results in a series like this: `(f j l j j c f f j g g a c ...`

```
(setq mat1-4 (symbol-divide 30 nil nil (find-anacrusis mat))
      m1 (first mat1-4)
      m2 (second mat1-4)
      m3 (third mat1-4)
      m4 (fourth mat1-4)
      wd-1 (create-lists m1)
)
```

This code first uses the `find-anacrusis` function on the previously generated list of symbols. This searches the list for duplicated values and replaces the first of them with a rest. So `‘(f j l j j c)` becomes `‘(f j l = j c)`. The processed list is then divided into groups of 30, which are distributed between the variables `m1-m4`. This first chorale is based on the symbols in `m1`. The previously defined create-lists function is used to further divide the material.

```
(setq chord1 (g-chord 0.9 3 3 -24 -6
; (length chords-n) eq (length (/ symbol-melody 2))
                       (do-section :all
                               '(filter-delete '= x) wd-1))
)
```

Expand the processed material (`wd-1`) into a series of chords. Rests are removed from the list of material before it is passed to the `g-chord` function. In this instance a random seed of 0.9 is used and where possible chords should have both a minimum and maximum size of three symbols. Chord values may be transposed between –24 and –6 steps. The result looks like this:
`((-i-e-c) (-g-n) (-t-p) (-l-r-p -o-n-h -g-e -b-d) ...`

```
(setq st-4 (do-section :all '(symbol-demix 3 x :sort) chord1)
      pc-3 (do-section :all '(symbol-demix 1 x :sort) chord1)
      br-2 (do-section :all '(symbol-demix 2 x :sort) chord1)
)
```

Here the chords are broken up and distributed among the different instruments – strings, percussion and brass. The `symbol-demix` function sorts each chord into ascending order and pulls out a specified value – so the third value of each chord (where it exists) becomes part of the string layer: `((-i) (=) (=) (-r -o = =) (-w e) (=))`

```
(setq zones-1 (make-zone-list wd-1 (build-list '(1/8) (length wd-1))))
```

A set of 'zones' analogous to time-signatures or bar lengths are derived by considering each symbol in wd-1 to be 1/8 so: `((f j l) (= j c) (= f j) (= g a c b c i a c a i g) (= e a j b d l) (= f))` equates to the zone lengths `(3/8 3/8 3/8 3/2 7/8 1/4)`.

```
(setq wd-len (build-list '(1/8) (length wd-1))
      br-len '((3/8)(3/8)(3/8)(1/2 1/4 1/2 1/4)(5/8 2/8) (1/4))
      pc-len '((1/64)(1/64)(1/64) (1/4 1/4 1/2 1/2)(5/8 2/8) (1/64))
      st-len '((3/8)(3/8)(3/8) (1/2 1/2 1/4 1/4 )(5/8 2/8) (1/4))
)
```

Having defined symbol and zone information the lengths for each note in the various instrumental parts are now defined. A value of 1/8 is given to all the notes in the woodwind layer, while a series of lengths are hand composed for the others.

```
(setq dyn1 (symbol-divide (mapcar 'length wd-1) nil nil
                          (do-section :all '(vector-to-list
                                      (symbol-to-velocity 35 65 3 x)) m1)))
```

Finally a dynamic scheme is created which is dependent on the symbols – the original series of symbols from 'a to 'l is mapped to velocity values from 35 to 65.

```
;; score

(def-orchestra 'ensemble-1
 quartet (wd br pc st)
)
```

Quatuor des Timbres was composed using elements of SCOM's Class System. This allows the creation of hierarchical structures that can be easily accessed and populated with data. In this case an orchestral group named ensemble-1 is created. Within this is contained the subgroup quartet, consisting of four elements: woodwind, brass, percussion and strings.

```
(def-section a
default
 zone  zones-1
 velocity dyn1
 tonality (activate-tonality (chromatic e 5))
wd
 symbol wd-1
 length  wd-len
br
 symbol br-2
 length br-len
pc
 symbol (do-section '(x x x = = x) '(symbol-melodize x)
                                     (do-section :all '(make-octave x 12 ) pc-3))
 length pc-len
 velocity (append (gen-repeat 3 (list (gen-cresc-dim 0 80 24))) '((64 70 54 64))
                  '((67 56)) (list (gen-cresc 0 70 12)))
st
 symbol st-4
 length st-len
)

(def-channel
wd 1
br 2
pc 3
st 4
)

(def-program gm-sound-set
wd soprano-sax
br trombone
pc marimba
st cello
)
```

Here a section of the music (a) is being defined. This section has several 'global' parameters defining zone length, dynamic range and tonality. In this case the tonality is chromatic e 5, so the symbol a = e5, b = f5, c = f#5 and so on. These can be overridden by each part of the ensemble, as happens in the percussion part's velocity scheme. Each instrumental layer has its symbolic and length data passed to it. The percussion layer (a marimba in this movement) is more complicated since a template is used along with `symbol-melodize` to create tremolo-type ornaments. Note that the length of the zones in question, previously defined, is (1/64) this means that all the notes in the zone in question will be that length.

In this section of movement I, the wind layer plays what might be called the 'nuclear melody', supported by the other instruments playing chordal textures generated from it with the g-chord function.

```
(def-controller gm-controllers
  (wd  panning  '((20)))
  (br   panning  '((50)))
  (pc   panning  '((80)))
  (st   panning  '((120)))
)

(def-tempo 40)


(play-file-p "chorale-1"
quartet '(a)
)
```

Generate and play a MIDI file called chorale-1. This file is created by passing the previously defined section (a) to the quartet object. As you can see, lengthy compositions can quickly be built up in terms of section classes with repeats and so on easily managed.

```
;; Quatour des Timbres movement 1: Four Necessary Chorales – Chorale 2

;; functions

(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))

;; material

(setq mat (vector-to-symbol a l (gen-noise-white 128 1.0 0.5)))

(setq mat1-4 (symbol-divide 30 nil nil (find-anacrusis mat))
      m1 (first mat1-4)
      m2 (second mat1-4)
      m3 (third mat1-4)
      m4 (fourth mat1-4)
      wd-1 (create-lists m1)
      br-1i (create-lists m2)
)

(setq chord2 (g-chord 0.9 3 3 -24 -6 (do-section :all
                                                 '(filter-delete '= x)
                                                 br-1i))
)

(setq st-4i (do-section :all '(symbol-demix 3 x :sort) chord2)
      pc-3i (do-section :all '(symbol-demix 2 x :sort) chord2)
      wd-2i (do-section :all '(symbol-demix 1 x :sort) chord2)
)
```

