Piece d'Orgue

Symbolic Composer Score Files

Nigel Morgan

for the organists of Wakefield Cathedral

Piece d'Orgue

for the organists of Wakefield Cathedral

Nigel Morgan

This work for organ is based on the celebrated Fantasia in G Major BWV 572 by J.S.Bach, often known as the Piece d'Orgue. It is also one of a group of pieces written during the summer and autumn of 2003 that provide the technical expressions for the series of six orchestral concertos belonging to the series of ensemble works called *Instrumentarium Novum*. These include *Le Jardin Sece* and the *Seven Nuptial Blessing* for piano duo.

The code annotations provided here are for the second of the three sections of the composition. In this section the activity of the music across the two manuals and pedal of the baroque organ is mapped using an algorithmic construct loosely based on the timesheet design used by Milton Babbitt in his early Composition for Four Instruments.

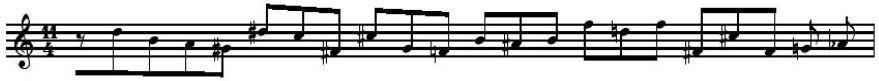
```
I     "---------- - ------- ----- ------- ---- ----- -----"
II    " ----- -- ---- ------------ -- ----- ---- ---------"
III   "-   --- - - - - --       ---  -    -  --  -  ---  -"
```

Timesheet of second movement, showing activity in manuals (I & II) and pedal (III).

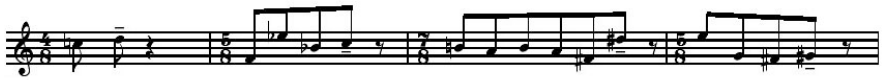
The music itself is generated from a white noise fractal and converted into symbols within a chromatic range of an octave. This symbol stream is then analysed to find possible phrase boundaries and then broken up into a collection of phrases of varying sizes. This phrase collection is randomized to provide a new collection that may or may not contain repetitions and omissions of the original material. The final collection then assumes the function of a nuclear melody, similar to that found in the music of the Balinese Gamelan, where the original pitch material provides a foundation for the variation and development of each instrumental part.

Once the activity scheme and phrase content is established the music is then processed phrase by phrase using a shorthand technique involving specially created I-Functions placed on a Scoresheet.

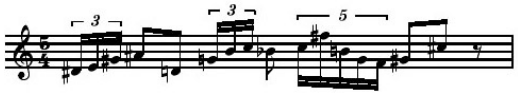
```
((   = j  g  e d  k h b  i d a   g f g  m j m   b i b  c d)
(   ( )
```



```
 (  h  j =)(=) (a k f h  =)   (g e g e b k =) (l c  b d  =)
 (su)      ( ) (pr)           ( )             ( )
```



```
   (d   f c  h   k  m g    d i =) ...
   (pr or)                        ...
```



Extract from phrase collection, I-Functions score-sheet for manual I and the notated result of processing (su = symbol-upwards, pr = play-registers, or = symbol-ornament)

```
;; Piece d'Orgue (Instrumentarium) 30.10.03 (2:3)

;; material

#|

key:

pr - play-registers
bs - symbol-bundle-i
su - symbol-upward
sd - symbol-downward
fl - symbol-floating
vi - verbal-intervention
or - symbol-ornamentation
cc - create-chords

ss - symbol-scale
sc - symbol-compress
sf - symbol-fold
si - symbol-inversion

sk - symbol-skip-i
ct - symbol-cut-i

sh - symbol-harmonize

|#
```

