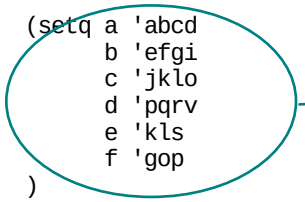
```
;;; Omphalos (Piano and 4 percussionsts) section 1

;; material


' drum-kits

(set-tonality kit-1
              d& 5        ; a   low bongo
              c 5         ; b   high bongo
)

(set-tonality kit-2
              e 5          ; a   low conga
              e& 5         ; b   high bongo
              d 5          ; c   mute conga
)

(set-tonality kit-3
             g& 5         ; a   low timbale
              f 5         ; b   high timbale
)

(set-tonality kit-4
              f 6         ; a   low wood-block
              e 6         ; b   high wood-block
)



; making clusters (as atoms)
; transcribed from improvised explorations at the keyboard using the Sym-Clavier set to a chromatic tonality

(setq a 'abcd
      b 'efgi
      c 'jklo
      d 'pqrv
      e 'kls
      f 'gop
)
```



These clusters were improvised using the Sym-Clavier, and the tonality (chormatic c 4). See the introductory notes for the equivalent of these symbol lists in musical notation.

```
;; basic percussion rhythms associated with particular instruments
(setq a1 '(= a b a) ; w-block
      b1 '(b b = a) ; timbale
      c1 '(a = a b) ; congas
      d1 '(b a b b) ; bongos
      e1 '(= ab = ab)
      f1 '(a = b ab)
)

(setq material (list a b c d e f '=))

(setq ratio-values2 '(1 2 3 3 2 1 4))


(setq gen-all
(prog (out)
  (for i 1 1 4 nil ; enables loop start from 1 rather than 0 (dotimes)
          (setq out
                (append out
                        (list
                        (gen-symbol-ratio 0.13
                            (change-length :times i ratio-values2)
                            material)))))
                       (return out)))

(setq gen-1
      ; splitting up some chords into arpeggios
      (do-section  (gen-template 0.7 1 3 (length (first gen-all)))
                        '(symbol-melodize x)
                        (mapcar 'list (first gen-all))))


(setq gen-2
      ; plus reversing the direction of some arpeggios
      (do-section  (gen-template 0.7 1 3 (length (second gen-all)))
                  '(reverse x)
                  (do-section  (gen-template 0.7 1 4 (length (second gen-all)))
                        '(symbol-melodize x)
                        (mapcar 'list (second gen-all)))))
```

There are four percussion instruments in this piece, each of which plays its own rhythmic phrase. There are also two other phrases (e1 & f1) which can be played by any of the instruments.

This code generates a long list consisting of the above defined clusters (a-f) or rests. The clusters and appear according to the corresponding ratio values (ratio-values2), so that the rest symbol and clusters c and d will be most widely represented, while clusters a and f will least represented. The code executes four times, the ratios (and thus the number of symbols generated) being doubled each time. This results in four lists of increasing length, which will provide the structure of the movement.

In the first list symbol-melodize is used to turn random clusters into arpeggios, so:
(= abcd jklo) might become ((=) (abcd) (j k l o))
In order to do this using do-section, a template must be created, in which x indicates a phrase to arpeggiate. Gen-template is used to create a template of random x and = values of the same length of the first list.

Second list is processed to result in some downward arpeggios.

```
(setq gen-3
      ; plus randomizing octave positions of some notes/chords
      (do-section  (gen-template 0.5 1 5 (length (third gen-all)))
                   '(randomize-octaves 0.23 nil -1 0 x)
                   (do-section  (gen-template 0.6 1 3 (length (third gen-all)))
                   '(reverse x)
                   (do-section  (gen-template 0.7 1 4 (length (third gen-all)))
                          '(symbol-melodize x)
                            (mapcar 'list (third gen-all))))))
```

In the third section, two instances of do-section, each with different random seeds, are used to create downward arpeggios and also to randomly transpose phrases (e.g. chords or arpeggios) down by an octave.

```
(defun compr-mkr (lis low high seed)
(if seed (init-rnd seed))
(cond ((is-pause-symbol (car lis)) '(=))
      (t  (symbol-compress (get-random low high) lis))))
```