The three subsequent sections of movement 1 are based on the same methods described above, only in each movement the set of symbols used changes as does the instrument which plays the complete melody – in this case the brass layer (br).

```
(setq  zones-2 (make-zone-list br-1i (build-list '(1/8) (length br-1i)))
)

(setq br-leni (build-list '(1/8) (length br-1i))
      wd-leni '((1/4 1/2 1/2 1/2 1/2 1/8 1/8)(1/4)(4/8 1/8)(-1/8 2/8))
      pc-leni '((1/4 1/4 1/4 1/4 1/4 -1/1 1/8 1/8)(1/64)(4/8 1/8)(1/64))
      st-leni '((-1/4 1/1 1/4 1/4 1/2.)(1/4)(4/8 1/8)(-1/8 2/8))
)

(setq dyn2 (symbol-divide (mapcar 'length wd-1) nil nil
                         (do-section :all '(vector-to-list
                                            (symbol-to-velocity 35 65 3 x)) m2)))

;; score

(def-orchestra 'ensemble-1
 quartet (wd br pc st)
)

(def-section b
default
 zone  zones-2
 velocity dyn2
 tonality (activate-tonality (chromatic e 5))
wd
 symbol (do-section :all '(symbol-transpose 12 x) wd-2i)
 length  wd-leni
br
 symbol (do-section :all '(symbol-transpose -12 x) br-1i)
 length br-leni
 velocity (do-section :all '(change-length :add 20 x) dyn2)
pc
 symbol (do-section '(= x = x) '(symbol-melodize x)
                                     (do-section :all '(make-octave 12 x ) pc-3i))
 length pc-leni
 velocity (append '((64 70 54 64 )) (gen-repeat 3 (list (gen-cresc-dim 0 80 24)))
                  '((67 56)) (list (gen-cresc 0 70 12)))
st
 symbol st-4i
 length st-leni
)

(def-channel
wd 1
br 2
pc 3
st 4
)

(def-program gm-sound-set
wd soprano-sax
br trombone
pc marimba
st cello
)

(def-controller gm-controllers
  (wd  panning  '((20)))
  (br   panning  '((50)))
  (pc   panning  '((80)))
  (st   panning  '((120)))
)

(def-tempo 40)


(play-file-p "chorale-2"
quartet '(b)
)
```

```
;; Quatour des Timbres movement 1: Four Necessary Chorales – Chorale 3

;; functions

(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))

;; material

(setq mat (vector-to-symbol a l (gen-noise-white 128 1.0 0.5)))

(setq mat1-4 (symbol-divide 30 nil nil (find-anacrusis mat))
      m1 (first mat1-4)
      m2 (second mat1-4)
      m3 (third mat1-4)
      m4 (append (fourth mat1-4)(fifth mat1-4))
      pc-1ii (create-lists (cdr m3))
)

(setq chord3 (g-chord 0.9 3 3 -24 -6 (do-section :all
                                             '(filter-delete '= x)
                                             (create-lists (cdr m3))))
)

(setq st-4ii (do-section :all '(symbol-demix 3 x :sort) chord3)
      wd-3ii (do-section :all '(symbol-demix 1 x :sort) chord3)
      br-2ii (do-section :all '(symbol-demix 2 x :sort) chord3)
)
```

```
(setq zones-3 (make-zone-list pc-1ii (build-list '(1/8) (length pc-1ii))))

(setq pc-lenii (build-list '(1/64) (length pc-1ii))
      br-lenii '((-1/2 1/4 1/4 1/4 1/4 1/4)(-1/2 1/4 1/4 1/4 1/8)(-1/4 1/4))
      wd-lenii '((-1/2 1/4 1/4 1/4 1/4 1/4)(-1/2 1/8 1/4 1/4 1/4)(-1/4 1/4))
      st-lenii '((-1/2 1/4 1/4 1/4 1/4 1/4)(-1/2 -1/4 1/4 1/4 1/8)(-1/4 1/4))
)

(setq dyn3 (symbol-divide (mapcar 'length pc-1ii) nil nil
                         (do-section :all '(vector-to-list
                                            (symbol-to-velocity 35 65 3 x)) m3)))

;; score

(def-orchestra 'ensemble-1
 quartet (wd br pc st)
)

(def-section c
default
 zone  zones-3
 velocity dyn3
 tonality (activate-tonality (chromatic e 5))
wd
 symbol (do-section :all '(symbol-transpose 12 x) wd-3ii)
 length  wd-lenii
br
 symbol br-2ii
 length br-lenii
pc
 symbol (do-section :all '(symbol-repeat 4 x) pc-1ii)
 length pc-lenii
 velocity '(30 40 50 40)
st
 symbol st-4ii
 length st-lenii
)
```

Note that percussion part is notated in 1/64 durations. Later the symbols for the part are repeated giving the impression of notes played with tremolo.

```
(def-channel
wd 1
br 2
pc 3
st 4
)

(def-program gm-sound-set
wd soprano-sax
br trombone
pc marimba
st cello
)

(def-controller gm-controllers
  (wd  panning  '((20)))
  (br   panning  '((50)))
  (pc   panning  '((80)))
  (st   panning  '((120)))
)

(def-tempo 40)


(play-file-p "chorale-3"
quartet '(c)
)
```

```
;; Quatour des Timbres movement 1: Four Necessary Chorales – Chorale 4

;; functions

(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))


;; material

(setq mat (vector-to-symbol a l (gen-noise-white 128 1.0 0.5)))

(setq mat1-4 (symbol-divide 30 nil nil (find-anacrusis mat))
      m1 (first mat1-4)
      m2 (second mat1-4)
      m3 (third mat1-4)
      m4 (append (fourth mat1-4)(fifth mat1-4))
      st-1iii (p-remove 1 (create-lists m4))
)

(setq chord4 (g-chord 0.9 3 3 -24 -6 (do-section :all
                                                '(filter-delete '= x) st-1iii)))


(setq br-4iii (do-section :all '(symbol-demix 3 x :sort) chord4)
      pc-3iii (do-section :all '(symbol-demix 1 x :sort) chord4)
      wd-2iii (do-section :all '(symbol-demix 2 x :sort) chord4)
)
```

```
(setq zones-4 (make-zone-list st-1iii (build-list '(1/8) (length st-1iii))))

(setq st-leniii (build-list '(1/8) (length st-1iii))
      br-leniii '((1/8 1/8 1/8 1/8 1/1 -1/8 1/8 1/8 1/8 1/1 -1/2)(3/8 3/8 3/8))
      wd-leniii '((-1/2 1/8 1/8 1/8 1/8 1/2 -1/4 1/8 1/8 1/4 1/2. -1/2)(3/8 3/8 3/8))
      pc-leniii '((-1/1 1/8 1/8 1/8 1/8 1/2 1/2 1/2 1/4 1/4)(3/8 3/8 3/8))
)


(setq dyn4 (symbol-divide (mapcar 'length pc-1ii) nil nil
                         (do-section :all '(vector-to-list
                                           (symbol-to-velocity 35 65 3 x)) m4)))

;; score

(def-orchestra 'ensemble-1
 quartet (wd br pc st)
)

(def-section d
default
 zone zones-4
 velocity dyn4
 tonality (activate-tonality (chromatic e 5))
wd
 symbol (do-section :all '(symbol-transpose 12 x) wd-2iii)
 length  wd-leniii
br
 symbol br-4iii
 length br-leniii
pc
 zone (flatten pc-leniii)
 length '((-1/1) (1/8) (1/8) (1/8) (1/8) (1/64) (1/64) (1/64) (1/4) (1/4)
         (1/4 1/8) (1/4 1/8) (-1/4 1/8))
 symbol (do-section '(= = = = = x x x = = = = =) '(symbol-melodize x )
         (symbol-divide 1 nil nil (make-octave 12 (flatten pc-3iii))))
st
 symbol  (do-section :all '(symbol-transpose -12 x) st-1iii)
 length st-leniii
 velocity (do-section :all '(change-length :add 20 x) dyn4)
)

(def-channel
wd 1
br 2
pc 3
st 4
)

(def-program gm-sound-set
wd soprano-sax
br trombone
pc marimba
st cello
)

(def-controller gm-controllers
  (wd  panning  '((20)))
  (br   panning  '((50)))
  (pc   panning  '((80)))
  (st   panning  '((120)))
)

(def-tempo 40)


(play-file-p "chorale-4"
quartet '(d)
)
```

```
;; Quatour des Timbres -  movement 2: Giuoco delle Coppie (The Game of Pairs)

;; from section 1 (six initial b)

(setq mat (vector-to-symbol a l (gen-noise-white 128 1.0 0.5))
      dyn (vector-round 35 75 (gen-noise-white 256 1.0 0.5))
)
```

The second movement begins with the same symbolic material as the first, in addition to this is a set of dynamic values between 35 and 75 which are generated from 256 samples of white noise using the same random seed as the symbolic material (0.5).

```
;; functions

(defun instrument-to-string (instrument array-output)
  (do-quietly
    (symbol-to-string
     (e-substitute '(=) '(a)
                   (mapcar 'integer-to-symbol
                           (flatten (mapcar
                                     (function (lambda (x)
                                   (e-count instrument x)))
                                     array-output)))))))
```

In this piece the orchestration is controlled by different groupings for each zone – for example ((wd st)(wd br)) means woodwind and strings play in zone 1, followed by woodwind and brass in zone 2. This function takes two arguments – instrument and array-output. The first of these indicates the instrumental section – e.g. wd, while the second is the list of zone orchestrations. The function checks each zone for the presence of the instrument, returning '-' is present or ' ' if absent. The final output for woodwind could look something like this '(--- --- - …)

```
(defun pattern-to-scale (lis)
(car (sort-tonality (list (c-pitch-to-tonality
(c-symbol-to-pitch (find-unique lis)))))))
```