Piece d'Orgue was one of the first pieces to use the I-Functions for the orchestration of a single melodic line. In this piece, each manual of the organ has its own 'scoresheet' (see p.5-6) made up of combinations of two-letter codes which indicate which I-Function to apply to the processing of each list of symbols in the basic phrase collection. The I-Functions themselves are contained in an external file, the two-letter forms being aliases of their more descriptive function names. To briefly explain the I-Functions used in *Piece d'Orgue*:

pr - Transposes notes in a phrase by intervals specified by the composer.
bs - Unlike the standard symbol-bundle, this command adds rests to the resultant phrase, preserving the relationship between symbols and rhythm.
su & sd - Creates ascending and descending phrases. Note that the direction is not completely linear - one might get (a b c d c e f), as opposed to (a b c c d e f).
fl - Reorganises a phrase so that the intervals are smaller.
vi - Elements in an existing phrase can be substituted for elements from a new collection of symbols (for example, derived from a word).
or - Creates ornamentations on certain notes in a phrase.
cc - Create chords from symbols while preserving the phrase length. Unlike bs, the chords generated are not limited to the symbols in the phrase, but can fall outside of the tonal schemes suggested by the phrase.
ss, sc & sf - Not used in *Piece d'Orgue*
si - Inverts the symbols in a phrase around a defined symbol value, e.g (a b c) inverted around c would give (e d c).
sk - This function replaces a number of symbols in a phrase with rests, determined by the length of the phrase.
ct - Cuts a number of symbols from the start, end or at random from a phrase, the number of symbols is also determined by length.
sh - Harmonises the phrase chromatically, adding chords as necessary, but still preserving the phrase length.

```
; material

(setq source (gen-noise-white 512 1.0 0.7))

(setq sym (vector-to-symbol a m source))

(setq f-sym (find-change sym)
      a-sym (find-anacrusis sym)
)
```

The basic symbolic material is generated from 512 samples of white noise, which are then mapped to symbols in the range a-m. For example:
`(g k m k k c f f k g h a c b c j a c a j h f f a k b d m f ...`

Find-change is then used to add rests wherever a duplicate symbol is found, resulting in the following set of symbols:
`(g k m = k c = f k g h a c b c j a c a j h = f a k b d m f ...`

```
(setq c-sym (c-list-rotate 0.1 (create-lists f-sym)))

(symbol-divide (mapcar 'length c-sym) 'setq 'x (flatten c-sym))

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

The symbolic data is then broken into lists wherever a rest is found, using *create-lists:*
`(g k m k) (= c f) (= k g h a c b c j a c a j h f) (= a k b d m f) ...`

Some of the lists are randomly selected to be 'rotated' one step to the left so that each phrase does not always start with a rest (thus (= e i) may become (e i =) and so on.

Symbol-divide is used to break the lists of symbolic data into statements that define each list individually. Initially these statements appear in SCOM's Listener window, and have to be pasted into the code. This results in the 51 lists of symbols, identified as x0 - x50, seen below. These will be organised into the core symbolic data of the piece.

```
(setq x34 '(= f e i))
(setq x35 '(= b m d e k h l d j e c i f))
(setq x36 '(= m k i b k j c g h b e h j m c k c d))
(setq x37 '(b f b e b f e b h d f h c m l f l b k d a e l c e m j i h i j =))
(setq x38 '(= h d k f m c f k l d h f k j e d b l))
(setq x39 '(= i c f l h b f i h l d))
(setq x40 '(a k f h =))
(setq x41 '(l j h e k g j e j h e h e g e l c k =))
(setq x42 '(= c a m l i j m))
(setq x43 '(l i =))
(setq x44 '(= g i b f))
(setq x45 '(= k i c))
(setq x46 '(l c b d =))
(setq x47 '(= k c i a h g j e m i e h a f l h j i c f k d))
(setq x48 '(e =))
(setq x49 '(d b h d b f d h l k i c =))
(setq x50 '(e =))

(setq n-list (gen-random 0.1 51 (list-a-scale 0 51)))

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

51 integers ranging between 0-50 are generated using the *gen-random* function:

```
(19 21 14 40 23 46 25 19 34 7 17 39 30 4 50 41 48 38 34 9 1 22
21 50 12 3 42 1 20 19 30 37 0 7 0 38 10 2 8 1 29 27 41 22 22 8
43 43 50 41 23))
```