This function is used to compress or expand the symbols in a phrase. If the phrase is a rest, then a rest is returned. Otherwise symbol-compress is executed using a randomly determined percentage of compression (-50 to -100, see below). Note that because the compression is negative, the effect is an expansion of the phrase or cluster!

```
(setq gen-4
      ; plus expanding and shuffling
 (do-section (gen-template 0.9 1 1 (length (fourth gen-all)))
                       '(symbol-shuffle x nil)
      (do-section (gen-template 0.8 1 3 (length (fourth gen-all)))
                       '(compr-mkr x -50 -100 nil)
      (do-section  (gen-template 0.5 1 5 (length (fourth gen-all)))
                   '(randomize-octaves 0.23 nil -2 0 x)
                   (do-section  (gen-template 0.6 1 3 (length (fourth gen-all)))
                   '(reverse x)
                   (do-section  (gen-template 0.7 1 4 (length (fourth gen-all)))
                          '(symbol-melodize x)
                          (mapcar 'list (fourth gen-all)))))))))
```

The fourth and longest section of this movement uses three templates, which control symbol-shuffle, compr-mkr (defined above) and downward arpeggios with transpositions. The intent is that the piece should be like a continually unfolding improvisation around the basic clusters.

```
(setq all-gen (append gen-1 gen-2 gen-3 gen-4))
```

```
;; rhythmic material

(defun rhy-mkr (lis)
(cond ((is-chord (car lis)) '1/8)
       ((equal lis '(=)) '1/4)
       ((equal (length lis) 3) '(1/32 1/32 1/32))
        (t (build-list '1/16 (length lis)))))

(setq rhy-1 (flatten (mapcar 'rhy-mkr gen-1))
      rhy-2 (flatten (mapcar 'rhy-mkr gen-2))
      rhy-3 (flatten (mapcar 'rhy-mkr gen-3))
      rhy-4 (flatten (mapcar 'rhy-mkr gen-4)))

(setq rhy-all (append rhy-1 rhy-2 rhy-3 rhy-4))


;; using neurons to make associations v clusters and rhythmic groups

(def-neuron match-p1
(in 1 'abcd) a1
(in 1 'kls) e1
(otherwise '=)
)

(setq p1-sym (run-neuron 'match-p1 (flatten all-gen)))

(def-neuron match-r1
(in 1 'abcd) '(1/32 1/32 1/32 1/32)
(in 1 'kls) '(1/32 1/32 1/32 1/32)
(otherwise (in 2 0))
)

(setq p1-rhy (run-neuron 'match-r1 (flatten all-gen)
                         rhy-all))

(def-neuron match-p2
(in 1 'efgi) b1
(in 1 'gop) f1
(otherwise '=)
)

(setq p2-sym (run-neuron 'match-p2 (flatten all-gen)))
```

Now we have defined the symbolic data for the piano we need to create the corresponding rhythmic material. This function analyses the symbolic data and creates appropriate length values, so chords have 1/8 duration, rests 1/4, 3 note phrases 1/32 and other notes 1/16.

We now begin to define the percussion parts. The percussion gestures are assocaited with particular chords in the piano part, in this case a chord of (abcd) will correspond to the wood-block playing a1 (= a b a)

Here a neuron is similarly used to create length values corresponding with the above symbolic gestures.

