String Trio (2012)

for violin, viola and violoncello

Nigel Morgan

Symbolic Composer score files
Annotated by Phil Legard

String Trio (2012)

for violin, viola and violoncello

Nigel Morgan

About the music

My objective in composing for the string trio was to explore the potential of the trio as a medium for my own musical ideas and to apply two techniques I had recently developed in composing with algorithmic means

In my work for solo piano *The White Light of Wonder* I developed a way of creating sequences of 4-part chords through algorithmic means. The resultant harmony was often surprising and unusual, chords being separate and independent objects rather than formed from a distinct tonality into any kind of hierarchy. Much later, when writing *To the Dark Unseen* for a string ensemble of 10 players I experimented with a method of organising when and where chords could sound using a beat / space technique in which the composition of the chords on every beat changed. In this short String Trio I have brought the two devices together.

There are three distinct sections in this Trio played without a break. Both the first and the third have extensive links of ornamentation between chords. The second is a pulsating movement of iterations of chords. The harmonic sequence is identical in one and two, but in three the harmony is generated entirely from the starting chord of movements 1 and 2 to produce just eight chords that are then arranged in a series found in Franco Donatoni's String *Quartet The Heart's Eye* – 1 2 8 1 2 3 7 8 1 2 3 6 7 8 1 2 3 4 5 6 7 8 1 2 3 4 5 then subjected to a palindrome.

```
;; String Trio (2012)
; 1/3
#|
vn   - violin
va   - viola
vc   - cello
|#

;; Functions necessary for this piece

(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)))))))
```

Instrument-to-string produces a beat/space timesheet (e.g. "---- - -") based on a series of orchestration descriptors (e.g. `'((fl pno)(pno)(fl)(fl pno))`). So, `(instrument-to-string 'va '((va vn) (va vc) (vn vc)))` would yield `"-- "`.

```
(defun string-to-length (length string)
(let (out)
(dovector (x string)
 (if (equal x #\Space)
 (push (* (abs (get-ratio length)) -1) out)
 (push length out)))
(nreverse out)))
```

String-to-length will process a timesheet into tokens indicating action or rest (positive or negative) values. So, `(string-to-length '1 '"-- - --")` yields `(1 1 -1 1 -1 1 1)`. This function is not used in the piece per se, but is called as part of **zone-convertor**.

Zone-converter extends the functionality of **string-to-length** by processing a timesheet against a list of zone lengths. These are then expressed as either rests or areas of possible musical activity. So, `(zone-convertor '(1/8 2/4 3/8) "- -")` yields `(1/8 -1/2 3/8)`.

```
 (defun zone-convertor (list-of-zones timesheet-string)
"maps beat/space values onto a list of zones"
(do-quietly
 (get-ratio-sc (mapcar (function (lambda (x y) (* (get-ratio-cl x) y)))
        (mapcar 'get-ratio list-of-zones)
         (string-to-length '1 timesheet-string)))))
```

```
(defun p-replace-sections (lis sym-source sym-target)
(p-replace nil (calculate-positions lis)
          (p-select (calculate-positions lis) sym-source) sym-target))
```

P-replace-sections can be used to replace items in one list with those in a second list according to a template. So: `(p-replace-sections '(= x x =) '((a b)(c d)(e f)(g h)) '((1 2)(3 4)(5 6)(7 8)))` yields `((1 2)(c d)(e f)(7 8)`.

```
(defun calculate-rests (lis-of-len)
   (prog (out l-element)
    loop
    (setq l-element (car lis-of-len))
    (cond ((null lis-of-len) (return out)))
    (setq out (append out
                      (cond ((is-minus-length-symbol l-element)
                             (list 'x))
                            (t (list '=)))))
    (setq lis-of-len (cdr lis-of-len))
    (go loop)))
```

Calculate-rests is used to construct a template from a series of note lengths. So, `(calculate-rests '(1/8 -1/8 1/16 -1/16))` yields `(= x = x)`.

```
(defun calculate-positions (lis)
 (cond ((symbolp (car lis))
       (e-position 'x lis))
      (t (e-position 'x
(string-to-symbol 'x (integer-to-string-1 lis))))))
```

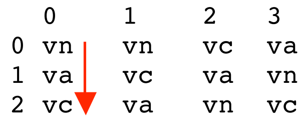
Used internally by **p-replace-sections.** This function will examine a template and return the indices of the `x` values in the template. So, `(calculate-positions '(x = = x))` yields `(0 3)`.

```
;; Code for the first part of the string trio

(setq moments
(build-array
 ; column 0    1     2   3
      '(( vn   vn   vc   va)
        ( va   vc   va   vn)
        ( vc   va   vn   vc)
))
```