These integers are then mapped to the phrases generated above using the *eval-section-integer* function. This results in a phrase collection containing some repeats of the above phrases, while others will not be represented at all. For example:

```
 x19                                          x21     x14
(= j g e d k h b i d a g f g m j m b i b c d) (h j =) (=)
 x40          x23            x46
(a k f h =) (g e g e b k =) (l c b d =) ...
```

```
;; algorithmic distribution of music between manuals and pedals

(setq manual '(I II III))

(setq manuals
(build-array
 ; column 0     1      2    3
       '((I     II     I     II )  ; 0
         (II    III    I     I )   ; 1
         (III   I      II    II)   ; 2
         (I     II     I     II))) ; 3

)

(setq lisn '(0 1 2 3))

(setq output-all
     (mapcar 'remove-duplicates
             (delete 'nil
                     (gen-collect 0.12 51 :list
; 51 is number of sections created from a generation of 512 symbols
                                  '(pick-array
                                    (pick1 nil lisn)
                                    (pick1 nil lisn) 3 manuals
                                    (pick1 nil '(:left
                                                 :right
                                                 :up
                                                 :down
                                                 )))))))

;; instrumentation per section

(instrument-to-string 'I output-all)
;  ---------- - ------- ----- ------- ---- ----- -----


(instrument-to-string 'II output-all)
;  ----- -- ---- ------------ -- ----- ---- ---------
```

For each of the 51 phrases gathered together above the program will decide which manuals on the organ should be played in its interpretation.

Build-array is used to create a 4x4 array with different manual weightings. Since this function was originally implemented as an aid to 12-tone composition, the array can be read from left-to-right, right-to-left, top-to-bottom or bottom-to-top, as well as diagonals.

The manuals that will be employed in each phrase are selected from the above array using the *pick-array* function. This uses the list *lisn* to look up a value in the array *manuals*, for example (0,3). Then a row of three values are selected extending either left, right, up or down from the initial value, so (0,3) :up would give us (I III II). The output for the first few phrases looks something like this:

```
 x19        x21       x14        x40
(iii i i) (ii i ii) (ii i ii) (i ii i)
 x23        x46
(iii ii i) (iii ii i) ...
```

Piece d'Orgue is written using a 'scoresheet', which maps the I-functions (p.1) to the 51 phrases in the composition.

To facilitate the composition of the scoresheet (see next page) a timesheet is created. This uses the 'beat-space' notation, wherein if an instrument is playing it is shown by a '-', otherwise by an empty space. The *instrument-to-string* function is used to examine the selections of manuals (*output-all*) and draw the appropriate timesheets.

```
(instrument-to-string 'III output-all)
; -   --- - - - - --       ---  -    -  --  -  ---  -


; timesheet graphic

#|
 ;
 ;    |   |   |   |   |   |   |   |
I     "---------- - ------- ----- ------- ---- ----- -----"
II    " ----- -- ---- ------------ -- ----- ---- ---------"
III   "-   --- - - - - --       ---  -    -  --  -  ---  -"

|#


(setq tsheet-lengths (mapcar 'length r-sym))
; (22 3 1 5 7 5 10 22 4 17 2 12 13 21 2 19 2 19 4 1 3 11 3 2 2 7 8 3 1 22 13 32 4 17
; 4 19 10 15 8 3 13 12 19 11 11 8 3 3 2 19 7)


; scoresheet

(setq output-all/mx ; edited by hand (I)

'(() (su) () (pr) () () (pr or)
(sh) (sh) (or) () (pr)
() (cc) () (or si pr) ()
(sh) () () () (or ct) (sh) () () ()
(si) (sh) () (sd sk) (bs)
(cc) () (sh) ()
(sd sk) () () () ()
() (pr or) (or pr) (or)
(or) () (or) (or) (or) (or) (su)))
```

This is the resultant timesheet compiled from the instrument-to-string operations carried out above.

This list shows which I-Functions will be applied to each phrase played on the first manual. Therefore, the first phrase will be unchanged, the second will have symbol-upward (su) applied, the fourth play-registers (pr), the seventh both play-registers and symbol-ornamentation (or). Scoresheets for the other manuals follow.