The rhythmic content for the other three instruments is generated in a similar manner.

```
(def-neuron match-r2
(in 1 'efgi) '(1/32 1/32 1/32 1/32)
(in 1 'gop) '(1/32 1/32 1/32 1/32)
(otherwise (in 2 0))
)

(setq p2-rhy (run-neuron 'match-r2 (flatten all-gen)
                         rhy-all))

(def-neuron match-p3
(in 1 '=) 'c
(in 1 'jklo) c1
(in 1 'kls) e1
(otherwise '=)
)

(setq p3-sym (run-neuron 'match-p3 (flatten all-gen)))

(def-neuron match-r3
(in 1 'jklo) '(1/32 1/32 1/32 1/32)
(in 1 'kls) '(1/32 1/32 1/32 1/32)
(otherwise (in 2 0))
)

(setq p3-rhy (run-neuron 'match-r3 (flatten all-gen)
                         rhy-all))
(def-neuron dyn-r3
 (in 1 'c) '120
(otherwise (in 2 0))
)

(setq p3-dyn (run-neuron 'dyn-r3
                         p3-sym
                         (vector-to-list
                          (vector-round 120 44
                                        (gen-noise-white
                                         (length (filter-delete '= p3-sym)))))))


(def-neuron match-p4
(in 1 'pqrv) d1
(in 1 'gop) f1
(otherwise '=)
)
```

```
(setq p4-sym (run-neuron 'match-p4 (flatten all-gen)))

(def-neuron match-r4
(in 1 'pqrv) '(1/32 1/32 1/32 1/32)
(in 1 'gop) '(1/32 1/32 1/32 1/32)
(otherwise (in 2 0))
)

(setq p4-rhy (run-neuron 'match-r4 (flatten all-gen)
                         rhy-all))



;; score

(def-tonality
 piano (activate-tonality
; follows the root of each cluster
       (chromatic c 4) (chromatic e 4)  (chromatic a 4) (chromatic d# 4))
perc1 kit-1
perc2 kit-2
perc3 kit-3
perc4 kit-4
)

(def-symbol
 piano (flatten all-gen)
 perc1 p4-sym
 perc2 p3-sym
 perc3 p2-sym
 perc4 p1-sym

)

(def-length
 piano rhy-all
 perc1 p4-rhy
 perc2 p3-rhy
 perc3 p2-rhy
 perc4 p1-rhy
)

(def-duration
 piano (change-length :divide 2 rhy-all)
```

Each section has its own tonality based on the root notes of the first four clusters generated at the start of the piece.