This array is used to define instrumental activity in each zone (or bar) of the piece. The table can be read up, down, left or right.

```
(setq lisn '(0 1 2))

(setq output-all
     (mapcar 'remove-duplicates
             (delete 'nil
                 (gen-collect 0.22 120 :list
                 '(pick-array
                        (pick1 nil lisn)
                        (pick1 nil lisn) (get-random 0 4) moments
                        (pick1 nil '(:left
                                     :right
                                     :up
                                     :down
                                     )))))))
```

Here, a series of orchestrations are chosen from the array above. This code is executed 120 times by **gen-collect** with a random seed of 0.22. A row and column is picked from the array using a value from **lis** (0, 1 or 2). A number of items between (0 and 4) are picked from the selected cell in the array, moving either left, right, up or down from that cell. So, `(pick-array 0 0 4 moments :down)` would start from the top left corner of the **moments** array (0,0). Starting with the value at 0,0, we would pick four values, moving down one cell at a time. Since the array only has three rows, our picker wraps back around to the start, yielding `(vn va vc vn)`.



	0	1	2	3
0	vn	vn	vc	va
1	va	vc	va	vn
2	vc	va	vn	vc

Finally, any null (zero length) picks are removed, as are duplicates in each pick, so `(vn va vc vn)` becomes `(vn va vc)`.

```
(setq len-n (length output-all))
```

Total length of sections stored as **len-n** (107).

```
; output of output-all

#|
((va vc vn) (va vn) (va) (vn va vc) (va vc) (vn vc) (va vc vn)
(vn vc) (vc vn) (vc va vn) (va vc vn) (vn va vc)
(va vc) (vc vn va) (vn va) (vn) (vn vc) (va vn vc) (vc vn va)
(vc) (va vc) (vc vn) (vc vn) (vn va) (vn va) (va)
(vc va) (va vn vc) (va vc vn) (vc vn) (va vc vn) (vc vn va) (vn
vc) (vc va vn) (vc va) (vc va vn) (vn) (vn vc)
(vn) (va) (va vc) (vc va) (vn) (vc) (va vn) (va) (va vn) (vn va
vc) (va) (vn va) (vc) (va) (va vc vn) (vc vn)
(va vn) (vn va) (vc) (va vc) (vc va vn) (va vn) (vn va vc) (va
vc vn) (vn) (vn) (vc vn) (vn) (va) (vc va vn)
(vc) (va) (vc) (va vc vn) (vn vc va) (va vc) (va vc vn) (va vn)
(vn va vc) (va vn vc) (vc va vn) (vn vc) (va vn)
(vn vc va) (vc vn va) (vn va vc) (vc va vn) (vc va) (va) (vc va
vn) (va) (vc va vn) (vc va vn) (vn va vc) (vn)
(vc va) (vc) (vc) (vn) (vc) (vn) (vc va) (vc va vn) (vc va) (va)
(vn) (vc) (vn vc) (vn))
|#
```

A dump of all the 107 choices of instrumentation, for reference.

```
;; instrumentation - activity and silence

(instrument-to-string 'vn output-all)

(instrument-to-string 'va output-all)

(instrument-to-string 'vc output-all)

#| ;; one minute of timesheet for trio

vn  "-- - ------- ------  ----  ------- ----   - - -- -  ---- -------- -   -- -----------  - ----   - - -  - --"
va  "----- -  ------  -- -  ------ -- ---   ---  ------ -- -------    -- - -------- ------------ -     ----    "
vc  "-  -----------  -------   ---------- -  -- -   -  - --  -----  -  -- ----- ---- ----- - --- --- - ---  -- "

|#

;; chord generation formula as developed for The White Light of Wonder

(setq sym-rh (mapcar 'compress2 (gen-collect 0.12 len-n :list
   '(gen-random nil (get-random 2 2) '(a b c d e f g h i j k l))))
    sym-lh (mapcar 'compress2 (gen-collect 0.23 len-n :list
   '(gen-random nil (get-random 1 1) (reverse '(a b c d e f g h i j k l)))))
)


(setq rh-lh-mix (symbol-mix sym-rh (symbol-transpose -12 sym-lh)))
;; (fd-b dk-f af-k jh-d . . .


(setq vn-1  (symbol-demix 3 rh-lh-mix :sort) ; (f k f j d i . . .
      va-1  (symbol-demix 2 rh-lh-mix :sort)  ; (d d a h c b a . . .
      vc-1  (symbol-demix 1 rh-lh-mix :sort)) ; (-b –f –k –d . .
```

Using the abovementioned **instrument-to-string** function, a beat/space timesheet can be produced for each instrumental line by checking whether or not it was picked in any of the collections above. The timesheet is shown below the code for reference.