This composition with I-functions was carried out using the above timesheet as a guide so that the composer knows when the manual in question is playing and can make compositional decisions based on the beat-space distribution.

```
 (setq output-all/mz ; edited by hand (II)
'(() (cc) () (su) (su) () (sk su)
(sd sk) (sd) () () (si pr)
(bs) (pr) () (or) ()
(sd sk) (su) () (sh) (or sk) (sh) () () (si)
() (sh) () () ()
(or pr) () (or sc) (or)
(sd sk) () (si) (sh) (sh)
(si) () (or si pr) (or)
(or) (sh) (or) () (or) (or si pr) ()))

(setq output-all/my ; edited by hand (III)
'((sd sk) () () () (sd) (su) (sk sd)
() (su) () () ()
(pr) () () () ()
(sk) () () () () () () () (sd)
(su) () () () (pr)
() () () ()
(su sk) () () () ()
() () (sk si) ()
() () () () (r) (si sc) (su)))

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


(setq I-zone (zone-convertor z-len (instrument-to-string 'I output-all))
      II-zone (zone-convertor z-len (instrument-to-string 'II output-all))
      III-zone (zone-convertor z-len (instrument-to-string 'III output-all)))


(setq I-temp (do-section :all '(calculate-rests x) I-zone)
      II-temp (do-section :all '(calculate-rests x) II-zone)
      III-temp (do-section :all '(calculate-rests x) III-zone))
```

Length data is generated as a series of 1/8 durations grouped according to the length of the phrases in the basic melody, for example:

```
x19        x21            x14  ...
(22)       (3)            (1)  ...
(1/8…1/8) (1/8 1/8 1/8) (1/8) ...
```

From this, zone lengths are derived using *zone-ratio-sc*, so that:

```
 x19    x21   x14  ...
(11/4) (3/8) (1/8) ...
```

The functions *zone-ratio-sc* an *zone-convertor* (see below) are both specially written functions. Their code is included as an appendix to these annotations.

The zones are then checked against the timesheet (see above) using *zone-convertor*. If a manual is not playing then the zone is given a negative value. This step and the two which follow are concerned with formatting the data in such a way as the *symbol-swallow* function can be used to turn any symbols that should not be played into rests (=).

Calculate-rests is a function that creates a template based on whether the zone is silent or not. This template is analogous to the time-sheet above, but is in a format that can be used by do-section (see below) to process only selected phrases. The template III-temp looks something like this (compare with timesheet on p.5): `= x x x = = = x = x = x = x = x = = ...`

```
(setq I-len (do-section I-temp '(lengths-to-rests x)  r-len)
      II-len (do-section II-temp '(lengths-to-rests x)  r-len)
      III-len (do-section III-temp '(lengths-to-rests x) r-len)
)
```

The template created above is used to manipulate the note-length data, turning the individual note-lengths into negative values if the zone they play in is silent. This needs to be done in order for *symbol-swallow* (below) to function properly.

```
(setq I-sym (mapcar (function (lambda (x y) (symbol-swallow x y))) I-len  r-sym)
      II-sym (mapcar (function (lambda (x y) (symbol-swallow x y))) II-len  r-sym)
      III-sym (mapcar (function (lambda (x y) (symbol-swallow x y))) III-len r-sym)
)
```