The durations of the notes in the piano are divided by 2 - this will give the resulting MIDI output a staccato quality. Useful for hearing and interpreting the music, since otherwise the notes tend to blur into one another.

```
)


(def-zone
 default (mapcar 'make-zone (list rhy-1 rhy-2 rhy-3 rhy-4))
)

(def-velocity
 piano (vector-round 44 120 (gen-noise-white (length (get-symbols-of 'piano))))
 perc1 (vector-round 120 44 (gen-noise-white (length (filter-delete '= p4-sym))))
 perc2  p3-dyn
 perc3 (vector-round 120 44 (gen-noise-white (length (filter-delete '= p2-sym))))
 perc4 (vector-round 120 44 (gen-noise-white (length (filter-delete '= p1-sym))))

)

(def-channel
 piano 1
 perc1 10
 perc2 10
 perc3 10
 perc4 10
)

(def-program gm-sound-set
 piano acoustic-grand-piano
)

(def-tempo 80)

(compile-instrument-p "ccl;output:" "Omphalos1"
 piano
 perc1
 perc2
 perc3
 perc4
)
```

```
#|

; gen-1
((=) (pqrv) (jklo) (jklo) (pqrv) (k l s) (efgi) (=) (kls) (abcd) (jklo)
(e f g i) (=) (g o p) (p q r v) (=))

; gen-2
((pqrv) (kls) (abcd) (jklo) (jklo) (v r q p) (=) (pqrv) (efgi) (=) (=) (s l k)
(=) (s l k) (=) (jklo) (o l k j) (v r q p) (i g f e) (jklo) (pqrv) (gop) (abcd)
(=) (kls) (i g f e) (jklo) (efgi) (=) (gop) (v r q p) (=))

; gen-3
((kls) (=) (kls) (jklo) (efgi) (-d -c -b c) (pqrv) (=) (kls) (pqrv) (abcd)
(d e f j) (-c -b g) (-i -h -g -e) (=) (abcd) (-d -c -b c) (=) (-d -c -b c)
(pqrv) (jklo) (pqrv) (=) (d e f j) (efgi) (=) (=) (kls) (=) (gop) (=) (j k l o)
(jklo) (pqrv) (efgi) (jklo) (pqrv) (-g c d) (abcd) (=) (kls) (-i -h -g -e)
(jklo) (efgi) (=) (gop) (pqrv) (=))

; gen-4
((=) (abcd) (jklo) (kls) (efgi) (-c -n -g) (jklo) (pqrv) (=) (jkmr) (kly)
(-d -o -n c) (=) (-c -n -g) (=) (pqrv) (-c -n -g) (=) (d -i -h j) (efhl)
(efhl) (jklo) (pqrv) (=) (gop) (d -i -h j) (abcd) (pqrv) (kls) (efgi) (=)
(d c a b) (jkmq) (=) (jklo) (pqrv) (jklo) (d -i -h j) (=) (pqrv) (efgi) (=)
(=) (kls) (gtu) (gop) (=) (jklo) (jklo) (pqrv) (efgi) (jklo) (pqsz) (=) (abde)
(=) (-g -n -c) (efgi) (jklo) (efhl) (=) (-g -j -k) (pqrv) (=))

; rhy-1
(1/4 1/8 1/8 1/8 1/8 1/32 1/32 1/32 1/8 1/4 1/8 1/8 1/8 1/16 1/16 1/16 1/16
1/4 1/32 1/32 1/32 1/16 1/16 1/16 1/16 1/4)

; rhy-2
(1/8 1/8 1/8 1/8 1/8 1/16 1/16 1/16 1/16 1/4 1/8 1/8 1/4 1/4 1/32 1/32 1/32
1/4 1/32 1/32 1/32 1/4 1/8 1/16 1/16 1/16 1/16 1/16 1/16 1/16 1/16 1/16 1/16
1/16 1/16 1/8 1/8 1/8 1/8 1/4 1/8 1/16 1/16 1/16 1/16 1/8 1/8 1/4 1/8 1/16
1/16 1/16 1/16 1/4)

; rhy-3
(1/8 1/4 1/8 1/8 1/8 1/16 1/16 1/16 1/16 1/8 1/4 1/8 1/8 1/8 1/16 1/16
1/16 1/16 1/32 1/32 1/32 1/16 1/16 1/16 1/16 1/4 1/8 1/16 1/16 1/16 1/16
1/4 1/16 1/16 1/16 1/16 1/8 1/8 1/8 1/4 1/16 1/16 1/16 1/16 1/8 1/4 1/4 1/8
1/4 1/8 1/4 1/16 1/16 1/16 1/16 1/8 1/8 1/8 1/8 1/8 1/32 1/32 1/32 1/8 1/4
1/8 1/16 1/16 1/16 1/16 1/8 1/8 1/4 1/8 1/8 1/4)
```

```
; rhy-4
(1/4 1/8 1/8 1/8 1/8 1/32 1/32 1/32 1/8 1/8 1/4 1/8 1/8 1/16 1/16 1/16 1/16
1/4 1/32 1/32 1/32 1/4 1/8 1/32 1/32 1/32 1/4 1/16 1/16 1/16 1/16 1/8 1/8 1/8
1/8 1/4 1/8 1/16 1/16 1/16 1/16 1/8 1/8 1/8 1/8 1/4 1/16 1/16 1/16 1/16 1/8
1/4 1/8 1/8 1/8 1/16 1/16 1/16 1/16 1/4 1/8 1/8 1/4 1/4 1/8 1/8 1/8 1/4 1/8
1/8 1/8 1/8 1/8 1/8 1/4 1/8 1/4 1/32 1/32 1/32 1/8 1/8 1/8 1/4 1/32 1/32 1/32
1/8 1/4)


|#
```

```
;;; Omphalos (Piano and 4 percussionsts) section 2

;; functions

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


(defun add-on-repeats (lis-repeats lis-symbols fill)
  "creating symbol-lists that respond to repeat-lists by playing the original
   once then continuing with variants on that original"
  (prog (out el1 el2)
    loop
    (cond ((null lis-symbols) (return out)))
    (setq el1 (car lis-repeats))
    (setq el2 (car lis-symbols))
    (setq out (append out
                      (list (cond ((< 1 el1)
                                   (gen-variants nil 2
                                                 (fill-template el2 fill)))
                                  (t (build-list '=  (length el2))))))))
    (setq lis-repeats (cdr lis-repeats))
    (setq lis-symbols (cdr lis-symbols))
    (go loop)))
```

If the length of a zone is longer than the material in a corresponding phrase then the material is repeated until the end of the zone. This function multiplies the length of a zone - in this movement, phrases will be repeated up to three times using this function.