The method of generating chordal material in this piece was first used in the piano piece *The White Light of Wonder*. First, three symbols are created: a pair in the 'right hand' and a single symbol in the 'left hand'.

A set of aggregate chords is then created by transposing each individual 'left hand' symbol by -12 steps and combining them with the 'right hand' pairs. Although the chords are for the most part triads those with unisons, such as bb-d will be resolved to pairs such as b-d by **symbol-mix**.

For each chord **symbol-demix** is used to sort them into ascending order, stripping out the 3rd (uppermost) values for the violin and so on. In the case of a pair, such as b-d, this will be sorted as –db, resulting in no third value for the violin. The violin will rest in this case.

```
(setq zne-1 (gen-repeat  len-n '(1/8)))

(setq zn-1 (zone-convertor zne-1 (instrument-to-string 'vn output-all))
      zn-2 (zone-convertor zne-1 (instrument-to-string 'va output-all))
      zn-3 (zone-convertor zne-1 (instrument-to-string 'vc output-all)))
```

A series of 107 1/8 zones lengths is created. **Zone-convertor** is used to create a plan of sounding and resting zones in, for example (1/8 1/8 -1/8 1/8 . . .)

```
(setq vn-1s (mapcar 'list (symbol-swallow zn-1 vn-1)) ; ((f) (k) (=) (j) (=) (i . . .
      va-1s (mapcar 'list (symbol-swallow zn-2 va-1)) ; ((d) (d) (a) (h) (c) (=) (a) . .
      vc-1s (mapcar 'list (symbol-swallow zn-3 vc-1))) ; ((-b) (=) (=) (-d) (-h) . .
```

These zone plans are used to 'swallow' symbols in each part which will not be played, replacing them with rests (=).

```
;; adding ornamental figures

(setq vn-1sx (p-replace-sections (template-invert (calculate-rests zn-1))
                                 (do-section (gen-template 0.12 1 2 len-n)
                                             '(symbol-shuffle
                                               (gen-variants-t
                         nil  (pick-random '(0 1 3)) nil x)) vn-1s) vn-1s))

;; ((f) (n k l m) (=) (j k j) (=) (l k j i) . . .


(setq va-1sx (p-replace-sections (template-invert (calculate-rests zn-2))
                                 (do-section (gen-template 0.147 1 3 len-n)
                                             '(symbol-shuffle
              (gen-variants-t nil (pick-random '(0 1 3)) -5  x )) va-1s) va-1s))


(setq vc-1sx (p-replace-sections (template-invert (calculate-rests zn-3))
                                 (do-section (gen-template 0.14 1 4 len-n)
                                             '(symbol-shuffle
              (gen-variants-t nil (pick-random '(0 1 3)) nil  x )) vc-1s) vc-1s))
```

Each instrument may play an ornament on the note that has been assigned them in each zone. Here the non-resting zones are located (using **calculate-rests** and inverting the data. A second template is created which decides which notes to ornament. The ornaments are created using **gen-variants-t**, which normally states the 'theme' first. Therefore it is also shuffled into a random order.

This process is repeated below for viola and violoncello.

```
;; adding note-lengths to ornaments

(def-neuron conv
  (in 1 '2) (pick-random '(((1/16 1/16))((-1/16 1/32 1/32))((1/32 1/32 -1/16 ))))
  (in 1 '3) (pick-random '(((1/16 1/32 1/32))((-1/32 1/32 1/32 1/32))))
  (in 1 '4) (pick-random '(((1/32 1/32 1/32 1/32))((-1/16 1/64 1/64 1/64 1/64))))
  (otherwise (pick-random '(((1/8)) ((1/8))((-1/16 1/16))((1/16 -1/16))))))

(setq rhy-vn (run-neuron 'conv (mapcar 'length vn-1sx))
      rhy-va (run-neuron 'conv (mapcar 'length va-1sx))
      rhy-vc (run-neuron 'conv (mapcar 'length vc-1sx)))
```