Convert a pattern, such as '(f a d = a b) to a tonality, or scale. The pattern is reduced to its unique elements '(f a d = b), and then c-symbol-to-pitch is used to convert the symbols to chromatic pitches, where a = c4: '(f#4 c4 d#4 c#4). These pitches are then converted to a tonality definition using c-pitch-to-tonality and sort-tonality.

```
(defun patterns-to-scales (symbol-lists)
 (cond ((is-flat symbol-lists)
        (pattern-to-scale symbol-lists))
        (t (mapcar 'pattern-to-scale symbol-lists))))

 (defun symbol-to-string (s)
   (let ((out ""))
     (dolist (x s)
       (if (equal x '=)
         (setq out (str-cat out " "))
         (setq out (str-cat out "-"))))
     out))
```

This function will render a series of symbols into a string to be used as a template for further processing. So `'(a = b c d = f)` would become `'(- --- -)`.

```
(defun string-to-symbol (symbol string)
(let (out)
(dovector (x string)
 (if (equal x #\Space)
 (push '= out)
 (push symbol out)))
(nreverse out)))
```

Similar to symbol-to-string but will turn a series of dashes and spaces (also known as beat/space notation) into a series of symbols and rests. For example, `'(- --- -)` yields `'(a = a a a = a)`.

```
#|

(setq b-ens '(wd br pc st))
```

Define a list of items representing the ensemble – wood, brass, percussion and strings.

```
(setq b-ens1 (power-set '(wd br pc st))
      ps-len (* (length b-ens1) 2)
)
```

To items are defined here. B-ens1 contains the powerset of the list of ensemble elements. This set presents every combination of the items in the group: `((wd br pc) (wd br) (wd br st) (wd pc) (wd)...` ps-len is defined to hold a numerical value: the number of items in the powerset multiplied by two.

```
(setq b-ens2
(build-array
 ; column 0     1     2     3
       '((wd    br    pc    st )    ; 0
         (pc    br    st    wd )    ; 1
         (st    wd    pc    br )    ; 2
         (br    st    wd    pc )))  ; 3
)
```

Build an array containing elements describing the ensemble: wd = woodwind, br = brass, pc = percussion, st = strings

```
(setq lisn '(0 1 2 3)
      lisx '(1 2 3 4)
)
```

Define two groups containing integers. These will be used in conjunction with pick1 to quickly generate values between 0 - 3 and 1 - 4.

```
(setq a-ens3
     (mapcar 'remove-duplicates
             (delete 'nil
                     (gen-collect 0.9 ps-len :list
                                  '(pick-array
                                    (pick1 nil lisn)
                                    (pick1 nil lisn) 2 b-ens2
                                    (pick1 nil '(:left
                                                 :right
                                                 :up
                                                 :down
                                                 :diagonal-up
                                                 :diagonal-down)))))))
```

Create a series of ensemble combinations from the array. A number of 'picks' take place equal to ps-len (e.g. 30) using the pick-array function. This enables to one to select a row and column – defined by picking a random value from lisn – and a number of items to pick, here always 2. Furthermore a direction is specified so `(pick-array 0.9 1 0 2 2 b-ens2 :right)` would yield `(st wd)`.

```
|#

(setq output-2
      '  ((wd st) (wd br) (st wd) (st br) (wd st) (wd st) (pc wd) (br pc) (pc st) (pc wd)
         (st pc) (wd) (br) (pc) (pc br) (pc st) (st br) (pc wd) (st wd) (pc wd) (st pc)
         (wd st) (pc st) (wd st) (st br) (pc wd) (pc st) (st pc) (st pc) (pc br)))
```

This list contains the output of the above selections from the array.

```
;; timesheet material for phrase 1 and 2

(setq attack-nos '(1 1 1 2 2 2 3 3 4 4 5 6 6 7 8 8)
      ;; source numbers for symbols and note-lengths
      wd-v (instrument-to-string 'wd output-2)
      br-v (instrument-to-string 'br output-2)
      pc-v (instrument-to-string 'pc output-2)
      st-v (instrument-to-string 'st output-2))
```

The various pairs selected from the array are here used to create a 'timesheet', or plan of instrumental activity for the first two phases of the movement. Instrument-to-string will look for certain values in a list and if present output a '-' or if absent a ' ' (blank space).

```
;; creating lists of integers for filling individual beat/space
;; units with  symbols and note-lengths in phrase 1

(setq wd-n
      (e-substitute '(0) '(=)
                    (fill-template
                     (string-to-symbol 'a wd-v) (gen-random 0.1 30 attack-nos)))
      br-n
      (e-substitute '(0) '(=)
                    (fill-template
                     (string-to-symbol 'a br-v) (gen-random 0.1 30 attack-nos)))
      pc-n
      (e-substitute '(0) '(=)
                    (fill-template
                     (string-to-symbol 'a pc-v) (gen-random 0.1 30 attack-nos)))
     st-n
     (e-substitute '(0) '(=)
                    (fill-template
                     (string-to-symbol 'a st-v) (gen-random 0.1 30 attack-nos))))
```

Here the timesheet templates are filled with values from attack-nos. The gen-random function is used to create a list of 30 items from attack-nos, for example: `(3 3 8 2 1 7 1 3 3 4 6 1 1 1 6 2 1 1 1 4 4 7 3 3 1 7 7 8 7 3)`… this is then applied to each template, resulting in something like this for the wind layer: `(3 3 8 = 2 1 7 = = 1 = 3 = = = = = 3 4 6 = 1 = 1 = 1 = = = =)` The rests are then replaced with 0s.

```
;;; timesheet material for phrase 3

(setq attack-nos-1 '(1 1 1 2 2 2 3 3 4 4 4 5 5 5 6 6 7 7 8 8))
(setq wd-v1 (invert-string (instrument-to-string 'wd output-2)))
(setq br-v1 (invert-string (instrument-to-string 'br output-2)))
(setq  pc-v1 (invert-string (instrument-to-string 'pc output-2)))
(setq  st-v1 (invert-string (instrument-to-string 'st output-2)))
```

Once again create a timesheet from the lists of orchestrations, only this time it is inverted by using invert-string, so `“--- - --“` becomes `“    - -    “`.

```
(setq wd-ni
      (e-substitute '(0) '(=)
                    (fill-template
                     (string-to-symbol 'a wd-v1) (gen-random 0.12 30 attack-nos-1)))
      br-ni
      (e-substitute '(0) '(=)
                    (fill-template
                     (string-to-symbol 'a br-v1) (gen-random 0.12 30 attack-nos-1)))
      pc-ni
      (e-substitute '(0) '(=)
                    (fill-template
                     (string-to-symbol 'a pc-v1) (gen-random 0.12 30 attack-nos-1)))
     st-ni
     (e-substitute '(0) '(=)
                    (fill-template
                     (string-to-symbol 'a st-v1) (gen-random 0.12 30 attack-nos-1)))
)


;; the objective here is to gather lists of symbols from a generated super-list (mat)
;; symbol lists now match timesheet strings beat/space layout
;; review the use of seq: and gen-seq:
;; new application here - see also Interactions 5, Flights 3, Albers Studies 4

;; phrase one material

(seq :data mat)


(setq wd-s (do-section :all  '(gen-seq :data (car x)) (mapcar 'list wd-n))

      wd-s1   (e-insert '((=))
                        (e-position 'nil ; finds position of all 'nil occurences
                                    wd-s) (delete 'nil wd-s)))
```

Here the core musical material (mat) is converted into a sequence. A sequence is a list with various unique behaviours. For example, calling `(seq :data)` will return the first value in the list. If this function is called again, the second value will be returned, and so on through the list.