This function will be used to generate symbolic data for the percussion instruments. When a phrase is repeated more than once (indicated by lis-repeats), streams of rests and symbols (from lis-symbols) are created.

```
;; drum-kits

(set-tonality kit-1
              d& 5       ; a   low bongo
              c 5        ; b   high bongo
)

(set-tonality kit-2
              e 5         ; a   low conga
              e& 5        ; b   high bongo
              d 5         ; c   mute conga
)

(set-tonality kit-3
             g& 5        ; a   low timbale
              f 5        ; b   high timbale
)

(set-tonality kit-4
              f 6        ; a   low wood-block
              e 6        ; b   high wood-block
)


;; material

(setq source '(a b d g h j m)) ; Slonimsky 06

(setq mat
      (gen-hopalong-symbol pnlh (-g d) pnrh (g p) 100 200 300 0.45 144 0))

(setq pnlh1 (filter-delete '(-e -c -b c) pnlh)
      pnrh1 (filter-delete '(i k l o) pnrh))

(setq pnrh2 (symbol-divide (gen-random 0.11  6 '(6 12 18 6)) nil nil pnrh1)
      pnlh2 (symbol-divide (gen-random 0.11  6 '(6 12 18 6)) nil nil pnlh1)
)

(setq rnd-rpt-list (gen-random 0.1234 (length pnrh2) '(1 2 3 2 1)))

(setq p1-sym (add-on-repeats rnd-rpt-list pnrh2 '(a b))
      p2-sym (add-on-repeats rnd-rpt-list pnrh2 '(a b b))
      p3-sym (add-on-repeats rnd-rpt-list pnlh2 '(a a b))
      p4-sym (add-on-repeats rnd-rpt-list pnlh2 '(b b a)))
```

Gen-hopalong-symbol is a fractal algorithm with two outputs, here mapped onto the hands of the pianist (pnlh, pnrh). 144 symbols are generated between (-g d) and (g p) in the left and right hands respectively.

Filter-delete replaces indicated values with rests.

The content is then divided up into phrases of 6-18 symbols in length.

Use the above defined add-on-repeats function to create the symbolic data for the percussion part, based on the list of repeats.

```
(setq dyn
      (gen-hopalong-vector dyn-lh dyn-rh 100 200 300 0.45 144 0))

(setq dyn-l (vector-to-list (vector-round 36 96 dyn-lh))
      dyn-r (vector-to-list (vector-round 36 96 dyn-rh)))

(setq prh-dyn (symbol-divide
               (gen-random 0.11  6 '(6 12 18 6)) nil nil dyn-r)
      plh-dyn (symbol-divide
               (gen-random 0.11  6 '(6 12 18 6)) nil nil dyn-l))


;; score

(def-tonality
  default (activate-tonality (chromatic b& 2))
  perc1 kit-1
  perc2 kit-2
  perc3 kit-3
  perc4 kit-4
)


(def-symbol
  piano-rh pnrh2
  piano-lh pnlh2
  perc1   p1-sym
  perc2   p2-sym
  perc3   p3-sym
  perc4   p4-sym
)

(def-length
  default '(1/16)
)

(def-duration
  default '(1/32)
  perc1  '(1/16)
  perc2  '(1/16)
  perc3  '(1/16)
  perc4  '(1/16)
)
```

Use the hopalong algorithm to generate dynamic values for each hand, with a MIDI value of between 36-96. The dynamics are also divided up so as to correspond with the phrases being played above.

```
(def-zone
; zone-lengths chosen at random are expanded to create repeats
  default (zone-expand rnd-rpt-list
          (mapcar 'make-zone
                  (do-section :all
                              '(symbol-repeat (length x) '(1/16))
                              (get-symbols-of 'piano-rh))))
)

(def-velocity
  piano-rh (symbol-divide
            (gen-random 0.11  6 '(6 12 18 6)) nil nil dyn-r)
  piano-lh  (symbol-divide
             (gen-random 0.11  6 '(6 12 18 6)) nil nil dyn-l)
  perc1  prh-dyn
  perc2  prh-dyn
  perc3  plh-dyn
  perc4  plh-dyn
)

(def-channel
  default 1
  perc1  10
  perc2  10
  perc3  10
  perc4  10
)

(def-tempo 80)

(def-program gm-sound-set
  default acoustic-grand-piano
)

(compile-instrument-p "ccl;output:" "Omphalos2"
 piano-rh
 piano-lh
 perc1
 perc2
 perc3
 perc4
)
```