Def-neuron is used to define a ruleset that will assign durations to all the symbols in each instrumental line. A random rhythmic scheme is chosen dependent on the length of the symbols in each zone. These rules are applied by calling **run-neuron** on each set of symbolic data. So, for a two-symbol ornamentation (k l) we might choose (1/16 1/16) or even interpolate a rest (-1/16 1/32 1/32) which the compiler will interpret as though the related symbol set was (= k l).

```
;; adding dynamics based on number of ornaments in a bar

(def-neuron dynx
(in 1 '2) (pick-random '(((32 30)) ((45 40)) ((64 60)) ((72 65))))
(in 1 '3) (pick-random '(((32 30 25)) ((45 40 35 )) ((64 60 55)) ((72 70 65))))
(in 1 '4) (pick-random '(((32 30 25 30)) ((45 40 35 40 )) ((64 60 55 60)) ((72 70 65 70))))
(otherwise  (pick-random '(((32)) ((45)) ((64)) ((72))))))

(setq dhy-vn (run-neuron 'dynx (mapcar 'length vn-1sx))
      dhy-va (run-neuron 'dynx (mapcar 'length va-1sx))
      dhy-vc (run-neuron 'dynx (mapcar 'length vc-1sx)))
```

A similar rule-based approach is used to create sets of dynamics (MIDI velocities) for each instrument.

```
;; score-template

(def-tonality
 vn (activate-tonality (chromatic a 5))
 va (activate-tonality (chromatic a 4))
 vc (activate-tonality (chromatic a 3))
)
```

These final definitions declare to SCOM how to compile the above data into a musical (MIDI) format. Here a single tonality is applied to the instrumental lines: a chromatic scale based on an A tonic, spaced an octave apart for each voice.

```
(def-symbol
 vn vn-1sx
 va va-1sx
 vc vc-1sx
)
```

The symbols that have been generated above from breaking apart chords and ornamenting them are now formally assigned to what will become MIDI channels. In the violin the symbol ‘a’ therefore relates to the pitch A5, ‘b’ to A#5, ‘c’ to B5 and so on.

```
(def-length
 vn rhy-vn
 va rhy-va
 vc rhy-vc
)
```

Assign length data to each instrument.

```
(def-duration
 vn rhy-vn
 va rhy-va
 vc rhy-vc
)
```

The duration of each note is equal to the length (e.g. no staccato notes).

```
(def-velocity
 vn dhy-vn
 va dhy-va
 vc dhy-vc
)
```

Assign velocity data to each instrument.

```
(def-zone
 default zne-1
)
```

The definition of zones will dictate to the compiler how much material to output. Here we want all 107 1/8 duration zones mapping on to the 107 phrases of the trio. Note that repeats could be created, for example a zone length of (2/8 1/8) would play the first phrase twice and the second phrase once and then stop.

```
(def-channel
 vn 1
 va 2
 vc 3
)

(def-tempo 30)

(compile-instrument-p "ccl;output:" "trio-1b"
 vn
 va
 vc
)
```

Assign MIDI channels, tempo and then render to a file.

```
;; String Trio
; 2/3
#|
vn   - violin
va   - viola
vc   - cello
|#

;; Functions necessary for this section (in addition to those of section 1)

(defun gen-process-list (f-expr values patterns)
(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)))
```

Gen-process-list can be used to enable an expression (such as **gen-repeat**, as used below) to be applied to a list, with a secondary list acting as a parameter for that expression. So, if we have two lists:

A = (1 2 3) and B = ((a) (b) (c))

Then `(gen-process-list '(gen-repeat x y ) A  B)` would yield `((a) (b b) (c c c))`.

```
(defun zone-expand (x-by zne-lis) ; adjusted 15.3.04
(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)))
```