To generate the symbolic (e.g. pitch) data of the wind instrument `gen-seq` is used. This is a way to get a series of values from a sequence, so `(gen-seq :data 3)` would return something like `'(f b –b)`. The number of attacks per zone is used to determine the list sizes. A value of 0 will return 'nil', which are then replaced by rests (=).

```
(seq :data :reset)

(setq br-s (do-section :all  '(gen-seq :data (car x)) (mapcar 'list br-n))
      br-s1   (e-insert '((=))
                        (e-position 'nil br-s ) (delete 'nil  br-s)))

(seq :data :reset)

(setq pc-s (do-section :all  '(gen-seq :data (car x)) (mapcar 'list pc-n))
      pc-s1   (e-insert '((=))
                        (e-position 'nil pc-s) (delete 'nil pc-s)))

(seq :data :reset)

(setq st-s (do-section :all  '(gen-seq :data (car x)) (mapcar 'list st-n))
      st-s1   (e-insert '((=))
                        (e-position 'nil
                                    st-s) (delete 'nil st-s)))

(seq :data :reset)

;;; phrase two material

(setq wd-sx (do-section :all  '(gen-seq :data (car x)) (mapcar 'list pc-n))
      wd-s2 (e-insert '((=))
                        (e-position 'nil
                                    wd-sx) (delete 'nil wd-sx)))

(setq br-sx (do-section :all  '(gen-seq :data (car x)) (mapcar 'list st-n))
      br-s2   (e-insert '((=))
                        (e-position 'nil
                                    br-sx) (delete 'nil br-sx)))

(setq pc-sx (do-section :all  '(gen-seq :data (car x)) (mapcar 'list wd-n))
      pc-s2   (e-insert '((=))
                        (e-position 'nil
                                    pc-sx) (delete 'nil pc-sx)))
```

Reset the sequence so that any calls to it that follow will start from the first element in the list.

Carry out the same procedure for all instruments in the remaining phases. Note that there are no calls to reset the sequence in phase two, while phase three radically mixes up the material. It makes large jumps between areas of the core list by calling `(gen-seq)` after each layer of instrumental material has been generated.

```
(setq st-sx (do-section :all  '(gen-seq :data (car x)) (mapcar 'list br-n))
      st-s2 (e-insert '((=))
                      (e-position 'nil
                                  st-sx)  (delete 'nil  st-sx)))


;;; phrase three material


(setq wd-sy (do-section :all  '(gen-seq :data (car x)) (mapcar 'list wd-ni))
      wd-s3   (e-insert '((=))
                      (e-position 'nil ; finds position of all 'nil occurences
                                  wd-sy) (delete 'nil wd-sy)))

(gen-seq :data 5)

(setq br-sy (do-section :all  '(gen-seq :data (car x)) (mapcar 'list br-ni))
      br-s3   (e-insert '((=))
                      (e-position 'nil br-sy) (delete 'nil  br-sy)))

(gen-seq :data 66)

(setq pc-sy (do-section :all  '(gen-seq :data (car x)) (mapcar 'list pc-ni))
      pc-s3   (e-insert '((=))
                      (e-position 'nil pc-sy) (delete 'nil pc-sy)))

(gen-seq :data 116)

(setq st-sy (do-section :all  '(gen-seq :data (car x)) (mapcar 'list st-ni))
      st-s3   (e-insert '((=))
                      (e-position 'nil
                                  st-sy) (delete 'nil st-sy)))

(gen-seq :data 20)
```

```
;; coda material - timesheet zone-length is doubled

(seq :data (symbol-inversion 'f mat))

(setq wd-sr (do-section :all  '(gen-seq :data (car x))
                        (mapcar 'list (change-length :times 2 (reverse wd-n))))
      wd-s1r   (e-insert '((=))
                      (e-position 'nil ; finds position of all 'nil occurences
                                  wd-sr) (delete 'nil wd-sr)))

(seq :data :reset)

(setq br-sr (do-section :all  '(gen-seq :data (car x))
                        (mapcar 'list (change-length :times 2 (reverse br-n))))
      br-s1r   (e-insert '((=))
                      (e-position 'nil br-sr ) (delete 'nil  br-sr)))

(seq :data :reset)

(setq pc-sr (do-section :all  '(gen-seq :data (car x))
                        (mapcar 'list (change-length :times 2 (reverse pc-n))))
      pc-s1r   (e-insert '((=))
                      (e-position 'nil pc-sr) (delete 'nil pc-sr)))

(seq :data :reset)

(setq st-sr (do-section :all  '(gen-seq :data (car x))
                        (mapcar 'list (change-length :times  2 (reverse st-n))))
      st-s1r   (e-insert '((=))
                      (e-position 'nil
                                  st-sr) (delete 'nil st-sr)))

(seq :data :reset)
```

The creation of coda is similar in some ways to phrase one. However, the symbolic (pitch material) is inverted around symbol ‘f – the first note in the piece. Furthermore the number of attacks are reversed and then doubled, so ‘(0 0 2 4 3) would become ‘(6 8 4 0 0).

A function called Six-r is defined: Six was the original name of this piece, while –r indicates rhythmic processing. This function relies on an external file called "six-r1" appended at the end of this code (p.28). This file contains code for a neural operator that will examine sets of attacks and then return an appropriate rhythmic structure, so `(six–r '(3 3 0))` might return `'((1/8 1/16 1/16) (1/8 1/16 1/16) (-1/4))`. Note that each rhythmic group adds up to 1/4 to ensure synchronisation across the various parts.

```
;; note-length material

(defun six-r (&rest lis)
(apply 'run-neuron 'six-r1 lis))

(include-file "six-r1") ; see below
;; the file six-r1 'must' be present in the same directory as this score-file

(setq wd-l (six-r wd-n)
      br-l (six-r br-n)
      pc-l (six-r pc-n)
      st-l (six-r st-n)
)

(setq wd-lx (six-r pc-n)
      br-lx (six-r st-n)
      pc-lx (six-r wd-n)
      st-lx (six-r br-n)
)

(setq wd-ly (six-r wd-ni)
      br-ly (six-r br-ni)
      pc-ly (six-r pc-ni)
      st-ly (six-r st-ni)
)

(setq wd-lr (six-r (reverse wd-n))
      br-lr (six-r (reverse br-n))
      pc-lr (six-r (reverse pc-n))
      st-lr (six-r (reverse st-n))
)
```

Define the 'zones' (roughly equivalent to bars). Each zone for phases one to three is 1/4 in duration and there is an equal amount in each phase – these are defined as znes. In the coda the lengths of the zones are doubled to 2/4 since the number of attacks has also doubled. Znesr presents a randomised series of zones allowing for contraction and expansion of zones (in the latter case material would repeat) – however this zone structure is not used in the present movement.

```
(setq znes (build-list  '1/4 (length wd-n))
      znesx (build-list  '2/4 (length wd-n)) ;; used in 'final' version d6
      znesr (randomize-zones '1/8 -5 1 (build-list  '1/4 (length wd-n)) 0.1)
)
```

```
#| Timesheet graphics for Phrases 1-3 and Coda

 ; 1
 ;    |   |   |   |   |   |   |   |
wd   "--- ---  - -     --- - - -    "
br   " - -   -    - - -       -    -"
pc   "      -----  --- - -- -  -----"
st   "- ----  - -    -- - ----- --- "

; 2

 ;    |   |   |   |   |   |   |   |
wd   "      -----  --- - -- -  -----"
br   "- ----  - -    -- - ----- --- "
pc   "--- ---  - -     --- - - -    "
st   " - -   -    - - -       -    -"

; 3
;; inversion of 1

;    |   |   |   |   |   |   |   |
wd   "   -   -- - -----   - - - ----"
br   "- - --- ---- - - ------- ---- "
pc   "------     --   - -  - --     "
st   " -    -- - ----  - -     -   -"

; 4
;; reversal of 1 (resolution doubled)

 ;    |   |   |   |   |   |   |   |
wd   "    - - - ---     - -  --- ---"
br   "-    -       - - -    -   - - "
pc   "-----  - -- - ---  -----      "
st   " --- ----- - --    - -  ---- -"

|#
```

```
;; seq: output captured after phrase 2: applied for phrase 3

(length '(f j l j j)) ; 5 - wd

(length '(f j l j j c f f j g g a c b c i a c a i g e e a j b d l f f a c h i k j
 l b h j e c h e g k f j d f e e h h g h b d d h b b e h g d)) ; 66 br

(length '(f j l j j c f f j g g a c b c i a c a i g e e a j b d l f f a c h i k j l
 b h j e c h e g k f j d f e e h h g h b d d h b b e h g d e g i d h k i k h e e c
b e l b h b d i h h j e f k g k l i d e h i k j f g a k f b e g i e c d e j)) ; 116 - pc

(length '(f j l j j c f f j g g a c b c i a c a i)) ; 20 - st


;; score

(def-orchestra 'ensemble-1
 quartet (wd br pc st)
)

(create-tonality six-t (patterns-to-scales (symbol-trim 12 mat)))
; (c 4 d 4 f 4 f# 4 a 4 b 4)
(create-tonality six-t1 (patterns-to-scales (subseq (symbol-inversion 'f mat) 12 23)))
; (c 5 d 5 e 5 f# 5 g 5 g# 5)
```

Once again the Class System is employed to create an ensemble named ‘quartet’, consisting of wind , brass, percussion and strings.