The symbolic data for each manual comes from the basic phrase collection. The silent phrases are replaced with rests (=) using the *symbol-swallow* function. So III-sym ends up looking like this:
`(= j g e d k h b i d a g f g m j m b i b c d) (= = =) (=) (= = = = =) (g e g e b k =) (l c b d =) ...`
Compare this with the above template – note the phrases that correspond to the 'x's are now filled with rests.

```
(setq I-symx
(do-section
(mtypes-to-template 'sh output-all/mx)
'(symbol-harmonize nil 'mix -7 0 x)
(do-section
(mtypes-to-template 'si output-all/mx)
'(symbol-inversion 'g x)
(do-section
(mtypes-to-template 'cc output-all/mx)
'(chord-creator '(2 4) x)
(do-section
(mtypes-to-template 'sk output-all/mx)
'(symbol-skip-i 3 7 x)
(do-section
(mtypes-to-template 'ct output-all/mx)
'(symbol-cut-i 3 7 x)
(do-section
 (mtypes-to-template 'pr output-all/mx)
  '(play-registers 2 '(-12 12 -7 -5) x)
  (do-section
 (mtypes-to-template 'bs output-all/mx)
 '(symbol-bundle-i 3 x)
  (do-section
 (mtypes-to-template 'su output-all/mx)
 '(symbol-upward x)
  (do-section
```

This code analyses the scoresheet for the first manual (see p 5) and applies the corresponding I-Function to each phrase. This is done by looking at the scoresheet (output-all/mx) and creating a template for each kind of I-function. This template is passed to a *do-section* function that applies the I-function to the necessary phrases. Therefore the phrases that have the I-function 'pr' associated with them have *play-registers 2 '(-12 12 –7 –5)* applied to them. For example, 'pr' applied to phrase x41:
`(l j h e k g j e j h e h e g e l c k =)` becomes:
`(e c h e d -b j e c a e h -d -b e l -f d =)`

Note that the parameters for each of the I-Functions are defined here. Therefore play-registers is defined as solely working with transpositions of an octave, fourth and fifth, and so on.

```
 (mtypes-to-template 'sd output-all/mx)
 '(symbol-downward x)
 (do-section
 (mtypes-to-template 'fl output-all/mx)
 '(symbol-floating x)
 (do-section
 (mtypes-to-template 'vi output-all/mx)
 '(verbal-intervention 'petermorgan x)
(do-section (mtypes-to-template 'or output-all/mx) '(flatten x)
 (do-section
 (mtypes-to-template 'or output-all/mx)
 '(car (symbol-ornamentation 2 3 0.49 20 x '(1/8)))
  I-sym))))))))))))))

(setq I-lenx  ; replaces length-lists when symbol ornamentation occurs . . .
(p-replace-sections (mtypes-to-template 'or output-all/mx)
(do-section (mtypes-to-template 'or output-all/mx) '(flatten x)
(do-section
 (mtypes-to-template 'or output-all/mx)
 '(cdr (symbol-ornamentation 2 3 0.49 20 x '(1/8)))
  I-sym)) I-len))


(setq II-symz
(do-section
(mtypes-to-template 'sh output-all/mz)
'(symbol-harmonize nil 'mix -7 0 x)
(do-section
(mtypes-to-template 'si output-all/mz)
'(symbol-inversion 'g x)
(do-section
(mtypes-to-template 'cc output-all/mz)
'(chord-creator '(2 4) x)
(do-section
(mtypes-to-template 'sk output-all/mz)
'(symbol-skip-i 3 7 x)
(do-section
(mtypes-to-template 'ct output-all/mz)
'(symbol-cut-i 3 7 x)
```

Wherever the I-Function ‘or’ (symbol-ornamentation) is used the number of symbols will be more than the corresponding number of note-lengths. For example, ‘or’ applied to x41 would yield: `((l m l) (j) (h) (e d e) (k) (g h) (j k j) (e) (j) (h) (e) (h g) (e f e) (g) (e) (l) (c) (k) (=))`