Zone-expand receives two lists – one numeric and the other a series of zone lengths (e.g. 1/8). The numeric list is used to multiply each zone length in the second list. So, `(zone-expand '(1 2 3) '(1/4 1/4 1/4))` yields `(1/4 1/2 3/4)`.

```
;; String trio - section two

(setq moments
(build-array
 ; column 0     1      2    3

      '(( vn    vn    vc   va)
        ( va    vc    va   vn)
        ( vc    va    vn   vc)
)))

(setq lisn '(0 1 2))

(setq output-all
     (mapcar 'remove-duplicates
              (delete 'nil
                  (gen-collect 0.22 120 :list
                        '(pick-array
                         (pick1 nil lisn)
                         (pick1 nil lisn) (get-random 0 4) moments
                         (pick1 nil '(:left
                                      :right
                                      :up
                                      :down)))))))

(setq len-n (length output-all))

; output of output-all

#|
((va vc vn) (va vn) (va) (vn va vc) (va vc) (vn vc) (va vc vn)(vn vc) (vc vn) (vc va vn) (va vc vn) (vn va vc)
(va vc) (vc vn va) (vn va) (vn) (vn vc) (va vn vc) (vc vn va) (vc) (va vc) (vc vn) (vc vn) (vn va) (vn va) (va)
(vc va) (va vn vc) (va vc vn) (vc vn) (va vc vn) (vc vn va) (vn vc) (vc va vn) (vc va) (vc va vn) (vn) (vn vc)
(vn) (va) (va vc) (vc va) (vn) (vc) (va vn) (va) (va vn) (vn va vc) (va) (vn va) (vc) (va) (va vc vn) (vc vn)
(va vn) (vn va) (vc) (va vc) (vc va vn) (va vn) (vn va vc) (va vc vn) (vn) (vn) (vc vn) (vn) (va) (vc va vn)
(vc) (va) (vc) (va vc vn) (vn vc va) (va vc) (va vc vn) (va vn) (vn va vc) (va vn vc) (vc va vn) (vn vc) (va vn)
(vn vc va) (vc vn va) (vn va vc) (vc va vn) (vc va) (va) (vc va vn) (va) (vc va vn) (vc va vn) (vn va vc) (vn)
(vc va) (vc) (vc) (vn) (vc) (vn) (vc va) (vc va vn) (vc va) (va) (vn) (vc) (vn vc) (vn))
|#
```

The same approach to choosing instrumental interplay as section 1 is employed here.

```
;; instrumentation - activity and silence

(instrument-to-string 'vn output-all)

(instrument-to-string 'va output-all)

(instrument-to-string 'vc output-all)

#| ;; one minute of timesheet for trio

vn  "-- - ------- ------  ----  ------- ----   - - -- -  ----  -------- -   -- -----------  - ----   - - -  - --"
va  "----- -  ------  -- -  ------ -- ---   ---  ------ -- -- -----    -- - -------- ----------- -     ----    "
vc  "-  ----------  -------   ---------- -  -- -   -  - --  --- --  -  -- ----- ---- ----- - --- --- - ---  -- "

|#

;; chord generation formula as developed for The White Light of Wonder

(setq sym-rh (mapcar 'compress2 (gen-collect 0.12 len-n :list

'(gen-random nil (get-random 2 2) '(a b c d e f g h i j k l))))
 sym-lh (mapcar 'compress2 (gen-collect 0.23 len-n :list
'(gen-random nil (get-random 1 1) (reverse '(a b c d e f g h i j k l)))))
)

(setq rh-lh-mix (symbol-mix sym-rh (symbol-transpose -12 sym-lh)))
;; (fd-b dk-f af-k jh-d . . .

(setq vn-1  (mapcar 'list (symbol-demix 3 rh-lh-mix :sort)) ; (f k f j d i . . .
      va-1  (mapcar 'list (symbol-demix 2 rh-lh-mix :sort)) ; (d d a h c b a . . .
      vc-1  (mapcar 'list (symbol-demix 1 rh-lh-mix :sort))) ; (-b -f -k -d . . .

(setq iteration (gen-random 0.1 120 '(2 3)))

(setq vn-1s (gen-process-list '(gen-repeat x y ) iteration vn-1)
      va-1s (gen-process-list '(gen-repeat x y ) iteration va-1)
      vc-1s (gen-process-list '(gen-repeat x y ) iteration vc-1))
```

'Chords' are generated and split between the voices as in section 1. However, here they are not ornamented, but **gen-process-list** is used with the **gen-repeat** function to create 2 or 3 repetitions of each note across all the instruments.

```
(def-neuron rhy-x
  (in 1 '2) '((1/8 1/8))
  (in 1 '3) '((1/8 1/8 1/8))
  (otherwise (in 1 0)))

(setq iter-1 (p-remove (e-position '= (flatten vn-1))
                       (run-neuron 'rhy-x iteration))
      iter-1x (mapcar 'length iter-1))
```

A rhythmic scheme of 1/8 notes is created – corresponding to the lengthened, repeated lists.

```
(def-neuron dur-x
  (in 1 '2) '((1/8 1/16))
  (in 1 '3) '((1/8 1/16 1/16))
 (otherwise (in 2 0)))
```