The zones are repeated according to the values in rnd-rpt-list. The zones will now be as long as the duration of the percussion parts, but the piano parts will be repeated as the corresponding phrases are played through.

```
;;; Omphalos (Piano and 4 percussionsts) section 3


;; functions

(defun gen-invert-chords (inv-pos chords type)
  "inverts chords in all positions"
  (diagnostic2 "gen-invert-chords " inv-pos chords type $cr$)
  (prog (out pat)
    loop
    (cond ((null type) (setq type 12)))
    (cond ((null chords) (return out)))
    (setq pat
          (cond ((neq (car chords) '=)
                 (symbol-scroll
                  (* (car inv-pos) -1)
                  (explode-to-symbols (car (mapcar 'list chords)))))
                (t '(=))))
    (setq out (append out
                      (list (compress2
                             (cond
                              ((eq (car pat) '=) pat)
                              ((eq (car inv-pos) 0) pat)
                              ((eq (car inv-pos) 1)
                               (e-substitute
                                (list
                                 (transpose-one-symbol
                                  (car (last pat)) type))
                                (last pat) pat))
                              ((eq (car inv-pos) 2)
                               (e-substitute
                                (symbol-transpose
                                 type (nthcdr 2 pat))
                                (nthcdr 2 pat) pat))
                              ((eq (car inv-pos) 3)
                               (e-substitute
                                (symbol-transpose
                                 type (nthcdr 1 pat))
                                (nthcdr 1 pat) pat)))))))

    (setq chords (cdr chords))
    (setq inv-pos (cdr inv-pos))
    (go loop)))
```

This function, now incorparated into new versions of Symbolic Composer inverts a chord around a given position. Essentially this is achieved by scrolling the chord by so many places and transposing the scrolled symbols by the type argument (so in a chromatic tonality, type would be 12). Therefore (gen-invert-chords '(0 1 2 3) '(abcd abcd abcd abcd) 12) yields (abcd bcdm cdmn dmno).

```
;; material

' drum-kits

(set-tonality kit-1
              d& 5        ; a   low bongo
              c 5         ; b   high bongo
)

(set-tonality kit-2
              e 5          ; a   low conga
              e& 5         ; b   high bongo
              d 5          ; c   mute conga
)

(set-tonality kit-3
             g& 5         ; a   low timbale
              f 5         ; b   high timbale
)

(set-tonality kit-4
              f 6         ; a   low wood-block
              e 6         ; b   high wood-block
)


; making clusters (as atoms)

(setq a 'abcd
      b 'efgi
      c 'jklo
      d 'pqrv
      e 'kls
      f 'gop
)

(setq a1 '(b b = a) ; w-block
      b1 '(= a b a) ; timbale
      c1 '(b a b b) ; congas
      d1 '(a = a b); bongos
      e1 '(a = b ab)
      f1 '(= ab = ab)
)
```

The third section is constructed along similar lines to the first. On the next page material is generated and arpeggiated, transposed and expanded in the same manner.

```
(setq material (list a b c d e f '=))

(setq ratio-values2 '(1 2 3 3 2 1 4))


(setq gen-all
(prog (out)
  (for i 1 1 4 nil ; enables loop start from 1 rather than 0 (dotimes)
            (setq out
                  (append out
                         (list
                         (gen-symbol-ratio 0.131
                             (change-length :times i ratio-values2)
                             material)))))
                        (return out)))

(setq gen-1
      ; splitting up some chords into arpeggios
      (do-section  (gen-template 0.71 1 3 (length (first gen-all)))
                         '(symbol-melodize x)
                         (mapcar 'list (first gen-all))))

(setq gen-2
      ; plus reversing the direction of some arpeggios
      (do-section  (gen-template 0.71 1 3 (length (second gen-all)))
                   '(reverse x)
                   (do-section  (gen-template 0.71 1 4 (length (second gen-all)))
                         '(symbol-melodize x)
                         (mapcar 'list (second gen-all)))))


(setq gen-3
      ; plus randomizing octave positions of some notes/chords
      (do-section  (gen-template 0.51 1 5 (length (third gen-all)))
                   '(randomize-octaves 0.231 nil -1 0 x)
                   (do-section  (gen-template 0.61 1 3 (length (third gen-all)))
                   '(reverse x)
                   (do-section  (gen-template 0.71 1 4 (length (third gen-all)))
                         '(symbol-melodize x)
                             (mapcar 'list (third gen-all))))))
```