Two new tonalities are created from the core material. In the first instance the first 12 values are taken from the material. Patterns-to-scales will sort these into ascending order, remove duplicates and the map them to a chromatic scale, so `(f j l j j c f f j g g a)` becomes `(a c f g j l)` and then `(c 4 d 4 f 4 f# 4 a 4 b 4)`.

```
(def-section phrase-1
default
 velocity dyn
 tonality (activate-tonality (chromatic e 5))
 zone znes
wd
 symbol wd-s1
 length wd-l
br
 symbol (do-section :all '(symbol-transpose -12 x) br-s1)
 length br-l
pc
 symbol (do-section :all '(octavise x -12) pc-s1)
 length pc-l
st
 symbol (do-section :all '(symbol-transpose -24 x) st-s1)
 length st-l
)

(def-section phrase-2
default
 velocity dyn
 tonality (activate-tonality (chromatic f# 5))
 zone znes
wd
 symbol wd-s2
 length wd-lx
br
 symbol (do-section :all '(symbol-transpose -12 x) br-s2)
 length br-lx
pc
 symbol (do-section :all '(octavise x -12) pc-s2)
 length pc-lx
st
 symbol (do-section :all '(symbol-transpose -24 x) st-s2)
 length st-lx
)
```

Define each phrase as a section. Note that the tonalities change from section to section and the extended zone values are used in the coda. Also the symbolic material for brass and strings has been transposed into a suitable octave. In the percussion layer (played by marimba) the `octavise` function is used – this means that most notes will be the same, but repeated notes will shift down an octave. So `(d e j a l b b b)` becomes `(d e j a l b -1 -1)`.

```
(def-section phrase-3
default
 velocity dyn
 tonality (activate-tonality (six-t c 5)) ; ((c 5 d 5 f 5 f# 5 a 5 b 5))
 zone znes
wd
 symbol wd-s3
 length wd-ly
br
 symbol (do-section :all '(symbol-transpose -6 x) br-s3)
 length br-ly
pc
 symbol (do-section :all '(octavise x 6) pc-s3)
 length pc-ly
st
 symbol (do-section :all '(symbol-transpose -12 x) st-s3)
 length st-ly
)

(def-section coda
default
 velocity dyn
 tonality (activate-tonality (six-t1 c# 5)) ; ((c# 5 d# 5 f 5 g 5 g# 5 a 5))
 zone znesx
wd
 symbol wd-s1r
 length wd-lr
br
 symbol (do-section :all '(symbol-transpose -12 x) br-s1r)
 length br-lr
pc
 symbol (do-section :all '(octavise x -6) pc-s1r)
 length pc-lr
st
 symbol (do-section :all '(symbol-transpose -6 x) st-s1r)
 length st-lr
)
```

```
(def-channel
wd 1
br 2
pc 3
st 4
)

(def-program gm-sound-set
wd soprano-sax
br trombone
pc marimba
st cello
)

(def-controller gm-controllers
  (wd  panning  '((20)))
  (br   panning  '((50)))
  (pc   panning  '((80)))
  (st   panning  '((120)))
)

(def-tempo 50)


(play-file-p "six-initial-d6"
quartet '(phrase-1 phrase-2 phrase-3 coda)
)


;; version d3 - znes
;; version d4 - znesx
;; version d5 - znesr
;; version d6 - alterations to coda tonality (reflects symbol-inversion)
```

```
#|


;; score-file "six-r1" - rhythmic 'table' for use in SIX - Quatour des Timbres

(def-neuron six-r1
 (in 1 '0) '((-1/4))
 (in 1 '1) (list (pick-random '((1/4)(-1/8 1/8) (-1/16 1/16 -2/16) (-1/16 -1/16 1/16 -1/16))))
 (in 1 '2) (list (pick-random '((1/8 1/8)(-1/8 1/16 1/16) (1/16 1/16 -1/8) (1/16 -1/8 1/16))))
 (in 1 '3) (list (pick-random '((1/8 1/16 1/16)(-1/16 1/16 1/16 1/16)
                                 (-1/32 1/32 1/16 1/8))))
 (in 1 '4) (list (pick-random '((1/16)(-1/32 1/32 1/32 1/32 -1/32 1/32 -1/16))))
 (in 1 '5) (list (pick-random '((1/8-5 1/8-5 1/8-5 1/8-5 1/8-5 -1/8)
                                 (1/16. 1/32 1/32 1/32 1/32 -1/32)
                                 (-1/32 1/32 1/32 1/32 -1/32 1/32 1/32 -1/32)
                                 (1/4-5 1/4-5 1/4-5 1/4-5 1/4-5))))
 (in 1 '6) (list (pick-random '((1/8-6 1/8-6 1/8-6 1/8-6 1/8-6 1/8-6-1/8)
                                 (1/32 -1/32 1/32 1/32 -1/32 1/32 1/32 1/32))))
 (in 1 '7) (list (pick-random '((1/32 1/32 1/32 1/32 -1/32 1/32 1/32 1/32)
                                (1/4-7 1/4-7 1/4-7 1/4-7 1/4-7 1/4-7 1/4-7))))
 (in 1 '8) (list (build-list '1/32 8))
 (in 1 '9) (list (build-list '1/4-9 9))
 (in 1 '11) (list (build-list '1/4-11 11))
 (in 1 '14) (list (build-list '1/4-7 7))
)


 |#
```

```
;; Quatour des Timbres movement 3: Jeux Diurnes

;; functions

(defun pattern-to-scale (l)
  (car (sort-tonality
      (list (c-pitch-to-tonality (c-symbol-to-pitch (find-unique l)))))))


(defun patterns-to-scales (symbol-lists)
 (cond ((is-flat symbol-lists)
        (pattern-to-scale symbol-lists))
        (t (mapcar 'pattern-to-scale symbol-lists))))

#|

(setq a-ens '(wd br pc st))

(setq a-ens1 (power-set '(wd br pc st))
      ps-len (* (length a-ens1) 2)
)


(setq a-ens2
(build-array
 ; column 0    1     2    3
       '((wd   br    pc   st )   ; 0
         (pc   br    st   wd )   ; 1
         (st   wd    pc   br )   ; 2
         (br   st    wd   pc ))) ; 3
)

(setq lisn '(0 1 2 3) ; array list
      lisx '(1 2 3 4) ; number of items
)
```

This section is enclosed in comment tags, meaning that the SCOM compiler will ignore it. However, it is included here since it gives an indication of the pre-compositional thinking and experiment for the movement, the results of which are used in the code that follows on p.31.