Note also the inclusion of durational rules: although the phrase may last, for example, two 8th notes, the second of these is to be played staccato (having an ‘actual’ duration of 1/16th),

```
(def-neuron dyn-x
  (in 1 '2) '((60 45))
  (in 1 '3) '((60 45 35))
  (in 1 '4) '((40 45 50 55))
  (in 1 '6) '((40 45 50 55 60 65))
 (otherwise (in 1 0)))
```

A set of dynamic rules – note that although we do not (yet) have phrases of any more than 3 notes, these rules will be applied later after an ornamentation process.

```
(def-neuron orn-x
  (in 1 '4) '((1/16 1/16 1/16 1/16))
  (in 1 '6) '((1/16 1/16 1/16 1/16 1/16 1/16))
  (otherwise (in 2 0)))
```

A second set of rhythmic rules for assigning note-lengths to ornamented symbols.

```
(setq vn-orn
      (do-section (gen-template 0.5 1 4 107)
                  '(symbol-inversion (car x) (list-a-scale (car x) (pick1 nil '(4 6))))
                   (mapcar 'list (p-remove (e-position '= (flatten vn-1)) (flatten vn-1)))))
; ((f) (k) (f) (j) (d) (i h g f) (k) (b) (f) (j) (h) (l k j i) (l k j i h g) (b a -b -c -d -e) . . .


(setq va-orn
      (do-section (gen-template 0.51 1 5 107)
                  '(symbol-shuffle (list-a-scale (car x) (pick1 nil '(4 6))))
                  (mapcar 'list (p-remove (e-position '= (flatten vn-1)) (flatten va-1)))))
; ((d) (i e d g h f) (a) (h) (e f d c) (f e c g b d) . . .

(setq vc-orn
      (do-section (gen-template 0.52 1 6 107)
                  '(list-a-scale (car x) (pick1 nil '(4 6)))
                  (mapcar 'list (p-remove (e-position '= (flatten vn-1)) (flatten vc-1)))))
; ((-b) (-f) (-k) (-d -c -b a) (-h) (-m) (-h -g -f -e -d -c) . . .
```

Each instrument is ornamented in a different way. In each instrument a template is generated, each one having a different random seed (0.5, 0.51 and 0.52) and thus having different results. In the violin selected symbols have **the list-a-scale** function applied on them for a length of 4 or 6 steps. **List-a-scale** can turn a list or symbol into a scale – in this case using ascending steps, so the symbol `(i)` may become `(i j k l)`. These groups are then inverted around the first symbol, so our previous example becomes a descending phrase `(i h g f)`.
In the viola the ornamentations are not inverted, rather they are shuffled. The cello ornaments are left as ascending sequences.

```
(setq vn-ornr (run-neuron 'orn-x (mapcar 'length vn-orn) (mapcar 'list iter-1))
      va-ornr (run-neuron 'orn-x (mapcar 'length va-orn) (mapcar 'list iter-1))
      vc-ornr (run-neuron 'orn-x (mapcar 'length vc-orn) (mapcar 'list iter-1)))
```

The ornamentation neuron ruleset is now applied to yield appropriate rhythmic data for the ornamented symbols.

```
(setq vn-l (mapcar 'length vn-ornr) ; (2 2 3 2 3 4 2 2 2 3 3 4 6 6 4 3 4 . . . to drive neural processing
      va-l (mapcar 'length va-ornr)
      vc-l (mapcar 'length vc-ornr))


(setq durt-vn (run-neuron 'dur-x vn-l (mapcar 'list vn-ornr ))
      durt-va (run-neuron 'dur-x va-l (mapcar 'list va-ornr ))
      durt-vc (run-neuron 'dur-x vc-l (mapcar 'list vc-ornr )))


(setq dyn-vn (run-neuron 'dyn-x vn-l)
      dyn-va (run-neuron 'dyn-x va-l)
      dyn-vc (run-neuron 'dyn-x vc-l))
```

The ornamentation and dynamic neuron rulesets are now applied to yield appropriate rhythmic and dynamic data for the ornamented symbols.

```
;; score-template

(def-tonality
 vn (activate-tonality (chromatic a 5))
 va (activate-tonality (chromatic a 4))
 vc (activate-tonality (chromatic a 3))
)

(def-symbol
 vn vn-orn
 va va-orn
 vc vc-orn
)

(def-length
 vn vn-ornr
 va va-ornr
 vc vc-ornr
)
```