Note differing random seed used in generation of material in this section (0.13 was used in section 1). The same is true for the subsequent processing.

```
(defun compr-mkr (lis low high seed)
(if seed (init-rnd seed))
(cond ((is-pause-symbol (car lis)) '(=))
      (t  (symbol-compress (get-random low high) lis))))

(setq gen-4
      ; plus expanding and shuffling
 (do-section (gen-template 0.91 1 1 (length (fourth gen-all)))
                        '(symbol-shuffle x nil)
      (do-section (gen-template 0.81 1 3 (length (fourth gen-all)))
                        '(compr-mkr x -50 -100 nil)
      (do-section  (gen-template 0.51 1 5 (length (fourth gen-all)))
                  '(randomize-octaves 0.23 nil -2 0 x)
                  (do-section  (gen-template 0.61 1 3 (length (fourth gen-all)))
                  '(reverse x)
                  (do-section  (gen-template 0.71 1 4 (length (fourth gen-all)))
                         '(symbol-melodize x)
                         (mapcar 'list (fourth gen-all)))))))))


(setq all-gen (flatten (mapcar 'eval (reverse '(gen-1 gen-2 gen-3 gen-4)))))

;; rhythmic material

(defun rhy-mkr (lis)
(cond ((is-chord (car lis)) '1/8)
       ((equal lis '(=)) '1/4)
       ((equal (length lis) 3) '(1/32 1/32 1/32))
        (t (build-list '1/16 (length lis)))))

(setq rhy-1 (flatten (mapcar 'rhy-mkr gen-1))
      rhy-2 (flatten (mapcar 'rhy-mkr gen-2))
      rhy-3 (flatten (mapcar 'rhy-mkr gen-3))
      rhy-4 (flatten (mapcar 'rhy-mkr gen-4)))

(setq rhy-all (flatten (mapcar 'eval (reverse '(rhy-1 rhy-2 rhy-3 rhy-4)))))
```

The order of the parts in our variation on the first section is also reversed, from the largest group of material to the smallest - of from the most complex (gen-4 - transposed, shuffled, expanded, arpeggiated) to the simplest (gen-1 - upward arpeggios only).

```
;; using neurons to make associations v clusters and rhythmic groups

(def-neuron match-p1
(in 1 'abcd) a1
(in 1 'kls) e1
(in 1 '=) (pick-random (list '(= = = = = = = =)
          (fill-template
          (pick-symbol ("binary rhythmics:8-bit binary" nil nil a))
          (gen-random nil 8 '(a b)))))
(otherwise '=)
)

(setq p1-sym (run-neuron 'match-p1 (flatten all-gen)))

(def-neuron match-r1
(in 1 'abcd) '(1/32 1/32 1/32 1/32)
(in 1 'kls) '(1/32 1/32 1/32 1/32)
(in 1 '=) '(1/32 1/32 1/32 1/32 1/32 1/32 1/32 1/32)
(otherwise (in 2 0))
)

(setq p1-rhy (run-neuron 'match-r1 (flatten all-gen)
                         rhy-all))

(def-neuron match-p2
(in 1 'efgi) b1
(in 1 'gop) f1
(in 1 '=) (pick-random (list '(= = = = = = = =)
          (fill-template
          (pick-symbol ("binary rhythmics:8-bit binary" nil nil a))
          (gen-random nil 8 '(a b)))))
(otherwise '=)
)

(setq p2-sym (run-neuron 'match-p2 (flatten all-gen)))

(def-neuron match-r2
(in 1 'efgi) '(1/32 1/32 1/32 1/32)
(in 1 'gop) '(1/32 1/32 1/32 1/32)
(in 1 '=) '(1/32 1/32 1/32 1/32 1/32 1/32 1/32 1/32)
(otherwise (in 2 0))
)

(setq p2-rhy (run-neuron 'match-r2 (flatten all-gen)
                         rhy-all))
```