This code looks through the scoresheet for instances of 'or' and creates a template. Using this template it creates a corresponding number of extra note-lengths.

```
(do-section
 (mtypes-to-template 'pr output-all/mz)
  '(play-registers 2 '(-12 12 -7 -5) x)
  (do-section
 (mtypes-to-template 'bs output-all/mz)
 '(symbol-bundle-i 4 x)
  (do-section
 (mtypes-to-template 'su output-all/mz)
 '(symbol-upward x)
  (do-section
 (mtypes-to-template 'sd output-all/mz)
 '(symbol-downward x)
 (do-section
 (mtypes-to-template 'fl output-all/mz)
 '(symbol-floating x)
 (do-section
 (mtypes-to-template 'vi output-all/mz)
 '(verbal-intervention  'petermorgan x)
(do-section (mtypes-to-template 'or output-all/mz) '(flatten x)
 (do-section
 (mtypes-to-template 'or output-all/mz)
 '(car (symbol-ornamentation 2 4 0.51 20 x '(1/8)))
  II-sym))))))))))))))

(setq II-lenz
(p-replace-sections (mtypes-to-template 'or output-all/mz)
(do-section (mtypes-to-template 'or output-all/mz) '(flatten x)
(do-section
 (mtypes-to-template 'or output-all/mz)
 '(cdr (symbol-ornamentation 2 4 0.51 20 x '(1/8)))
  II-sym)) II-len))
```

```
(setq III-symy
(do-section
(mtypes-to-template 'sh output-all/my)
'(symbol-harmonize nil 'mix -7 0 x)
(do-section
(mtypes-to-template 'si output-all/my)
'(symbol-inversion 'g x)
(do-section
(mtypes-to-template 'cc output-all/my)
'(chord-creator '(2 4 3) x)
(do-section
(mtypes-to-template 'sk output-all/my)
'(symbol-skip-i 3 7 x)
(do-section
(mtypes-to-template 'ct output-all/my)
'(symbol-cut-i 3 7 x)
(do-section
 (mtypes-to-template 'pr output-all/my)
  '(play-registers 2 '(-12 12 -7 -5) x)
  (do-section
 (mtypes-to-template 'bs output-all/my)
 '(symbol-bundle-i 4 x)
  (do-section
 (mtypes-to-template 'su output-all/my)
 '(symbol-upward x)
  (do-section
 (mtypes-to-template 'sd output-all/my)
 '(symbol-downward x)
 (do-section
 (mtypes-to-template 'fl output-all/my)
 '(symbol-floating x)
 (do-section
 (mtypes-to-template 'vi output-all/my)
 '(verbal-intervention  'petermorgan x)
(do-section (mtypes-to-template 'or output-all/my) '(flatten x)
 (do-section
 (mtypes-to-template 'or output-all/my)
 '(car (symbol-ornamentation 2 3 0.51 20 x '(1/8)))
  III-sym)))))))))))))
```

```
;; score-template

(def-tonality
      I (activate-tonality (chromatic f 6))
      II (activate-tonality (chromatic f 4))
      III (activate-tonality (chromatic f 2))
)

(def-symbol
      I I-symx
      II II-symz
      III  III-symy
)

(def-length
      I I-lenx
      II II-lenz
      III (l-rest-revert r-len)
)

(def-velocity
      I '(72)
      II '(64)
      III '(96)
)

(def-zone
      default z-len
    ; (11/4 3/8 1/8 5/8 7/8 5/8 5/4 11/4 1/2 17/8 1/4 3/2 13/8 21/8 1/4 19/8 1/4 19/8 1/2 1/8 3/8 11/8 3/8 1/4
    ; 1/4 7/8 1/1 3/8 1/8 11/4 13/8 4/1 1/2 17/8 1/2 19/8 5/4 15/8 1/1 3/8 13/8 3/2 19/8 11/8 11/8 1/1 3/8 3/8 1/4 19/8 7/8)
)
```

A tonality for the composition is here defined, as a chromatic scale beginning on f. This means that in the first manual, the symbol ‘a’ will correspond to the note f-6, and so on.

This code declares which symbol and length data sets will be output. Note that the velocity data is fixed, as appropriate for an organ piece.

The zone definition indicates the duration the piece. In this case the zones are as long as the phrases that they correspond to. In some cases zone lengths can be attenuated or lengthened, thus either abbreviating or looping the phrase in question.

```
(def-channel
      I    1
      II   2
      III  3
)

(def-program gm-sound-set
      I     church-organ
      II    church-organ
      III   church-organ
)

(def-tempo 70)

(compile-instrument-p "ccl;output:"  "study-3v"
      I
      II
      III
)
```

Definition of MIDI parameters, such as channel, instrument sound and tempo. *Compile-instrument-p* is then invoked to compile all the above defined data into a MIDI file and play it.