The **zone-expand** function was explained at the start of the second section code. Here it is driven by a long series of 1s and 2s, where a 2 will double the length of the zone, yielding a repeat of the material.

```
(def-duration
 vn durt-vn
 va durt-va
 vc durt-vc
)

(def-velocity
 vn dyn-vn
 va dyn-va
 vc dyn-vc
)

(def-zone
default   (zone-expand
            (append (gen-repeat 15 '(1)) '(2) (gen-repeat 40 '(1)) '(2) '(1 1 1 1)'(2) '(1 1 1 1 1 1 1 1 1) '(2)
            '(1) '(2)'(1) '(2) (gen-repeat 24 '(1)))
         (make-zone vn-ornr  :ratio)))

#|
(1/4 1/4 3/8 1/4 3/8 1/4 1/4 1/4 1/4 3/8 3/8 1/4 3/8 3/8 1/4 3/4 1/4 1/4 1/4 3/8 3/8 1/4 1/4 3/8 3/8 1/4 3/8 3/8 1/4
1/4 3/8 1/4 3/8 3/8 1/4 1/4 3/8 1/4 3/8 1/4 1/4 1/4 1/4 1/4 3/8 3/8 3/8 3/8 3/8 1/4 3/8 3/8 1/4 1/4 3/8 1/4 3/4 1/4
3/8 1/4 3/8 1/2 3/8 3/8 3/8 1/4 1/4 1/4 3/8 3/8 1/4 3/4 1/4 1/2 3/8 3/4 1/4 3/8 1/4 3/8 3/8 1/4 1/4 3/8 1/4 1/4 3/8
1/4 1/4 3/8 1/4 1/4 1/4 3/8 3/8 1/4 1/4 1/4 3/8 1/4)
|#

(def-channel
 vn 1
 va 2
 vc 3
)

(def-tempo 90)

(compile-instrument-p "ccl;output:" "trio-2"
 vn
 va
 vc
)
```

```
; String Trio
; 3/3
#|
vn   - violin
va   - viola
vc   - cello
|#

;; No additional functions required in this section

(setq lis-x '(1 2 8 1 2 3 7 8 1 2 3 6 7 8 1 2 3 4 5 6 7 8 1 2 3 4 5))

(setq ch-lis (e-substitute
 (g-chord 0.251 1 3 7 -5 (gen-repeat 8 ' ((-ndf))))
 (list-a-scale 1 8)
  (gen-palindrome lis-x)))

; ((-hjl) (-r-bb) (-gkm) (-iik) (-kgi) (-hjl) . . .
```

The material in the final section is derived from the chord `(-ndf)`, which appears in the opening of both sections 1 and 2. Initially **g-chord** is used with **gen-repeat** to create 8 variations transposed between -5 and 7 semitones. **E-substitute** can then use this list to pick from assigning chords from the list to the numbers 1 to 8 and mapping them to a palindrome based on the numbers in the **lis-x** list. So, the eight chords are:

```
  1       2       3      4
(-hjl) (-r-bb) (-gkm) (-iik)
  5       6       7      8
(-kgi) (-hjl) (-qac) (-meg)
```

And the result of **e-substitute** with the **lis-x** palindrome:

```
  1      2      8      1      2       3
(-hjl) (-r-bb) (-meg) (-hjl) (-r-bb) (-gkm) etc.
```

```
(setq vn-1  (symbol-demix 3 (flatten ch-lis) :sort)
      va-1  (ambitus :transpose '-e 'k (symbol-demix 2 (flatten ch-lis) :sort))
      vc-1  (symbol-demix 1 (flatten ch-lis) :sort))
```

A unique line is created from the palindromic chord groupings created above and stored as **ch-lis**. In the violin part the palindrome is flattened so `((-hjl) (-rbb)...)` becomes `(-hjl -rbb…)`. As in the first two movements, **symbol-demix** is used to choose one of the values in each grouping, in this case the third value, so `(l b)` in the above example. The same is true for the cello part, but with the first value: `(-h –r)` in the above example. The viola is a little different. It also uses **symbol-demix** on the second value, coupled with the **ambitus** function, which, when used with the **:transpose** flag wraps symbols in the range -e to k: -r, for example, becomes h. This function can be used to constrain symbols to the range of a given instrument.

```
(setq vn-1s (p-replace-sections (template-invert (gen-template 0.5 1 5 (length vn-1)))
                    (mapcar 'list vn-1)
                    (do-section (gen-template 0.5 1 4 (length vn-1))  '(symbol-shuffle x)
                                (gen-process-list '(list-a-scale x  y) vn-1
                                                   (append (mapcar 'abs (get-interval :all vn-1)) '(1))))))

; ((l) (b) (g) (l) (b) (v r n u p o m t q s) . . .
```