As in the preceding movement, the ensemble is defined as four different layers (wd br pc st) and the powerset (all combinations of one or more of these elements) is created. Ps-len is the length of the powerset doubled (30). An array is again created to inform the orchestration of the piece.

```
(setq a-ens3
     (mapcar 'remove-duplicates
              (delete 'nil
                      (gen-collect 0.9 ps-len :list
                                   '(pick-array
                                     (pick1 nil lisn)
                                     (pick1 nil lisn) (pick1 nil lisx) a-ens2
                                     (pick1 nil '(:left
                                                  :right
                                                  :up
                                                  :down
                                                  :diagonal-up
                                                  :diagonal-down)))))))
```

Again elements are picked from the array to build up an orchestration scheme, then abstracted into a beat/space notation for each of the instrumental layers.

```
(setq output-1
      '((wd br pc st) (br) (br pc) (wd st pc) (st) (pc br) (pc wd)
(st pc br wd) (wd st) (pc br wd) (br) (br) (st pc br wd) (br pc st)
(st br) (st br pc) (pc) (st) (st) (br st) (st wd br) (st br) (br pc wd st)
(pc) (pc br st wd) (pc wd) (pc) (pc wd st) (pc) (br pc)))


;; beat/space score of the above output written 'by hand'

 ;     |   |   |   |   |   |   |   |
wd   "-  -  ----  -       - - -- -  "
br   "---  - - -------   ---- -    -"
pc   "- -- --- -  -- --     --------"
st   "-  --  --   ---- ------ -  -  "

|#
```

```
;; score

(def-orchestra 'ensemble-1
 quartet (wd br pc st)
)

(setq mat (vector-to-symbol a l (gen-noise-white 128 1.0 0.5))
      dyn (vector-round 35 75 (gen-noise-white 256 1.0 0.5))
)

(create-tonality six-t (patterns-to-scales (symbol-trim 12 mat)))
; (c 4 d 4 f 4 f# 4 a 4 b 4)
(create-tonality six-t1  (patterns-to-scales (subseq mat 12 23)))
; (c 4 c# 4 d 4 e 4 f# 4 g# 4)

(def-section-timesheet phrase
;
with 2/4
;
tonality "." (activate-tonality (chromatic e 5))
;    |   |   |   |   |   |   |   |
wd   "-  -  ----  -       - - -- -  "
br   "---  - - -------   ---- -    -"
pc   "- -- --- -  -- --     --------"
st   "-  --  --   ---- ------ -  -  "
;
beat 1/16
;
wd "-" (find-anacrusis mat) with dyn
br "- - - -  -------" (symbol-transpose -12 mat) with dyn
pc " --------- -- --" (octavise mat -12) with dyn
st "-    ---- ---- -  - -" (symbol-transpose -24 mat) with dyn
)
```

Use white noise to generate symbolic material (128 samples between a-l) and dynamic material (256 samples between 35-75).

Use the first 12 values of the symbolic material to construct a tonality called six-t. The next 12 values are used to construct the tonality six-t1.

The opening section of this movement is composed using def-section-timesheet. This allows various timesheets to be defined, which control the various aspects of a piece.

In the first section a general scheme of instrumental activity is defined. Where a – is present the instrument plays. A resolution of 2/4 is used and a single tonality (chromatic e 5).

The beat resolution is then defined as 1/16 and the actual material being played is defined. This means that, in the case of the wind for every 1/16 note in a 2/4 portion of activity the instrument will play a constant 1/16th using `(find-anacrusis mat)` for its symbolic data and dyn for velocities. The brass has a series of rhythms adding up to 2/4 in which there are rests in the first half and constant activity in the second.

```
;; extracts zone-lengths from timesheet

(setq z-wd (change-length :times 2 (same-as zone of wd in phrase) :ratio)
      z-br (change-length :times 2 (same-as zone of br in phrase) :ratio)
      z-pc (change-length :times 2 (same-as zone of pc in phrase) :ratio)
      z-st (change-length :times 2 (same-as zone of st in phrase) :ratio)
)
```

Extract zone-lengths from phrase. Here they are doubled so that 2/4 becomes 1/1. Zones in which nothing is played will be returned as –1/1. So for woodwind we get `'(1/1 –1/1 –1/1 1/1 –1/1 –1/1 1/1 1/1 1/1 …)`

```
(setq z-wd2 (length-inversion z-wd)
      z-br2 (length-inversion z-br)
      z-pc2 (length-inversion z-pc)
      z-st2 (length-inversion z-st)
)
```

The subsequent sections of the movement use different manipulations of the zone-lengths. Here length-inversion is used, which is essentially the same as inverting the time-sheet, so that the wind layer now looks like `(–1/1 1/1 1/1 –1/1 1/1 1/1 –1/1 –1/1 –1/1 …_`

```
(setq wd-zn (randomize-zones '1/8 0 4 z-wd 0.1)
      br-zn (randomize-zones '1/8 0 4 z-br 0.1)
      pc-zn (randomize-zones '1/8 0 4 z-pc 0.1)
      st-zn (randomize-zones '1/8 0 4 z-st 0.1)
)
```

Randomize-zones shortens zone-lengths by a certain duration – here each 1/1 zone has between 0 and 4 1/8 notes subtracted from its total length. So the woodwind layer now has the following zone values: `(3/4 –5/8 –1/2 5/8 –5/8 –7/8 3/4 3/4 5/8 …)`

```
(setq wd-zn2 (randomize-zones '1/8 -1 5 z-wd2 0.2)
      br-zn2 (randomize-zones '1/8 -1 5 z-br2 0.2)
      pc-zn2 (randomize-zones '1/8 -1 5 z-pc2 0.2)
      st-zn2 (randomize-zones '1/8 -1 5 z-st2 0.2)
)
```

Here randomize-zones is applied to the inverted zone-lengths. Note also that the range is –1 to 5. The –1 means that zones may now also be extended by 1/8 in duration. The wind layer now looks like this: `(-3/8 7/8 3/4 –7/8 7/8 5/8 –3/8 –3/8 –7/8 9/8 …)`

```
(setq wd-zn3 (randomize-zones '1/8 1 6 z-wd 0.3)
      br-zn3 (randomize-zones '1/8 1 6 z-br 0.3)
      pc-zn3 (randomize-zones '1/8 1 6 z-pc 0.3)
      st-zn3 (randomize-zones '1/8 1 6 z-st 0.3)
)
```

The final section in this movement again uses randomize-zones with a high and low value of 1 and 6. Zone lengths will therefore all fall between 7/8 (e.g. 1/1 minus 1/8) and 1/4 (e.g. 1/1 minus 6/8).

```
(setq wd-rh (gen-repeat 8 (beat/space (get-ratio '1/16 :ratio)  "------ - -------"))
      br-rh (gen-repeat 8  (beat/space (get-ratio '1/16 :ratio) "- - - -  -------"))
      pc-rh (gen-repeat 8  (beat/space (get-ratio '1/16 :ratio) " --------- -- --"))
      st-rh (gen-repeat 8  (beat/space (get-ratio '1/16 :ratio) "-    ---- ---- -"))
)
```

Here the beat/space command is used to translate a series of beat/space notations into rhythms. These are then repeated 8 times and make up the basic rhythmic data for all but the first section of the movement. So the basic rhythm for the wind layer is `(1/16 1/16 1/16 1/16 1/16 1/16 -1/16 1/16 -1/16 …)`

```
(def-section phrase-1 ; expansion / compression of zone-lengths
default
 velocity dyn
 tonality (activate-tonality (chromatic f# 5))
 length '(1/16)
wd
 zone wd-zn
 symbol (symbol-swallow wd-rh (find-anacrusis mat))
br
 zone br-zn
 symbol (symbol-swallow br-rh (symbol-transpose -12 mat))
pc
 zone pc-zn
 symbol (symbol-swallow pc-rh (octavise mat -12))
st
 zone st-zn
 symbol (symbol-swallow st-rh (symbol-transpose -24 mat))
)
```