Appendix 1: Phrase output, timesheet and scoresheets, extracted from the main body of the code.

```
((= j g e d k h b i d a g f g m j m b i b c d) (h j =) (=) (a k f h =) (g e g e b k =) (l c b d =) (d f c h k m g d i =) ; 7
(= j g e d k h b i d a g f g m j m b i b c d) (= f e i) (e i g d e g j d i k j l i e f c =) (d =) (= i c f l h b f i h l d) ; 5
(m a c j h c m l f k c i =) (= a c i j l k m b h j e c i f g l f j d f) (e =) (l j h e k g j e j h e h e g e l c k =) (e =) ; 5
(= h d k f m c f k l d h f k j e d b l) (= f e i) (=) (= c f) (c i a l e f c l g f =) (h j =) (e =) (g =) (= a k b d m f) ; 9
(= c a m l i j m) (= c f) (=) (= j g e d k h b i d a g f g m j m b i b c d) (m a c j h c m l f k c i =) ; 5
(b f b e b f e b h d f h c m l f l b k d a e l c e m j i h i j =) (g k m k) (e i g d e g j d i k j l i e f c =) (g k m k) ; 4
(= h d k f m c f k l d h f k j e d b l) (= k e f l h l m j e) (= k g h a c b c j a c a j h f) (= e m b i b d i) (= c f) ; 5
(d c e f c h b k g c b j =) (= j h f a i g e d l b l) (l j h e k g j e j h e h e g e l c k =) (c i a l e f c l g f =) ; 4
(c i a l e f c l g f =) (= e m b i b d i) (l i =) (l i =) (e =) (l j h e k g j e j h e h e g e l c k =) (g e g e b k =)) ; 7

|      |    |    |        |    |   |    |   |
I     "---------- - ------- ----- ------- ---- ----- -----"
II    " ----- -- ---- ------------ -- ----- ---- ---------"
III   "-   --- - - - - --       ---  -    -  --  -  ---  -"

 (setq output-all/mx ; edited by hand (I)
'(() (su) () (pr) () () (pr or)
(sh) (sh) (or) () (pr)
() (cc) () (or si pr) ()
(sh) () () () (or ct) (sh) () () ()
(si) (sh) () (sd sk) (bs)
(cc) () (sh) ()
(sd sk) () () () ()
() (pr or) (or pr) (or)
(or) () (or) (or) (or) (or) (su)))

(setq output-all/mz ; edited by hand (II)
'(() (cc) () (su) (su) () (sk su)
(sd sk) (sd) () () (si pr)
(bs) (pr) () (or) ()
(sd sk) (su) () (sh) (or sk) (sh) () () (si)
() (sh) () () ()
(or pr) () (or sc) (or)
(sd sk) () (si) (sh) (sh)
(si) () (or si pr) (or)
(or) (sh) (or) () (or) (or si pr) ()))
```

```
(setq output-all/my ; edited by hand (III)
'((sd sk) () () () (sd) (su) (sk sd)
() (su) () () ()
(pr) () () () ()
(sk) () () () () () () () (sd)
(su) () () () (pr)
() () () ()
(su sk) () () () ()
() () (sk si) ()
() () () () (r) (si sc) (su)))
```

Appendix 2: Additional Functions used in Piece d'Orgue

```
(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 z-ratio-sc (length-pattern)
  "
(z-ratio-sc '(1/4 3/8 1/8 1/4))
(z-ratio-sc '((1/4 3/8) (1/8 1/4)))
"
  (diagnostic2 "z-ratio-sc" $cr$)
  (do-quietly (get-ratio-sc (z-ratio-cl length-pattern))))
```

```
(defun calculate-rests (lis-of-len)
" creates a template showing incidence of rests in a length or zone list"
  (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)))
```