The violin voice is expanded using a template the same length as the number of symbols in vn-1. This code creates interventions based on intervallic relations between each pitch. First **list-a-scale** is used with **get-interval** to create scales based on the number of intervals between consecutive tones, for example: `(l b g l)` becomes `(((l m n o p q r s t u) (b c d e f) (g h i j k)(l))`. These are then shuffled and used to replace sections according to a template scheme.

```
(find-unique (mapcar 'abs (get-interval :all vn-1)))
; to identify rhythmic groupings
;(sort< '(10 5 11 4 1 9 2 3)) ; (1 2 3 4 5 9 10 11)
```

This code is used to work out how many rhythmic groupings (based on intervals as above) are in each voice, in this case the violin. These values are then used to define the rhythm neuron later.

```
(setq va-1s (p-replace-sections (template-invert (gen-template 0.513 1 5 (length va-1)))
                                      (mapcar 'list va-1)
                                      (do-section (gen-template 0.513 1 5 (length va-1))  '(symbol-shuffle x)
                                                  (gen-process-list '(list-a-scale x  y) va-1
                                                  (append (mapcar 'abs (get-interval :all va-1)) '(1))))))
;; ((j) (j h i) (i f g h e) (k j) (h) (k) . . .

(find-unique (mapcar 'abs (get-interval :all va-1)))
;(sort< '(2 3 5 10 4 1 9)) ;(1 2 3 4 5 9 10)
```

```
(setq vc-1s (p-replace-sections (template-invert (gen-template 0.52 1 5 (length vc-1)))
                                (mapcar 'list vc-1)
                                (do-section (gen-template 0.52 1 6 (length vc-1))  '(symbol-shuffle x)
                                            (gen-process-list '(list-a-scale x  y) vc-1
                                            (append (mapcar 'abs (get-interval :all vc-1)) '(1))))))
;; ((-h) (-b) (-m) (-f -g -d -e -h -c) (-b) . . .

(find-unique (mapcar 'abs (get-interval :all vc-1)))
; (sort< '(6 11 5 10 4 1 9 2 3)) ; (1 2 3 4 5 6 9 10 11)

(def-neuron rhys
 (in 1 '2) '((1/8.))
 (in 1 '3) '((1/8 1/8 1/8))
 (in 1 '4) '((1/8 1/8 1/16 1/16))
 (in 1 '5) '((1/8 1/16 1/16 1/16 1/16))
 (in 1 '6) '((1/16 1/16 1/16 1/16 1/16 1/16))
 (in 1 '9) '((1/8-3 1/8-3 1/8-3 1/8-3 1/8-3 1/8-3 1/8-3 1/8-3 1/8-3))
 (in 1 '10) '((-1/16 1/32 1/32 1/32 1/32 1/32 1/32 1/32 1/32 1/32 1/32))
 (in 1 '11) '((-1/32 1/32 1/32 1/32 1/32 1/32 1/32 1/32 1/32 1/32 1/32))
 (otherwise '((1/4.))))

(setq vn-rhy (run-neuron 'rhys (mapcar 'length vn-1s))
      va-rhy (run-neuron 'rhys (mapcar 'length va-1s))
      vc-rhy (run-neuron 'rhys (mapcar 'length vc-1s)))

;; violin rhythmics ((1/4.) (1/4.) (1/4.) (1/4.) (1/4.) (-1/16 1/32 1/32 1/32 1/32 1/32 1/32 1/32 1/32 1/32 1/32)...

;; score-template

(def-tonality
 default (activate-tonality (chromatic a 4))
)

(def-symbol
 vn vn-1s
 va va-1s
 vc vc-1s
)
```

Having created the lines for each voice, including the scale-derived ‘interventions’, rhythmic data is applied to the symbolic structure. Note that for a two-note phrase the single zone value `(1/8.)` is applied – this is interpreted by the compiler as all notes in that ‘zone’ having `(1/8.)` values and is equivalent of writing `(1/8. 1/8.)`. All rhythmic phrases should amount to 1/4. duration.

```
(def-length
 vn  vn-rhy
 va  va-rhy
 vc  vc-rhy
)


(def-velocity
 vn '(40)
 va '(40)
 vc '(40)
)


(def-zone
 default (gen-repeat (length ch-lis) '(3/8))
)

(def-channel
 vn 1
 va 2
 vc 3
)


(def-tempo 50)

(compile-instrument-p "ccl;output:" "trio-3"
 vn
 va
 vc
)
```