As in the previous movement the various sections are defined with def-section, providing zone and symbolic data for each instrumental layer.

```
(def-section phrase-2 ; timesheet positions inverted
default
 velocity dyn
 tonality (activate-tonality (six-t c 5)) ; ((c 5 d 5 f 5 f# 5 a 5 b 5))
 length '(1/16)
wd
 zone wd-zn2
 tonality (activate-tonality (six-t c 4))
 symbol (symbol-swallow wd-rh (find-anacrusis (symbol-transpose 6 mat)))
br
 zone br-zn2
 symbol (symbol-swallow br-rh (symbol-transpose -6 mat))
pc
 zone pc-zn2
 symbol (symbol-swallow pc-rh (octavise mat -6))
st
 zone st-zn2
 symbol (symbol-swallow st-rh (symbol-transpose -12 mat))
)

(def-section phrase-3 ; timesheet reversed
default
 velocity (change-length :sub 5 (vector-to-list (reverse dyn)))
 tonality (activate-tonality (six-t1 c# 5)) ; ((c# 5 d 5 d# 5 f 5 g 5 a 5))
 length '(1/16)
wd
 zone (reverse wd-zn3)
 symbol (symbol-swallow wd-rh (find-anacrusis mat))
br
 zone (reverse br-zn3)
 symbol (symbol-swallow br-rh (symbol-transpose -12 mat))
pc
 zone (reverse pc-zn3)
 symbol (symbol-swallow pc-rh (octavise mat 6))
st
 zone (reverse st-zn3)
 symbol (symbol-swallow st-rh (symbol-transpose -6 mat))
)

(def-channel
wd 1
br 2
pc 3
st 4
)

(def-program gm-sound-set
wd soprano-sax
br trombone
pc marimba
st cello
)

(def-controller gm-controllers
  (wd  panning  '((20)))
  (br   panning  '((50)))
  (pc   panning  '((80)))
  (st   panning  '((120)))
)

(def-tempo 70)


(play-file-p "six-initial-b"
quartet '(phrase phrase-1 phrase-2 phrase-3)
)
```

```
;; Quatour des Timbres movement 4: In finem Psalmus David
;; (Psalm 19 vv.9-16)

;; functions

(defun pattern-to-scale (lis)
(car (sort-tonality (list (c-pitch-to-tonality
(c-symbol-to-pitch (find-unique lis)))))))


(defun patterns-to-scales (symbol-lists)
 (cond ((is-flat symbol-lists)
        (pattern-to-scale symbol-lists))
        (t (mapcar 'pattern-to-scale symbol-lists))))


(defun len-to-sym (len l-range h-range val)
"generates a list of symbols over a given range from a list of lengths"
(pick-random-n  (length len) (g-symbol l-range h-range ) :seed val))


(defun len-to-sym-r (len l-range h-range val)
"as for len-to-sym but aligns pause symbols to rest lengths"
(align-to-length len
(pick-random-n  (length len) (g-s len-to-symbol l-range h-range ) :seed val)))


(defun len-to-sym-sw (len l-range h-range val)
"generates a list of symbols over a len-to-sym given range from a list of lengths
- but swallows symbols when rests appear in list of lengths"
 (cond ((is-flat len)
        (symbol-swallow len (len-to-sym len l-range h-range  val)))
       (t (mapcar (function (lambda (x y) (symbol-swallow x y))) len
(mapcar (function (lambda (x y) (len-to-sym x l-range h-range  y))) len val)))))
```

For a given list of length values generate symbols ranging between l-range and h-range. For example `(len-to-sym '(1/8 -1/8 1/8) 'a 'c 0.1)` might result in `(b b a)`.

As above save that where rests occur (e.g. negative length values) pause symbols are aligned. The above become `(b = b)`.

In the above case the rest shifted the output `(b b a)` became `(b = b)`. This version of the function will swallow a symbol that occurs at a rest, so our example results in `(b = a)`.

```
#|

(len-to-sym (qlength '8-111111100000) '-c 'i 0.1)
; (d h i f e -c b a g c -b e)

(len-to-sym-r (qlength '8-110110100000) '-c 'i 0.1)
; (d h = i f = e = = = = =)

(len-to-sym-sw (qlength '8-110110100000) '-c 'i 0.1)
; (d h = f e = b = = = = =)

(setq len1 (list (qlength '8-110110100000)(qlength '8-110110101111)))
(setq seeds  '(0.1 0.2))

(len-to-sym-sw len1 '-c 'i seeds)
; ((d h = f e = b = = = = =) (i b = a c = h = -c -b e -c))

|#
```

Here are some examples of the new functions in action. Note the use of `qlength`. This can be used to generate a series of notes and rests. `(qlength '8-1010)` would give `'(1/8 -1/8 1/8 -1/8)`. Multiples can also be used, so `(qlength '8-1230)` would give `'(1/8 1/4 3/8 -1/8)`.

```
;; from section 1 (six initial b)

(setq mat (vector-to-symbol a l (gen-noise-white 128 1.0 0.5))
      dynx (vector-round 35 75 (gen-noise-white 256 1.0 0.5))
)
```

As in the previous movement white noise is used to define the symbolic and dynamic schemes.

```
#|

(setq mat-1 '((f j l j j c f f j g g a) (c b c i a c a i g e e a)
(j b d l f f a c h i k j) (l b h j e c h e g k f j) (d f e e h h g h b d d h)
(b b e h g d e g i d h k) (i k h e e c b e l b h b)
(d i h h j e f k g k l i) (d e h i k j f g a k f b) (e g i e c d e j a l b b)
(b j k f g b a k)))

(patterns-to-scales mat-1)
```

```
> ((c 4 d 4 f 4 f# 4 a 4 b 4) (c 4 c# 4 d 4 e 4 f# 4 g# 4)
 (c 4 c# 4 d 4 d# 4 f 4 g 4 g# 4 a 4 a# 4 b 4) (c# 4 d 4 e 4 f 4 f# 4 g 4 a 4 a# 4 b 4)
(c# 4 d# 4 e 4 f 4 f# 4 g 4) (c# 4 d# 4 e 4 f# 4 g 4 g# 4 a# 4)
(c# 4 d 4 e 4 g 4 g# 4 a# 4 b 4) (d# 4 e 4 f 4 f# 4 g 4 g# 4 a 4 a# 4 b 4)
(c 4 c# 4 d# 4 e 4 f 4 f# 4 g 4 g# 4 a 4 a# 4) (c 4 c# 4 d 4 d# 4 e 4 f# 4 g# 4 a 4 b 4)
(c 4 c# 4 f 4 f# 4 a 4 a# 4))

|#

;; material

(init-rnd 0.7)

(setq as 0.11
      bs 0.12
      cs 0.13
      ds 0.146
      es 0.159
      fs 0.16
      gs 0.17
      hs 0.184
      is 0.19
      js 0.20
      ks 0.21
      ls 0.22
      ms 0.23
      ns 0.249
      os 0.25
      ps 0.268
      qs 0.27
      rs 0.2851
      ss 0.29
      xs nil)
```

This movement assembles 20 sections a-x. Here different random seeds are defined for each section to control picking from the above defined symbolic data, mat.

```
(setq ax  (qlength '8-00200112) ; lex domini
      ay  ax
      aj  ax
      ak  ax
      bx  (qlength '(8-00000000 4-111 6-222)); immaculata convertens animas
      by  (qlength '(8-000000 4-0111 6-222))
      bj  (qlength '(8-022111 4-111 6-222))
      bk  (qlength '(8-011111 4-0111 6-222))
      cx  (qlength '8-011111221111) ; testimonium domini fidele
      cy  cx
      cj  cx
      ck  cx
      dx  (qlength '(8-0012211 4-11 16-112)) ; sapientiam praestans parvulis
      dy  (qlength '(8-0031111 4-11 16-112))
      dj  dx
      dk  dy
      ex (qlength '8-0002111112022) ; justitiae domini rectae
      ey ex
      ej ex
      ek ex
      fx  (qlength '8-0011111111322) ; laetificates corda
      fy  (qlength '8-0000111111322)
      fj  (qlength '8-0000001111322)
      fk  (qlength '8-000000001111122)
      gx  (qlength '8-0111111111)  ; praeceptum domini lucidum
      gy  gx
      gj  gx
      gk  gx
      hx  (qlength '(4-01 6-222 8-011200)) ; illuminans oculos
      hy hx
      hj hx
      hk hx
      ix (qlength '(8-1100 16-112220022221120011200))
       ; timor domini sanctus premamens in  saeculum saeculi
      iy ix
      ij ix
      ik ix
```