While the use of neurons to create a percussion part based on the association between chords and percussion gestures was used in section 1, we now add some further interest. When a rest occurs in the piano part, p1, p2 and p4 wil play a random fill using templats from SCOM's library. A template is chosen and then filled with (a b). For example, the template (a = = a a = a a) may give the percussive gesture (a = = b a = b a).

```
(def-neuron match-p3
(in 1 '=) 'c
(in 1 'jklo) c1
(in 1 'kls) e1
(otherwise '=)
)

(setq p3-sym (run-neuron 'match-p3 (flatten all-gen)))

(def-neuron match-r3
(in 1 'jklo) '(1/32 1/32 1/32 1/32)
(in 1 'kls) '(1/32 1/32 1/32 1/32)
(otherwise (in 2 0))
)

(setq p3-rhy (run-neuron 'match-r3 (flatten all-gen)
                         rhy-all))
(def-neuron dyn-r3
 (in 1 'c) '120
(otherwise (in 2 0))
)

(setq p3-dyn (run-neuron 'dyn-r3
                         p3-sym
                         (vector-to-list
                          (vector-round 120 44
                                        (gen-noise-white
                                         (length (filter-delete '= p3-sym)))))))


(def-neuron match-p4
(in 1 'pqrv) d1
(in 1 'gop) f1
(in 1 '=) (pick-random (list '(= = = = = = = =)
           (fill-template
           (pick-symbol ("binary rhythmics:8-bit binary" nil nil a))
           (gen-random nil 8 '(a b)))))
(otherwise '=)
)

(setq p4-sym (run-neuron 'match-p4 (flatten all-gen)))
```

```
(def-neuron match-r4
(in 1 'pqrv) '(1/32 1/32 1/32 1/32)
(in 1 'gop) '(1/32 1/32 1/32 1/32)
(in 1 '=) '(1/32 1/32 1/32 1/32 1/32 1/32 1/32 1/32)
(otherwise (in 2 0))
)

(setq p4-rhy (run-neuron 'match-r4 (flatten all-gen)
                         rhy-all))



;; score

(def-tonality
 piano (reverse (activate-tonality
; follows the root of each cluster
        (chromatic c 4) (chromatic e 4)  (chromatic a 4) (chromatic d# 4)))
perc1 kit-1
perc2 kit-2
perc3 kit-3
perc4 kit-4
)

(def-symbol
 piano (gen-invert-chords (gen-random 0.121 (length (flatten all-gen))
                                            '(1 2 3 0))
                                  (flatten all-gen) 12)

 perc1 p4-sym
 perc2 p3-sym
 perc3 p2-sym
 perc4 p1-sym

)

(def-length
 piano rhy-all
 perc1 p4-rhy
 perc2 p3-rhy
 perc3 p2-rhy
 perc4 p1-rhy
)
```

Where chords occur in the final material they are randomly inverted using the previously defined gen-invert-chords function.

```
(def-duration
 piano (change-length :divide 2 rhy-all)
)


(def-zone
 default (mapcar 'make-zone
                 (mapcar 'eval (reverse '(rhy-1 rhy-2 rhy-3 rhy-4))))
)

(def-velocity
 piano (vector-round 44 120 (gen-noise-white (length (get-symbols-of 'piano))))
 perc1 (vector-round 120 44 (gen-noise-white (length (filter-delete '= p4-sym))))
 perc2  p3-dyn
 perc3 (vector-round 120 44 (gen-noise-white (length (filter-delete '= p2-sym))))
 perc4 (vector-round 120 44 (gen-noise-white (length (filter-delete '= p1-sym))))

)

(def-channel
 piano 1
 perc1 10
 perc2 10
 perc3 10
 perc4 10
)

(def-program gm-sound-set
 piano acoustic-grand-piano
)

(def-tempo 80)

(compile-instrument-p "ccl;output:" "Omphalos3"
 piano
 perc1
 perc2
 perc3
 perc4
)
```