Here the rhythmic plans for each section are defined. These are abstractions of a psalm text. Each phrase has four variants x,y,j,k which will be mapped to the different instrumental layers. Note the use of multiple values to stress certain consonants – e.g. Lex do-mi-ni = 1/4 1/8 1/8 1/4. Note also the changes of rhythmic denomination – for example in the second phrase from 1/8ths to 1/4ths to 1/3rds!

```
      jx (qlength '(8-02111 16-11222 8-00022111011111))
       ; judicia domini vera justificata in semet ipsa
      jy jx
      jj jx
      jk jx
;; tonality changes here!
      kx (qlength '8-000144000000000002111110011004); delicta quis intellegit
                                                   ; ab occultis meis munda me
      ky (qlength '8-000144011112102111110011004)
      kj  ky
      kk  kx
      lx  (qlength '8-01211110000001111) ; et ab alienis parce servo tuo

      ly  (qlength '8-01211110000000000)
      lj  (qlength '8-01211110011001111)
      lk  (qlength '8-00000000011001111)
      mx  (qlength '8-0111111111100201111111) ; si mei non fuerint dominati
                                                     ; tunc immaculatus ero
      my  (qlength '8-0000000000000201111111)
      mj  my
      mk  mx
      nx  (qlength  '8-00000000011220212 00) ; et emundabor a delicto maximo
      ny  (qlength  '8-000111110112202120 0)
      nj  ny
      nk  nx
      ox (qlength '8-011101111121100110000)
       ; et erunt ut complaceant eloquia oris mei
      oy (qlength '8-011101111121100000011)
      oj (qlength '8-011101111121100110000)
      ok (qlength '8-011101111121100000011)
      px (qlength '8-0111111122202222110000 00)
       ; et meditatio cordis mei in conspectu tuo semper
      py (qlength '8-01111111222022220001100)
      pj (qlength '8-01111111222022221100000 0)
      pk (qlength '8-01111111222022220001100)
      qx (qlength '(8-000 4-11122    8-02211   4-00))
       ; domine adjutor meus et redemptor meus
      qy (qlength '(8-111 4-0000000  8-0000000 4-11))
      qj (qlength '(8-000 4-11122    8-02211   4-00))
      qk (qlength '(8-111 4-0000000  8-0000000  4-11))
      pz (qlength '8-00000000)
)

(setq lenW  (list ax bx cx dx ex fx gx hx ix jx)
      lenB  (list ay by cy dy ey fy gy hy iy jy)
      lenP  (list aj bj cj dj ej fj gj hj ij jj)
      lenC  (list ak bk ck dk ek fk gk hk ik jk)
      strL '(     a  b  b  c  d  e  a  f  g  d )
)

(setq lenW1 (list kx lx mx nx ox px qx)
      lenB1 (list ky ly my ny oy py qy)
      lenP1 (list kj lj mj nj oj pj qj)
      lenC1 (list kk lk mk nk ok pk qk)
      strL1 '(    l  k  m  j  n  r  p )
)
```

Gather all the phrases together to create rhythmic plans for each instrumental layer. Note that there are two collections whose groupings indicate where a change of tonality will occur. The final list indicates the random seed to be applied when the `len-to-sym` functions are applied to the rhythmic plans.

```
(setq seeds (eval-section strL 's 'list)
      seeds1 (eval-section strL1 's 'list))
```

Use eval-section to draw together a list of random seed numbers from the previously defined lists (as-xs).

```
(setq unisons '(x = x = x = x = x x))

(setq W-sym  (do-section unisons
                         '(symbol-scale '(a g) x) (len-to-sym-sw lenW 'a 'j seeds))
      B-sym  (do-section unisons
                         '(symbol-scale '(a g) x) (len-to-sym-sw lenB '-c 'h seeds))
      P-sym  (do-section unisons
                         '(symbol-scale '(a g) x) (len-to-sym-sw lenP '-e 'f seeds))
      C-sym (do-section unisons
                            '(symbol-scale '(g m) x) (len-to-sym-sw lenC  'c 'l seeds))
)
```

Create symbolic data corresponding to the rhythmic plan for the first section. In the sections where the rhythmic plan is the same over each instrumental layer (for example a and c above) perform a symbol-scale function to bring each part within the compass of a chromatic fifth (a to g).

```
;; symbol-scale cannot process lists of '(= = = =)

(setq pentad '(= x = = x = =))

(setq W-sym1 (do-section pentad
                         '(symbol-scale '(c g) x) (len-to-sym-sw lenW1 'a 'j seeds1))
      B-sym1 (do-section pentad
                         '(symbol-scale '(c g) x) (len-to-sym-sw lenB1 '-c 'h seeds1))
      P-sym1 (do-section pentad
                         '(symbol-scale '(-c f) x) (len-to-sym-sw lenP1 '-e 'f seeds1))
      C-sym1 (do-section pentad
                         '(symbol-scale '(e l) x) (len-to-sym-sw lenC1  'c 'l seeds1))
)
```

Perform a similar operation to create symbolic data for the second section. This time, selected passages are constrained to a pentatonic compass using symbol-scale (c g).

```
(setq dyn (mapcar (function (lambda (x)
                              (vector-round 70 55
                                        (gen-sin 0.5 0.5 x 240 (gen-sin 8 0.9 x)))))
                  (mapcar 'length (append W-sym W-sym1))))
```

Create a dynamic scheme by first counting the number of items in each list of symbols in W-sym and W-sym1. For each symbol a value from a modulated sine wave is rounded to between 70 and 55.

```
(create-tonality six-t (patterns-to-scales (symbol-trim 12 mat)))
; (c 4 d 4 f 4 f# 4 a 4 b 4)

(create-tonality six-t1 (patterns-to-scales (subseq mat 12 23)))
; (c 5 d 5 e 5 f# 5 g 5 g# 5)

(create-tonality six-t1i (patterns-to-scales (subseq (symbol-inversion 'f mat) 12 23)))
; (d 4 e 4 f# 4 g# 4 a 4 a# 4)
```

This movement builds on the tonal schemes developed in the previous movement, with the addition of a scheme based on an inversion of the source symbolic material.

```
(setq tonalW (append (gen-repeat (length lenW)  (activate-tonality (six-t1 c 5)))
                     (gen-repeat (length lenW1)
                                 (activate-tonality (six-t1i c# 5)))))
```

Create a tonality scheme based on tonalities six-t1 c 5 and six-t1i c# 5. Each tonality is repeated for as long as the relevant section lasts.

```
;; score

(def-tonality
wd tonalW
br (transpose-chords tonalW -12)
pc tonalW
st (transpose-chords tonalW -24)
)
```

The different parametric elements of this movement are not assembled in sections using the Class System as in the earlier movements. Symbol, length and velocity values are assembled into sections then synchronized by the control of zone-length. Here the tonality of the piece is set, applying tonalW to each instrumental layer. Note also transpositions for the brass and strings.

```
(def-symbol
wd (append W-sym W-sym1)
br (append B-sym B-sym1)
pc (append P-sym P-sym1)
st (append C-sym C-sym1)
)
```

Bring together all the symbolic data for each section.

```
(def-length
wd (mapcar 'l-rest-revert (append lenW lenW1))
br (mapcar 'l-rest-revert (append lenB lenB1))
pc (mapcar 'l-rest-revert (append lenP lenp1))
st (mapcar 'l-rest-revert (append lenC lenC1))
)
```

These expressions bring together all the rhythmic data for each section. Note the use of `l-rest-revert` to change all negative rhythmic values into positive ones. Negative values are not required, already having corresponding rests in the symbolic data.

```
(def-zone
default (mapcar 'z-ratio-sc (append lenW lenW1))
)
```

Define the zones of the piece, which are roughly analogous to bars. `z-ratio-sc` adds up all the durational values in a section and produces a zone length, so `(-1/8 -1/8 1/4 -1/8 -1/8 1/8 1/8 1/4)` becomes `(5/4)`.

```
(def-velocity
wd dyn
br dyn
pc dyn
st dyn
)
```

```
(def-channel
wd 1
br 2
pc 3
st 4
)

(def-program gm-sound-set
wd soprano-sax
br trombone
pc vibraphone
st cello
)

(def-controller gm-controllers
  (wd  panning  '((20)))
  (br  panning  '((50)))
  (pc  panning  '((80)))
  (st  panning  '((120)))
)

(def-tempo 90)

(compile-instrument-p "ccl;output:" "six-psalm19iv"
wd
br
pc
st
)
```