



Le Jardin Sec

For String Quartet

Symbolic Composer Score Files

Music by Nigel Morgan

Words by Margaret Morgan

Le Jardin Sec

For String Quartet

Nigel Morgan

This score is taken from a sequence of twelve short works for string quartet celebrating The Garden and its relationship with the elements that give it life: sun, water and shade.

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 *Piece d'Orgue* and the *Seven Nuptial Blessings* for piano duo.

The code annotations provided here are for the first of the four movements that make up *Le Jardin Sec*. In this movement a stream of vectors is generated from a white noise fractal and converted into symbols within a chromatic range of an octave.

(0.41130856599193066	-0.434444263621117
(l	b
-0.25312583669438027	0.12285457387042698
d	h
-0.3989770035841502	. . .
b	. . .

White noise vectors and corresponding symbols.

This symbol stream is 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 scheme that may or may not contain repetitions and omissions of the original material.

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

Nuclear melody for the first movement, repetitions have been highlighted.

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 but is never explicitly stated.

```

;; Le Jardin Sec
;; Introduction

;; experiment with scoring unison texture with
;; ornamentation / octave changes

;; functions

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

```

This function is a variant on *eval-section*. *Eval-section* enables you sequence material from predefined lists of items. For example:

```

(setq ax = (a b c)
      bx = (d e))
(eval-section '(a a b b a) 'x 'append)

```

would give:

```

(a b c) (a b c) (d e) (d e) (a b c)

```

This variant allows the lists to be identified numerically rather than by letters, so that one can create more possible lists (rather than just a-z).

```

(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 (-)) nil nil
                                               (reverse (append (e-position '= lisx)
                                                                (list (length lisx))))))
                   nil nil lisx))

```

This function divides a group of symbols into lists wherever a rest occurs, for example:

```

(create-lists '(a b c = d e = f g h i))

```

would give:

```

((a b c)(= d e) (= f g h i))

```

```

(defun gen-ornament-len (lis-of-len len-lis)
  "creates length values for ornaments reading ornament type (i.e size)
lists"
  (do-quietly
   (flatten
    (symbol-divide lis-of-len nil nil
                  (length-repeat-1 lis-of-len len-lis))))

```

This function receives an 'ornament list', which provides details of how a phrase should be ornamented, for example:

```

(gen-ornament-len '(1 2 4) '(1/8 1/8 1/8))

```

gives:

```

(1/8 1/16 1/16 1/32 1/32 1/32 1/32)

```

```
(defun gen-ornament-sym (lis-of-len sym-lis seed)
"creates symbol ornaments reading ornament type lists"
(do-quietly
(if seed (init-rnd seed))
(flatten
(do-section :all '(symbol-shuffle x nil)
(do-section :all (pick-random (list
'(ornament-lower 8 x) '(ornament-higher 8 x )))
(mapcar (function (lambda (x y) (symbol-repeat x y)))
lis-of-len (mapcar 'list sym-lis))))))
```

Similar to *gen-ornament-len*, this takes an 'ornament list' and generates appropriate symbolic output, so:

```
(gen-ornament-sym '(1 2 4) '(a b c) 0.2)
```

would give:

```
(a b c c f d e)
```

```
;; material
```

```
(setq source (gen-noise-white 256 1.0 0.51))
(setq sym (vector-to-symbol a m source))
```

Create 256 samples of white noise and convert it to symbols in the range a – m with *vector-to-symbols*.

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

Both *find-change* and *find-anacrusis* have the effect of replacing repeated symbols with rests. *Find-change* will keep the first symbol and replace subsequent symbols with rests ((a b b b) = (a b = =)), while *find-anacrusis* keeps the last repeated symbol ((a b b b) = (a = = b)). Note that in this movement the variable *a-sym* is not used.

```
(setq c-sym (create-lists f-sym))
```

Divide the symbol stream into lists using *create-lists* (see above).

```
;; produces setq variables from f-sym list
```

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

```
(setq x0 '(l b d h b))
(setq x1 '(= g i h d i))
(setq x2 '(= m k e))
(setq x3 '(= b e c g e l d a l g i d c l f b))
(setq x4 '(= l c f h k c f b i j g))
(setq x5 '(= h a b e b i c j f i k))
(setq x6 '(= h i k a j f l d e i h e))
(setq x7 '(= a l f c j f j c l h l a i m j b a c i g f g))
(setq x8 '(= h g e))
(setq x9 '(= l))
(setq x10 '(= b m b e c b l k l g i))
(setq x11 '(= c e a c b k))
(setq x12 '(= f c m b c l i d g e m h d g d m i j e m i g))
```

Here *symbol-divide* is used to return the contents of each list in *c-sym*. The flag 'setq is used to make the function output setq functions, which is shown below. These phrases, from *x0* to *x19* represent the basic material from which this piece will be constructed.

```
(setq x13 '(= d k j c f))
(setq x14 '(= k j d g h m b i k f j c l b i c j e i c d i d h f e c k b a l i c g f d k d a d k e c))
(setq x15 '(= d h))
(setq x16 '(= e))
(setq x17 '(= g c d i f g h b c j b d h))
(setq x18 '(= f h e m b e b f b f k h l k d f h a g e i l d))
(setq x19 '(= f j b h j i f h e k j d i l c e l d b e m f))
```

Create a sequence of lists drawn from those above ($x0-x19$). Twenty of the above lists are chosen; so there may be some repeats and some of the sections may not be represented at all.

```
(setq r-sym (eval-section-integer (gen-random 0.1 20 (list-a-scale 0 20)) 'x 'list))
; (14 0 2 0 15 3 1 3 0 11 10 16 8 8 3 16 16 19 16 9)
; ((= k j d g h m b i k f j c l b i c . . .) (l b d h b) (= m k e . . .
```

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

Generate length data in 1/8 note lengths to correspond with the symbols in r-sym. So (= e) (= d h) = (1/8 1/8) (1/8 1/8)

```
(init-rnd 0.1517)
```

Set the random seed to 0.1517.

```
(setq r-v1
  (do-section :all '(length-syncopate nil (get-random 1 1)(get-random 3 1) x) r-len)
  r-v2
  (do-section :all '(length-syncopate nil (get-random 1 1)(get-random 3 1) x) r-len)
  r-va
  (do-section :all '(length-syncopate nil (get-random 1 1)(get-random 3 1) x) r-len)
  r-vc
  (do-section :all '(length-syncopate nil (get-random 1 1)(get-random 3 1) x) r-len)
; ((1/8 -1/8 1/8 1/8 -1/8 1/8 1/8 1/8 -1/8 1/8 1/8 1/8 . . .
)
```

Create length data that is unique for each instrument. *Length-syncopate* is used on each list of symbols with a randomised ratio of rests-to-notes each time (hence the random seed being nil). So the basic lengths for the quartet look like this (negative values indicate rests/syncopation):

```
VnI. ((-1/8 -1/8 1/8 1/8 1/8 1/8 -1/8 -1/8 -1/8 -1/8 -1/8 -1/8 ...
VnII. ((-1/8 -1/8 -1/8 -1/8 1/8 -1/8 1/8 1/8 1/8 1/8 -1/8 1/8 ...
Va. ((-1/8 -1/8 -1/8 -1/8 -1/8 -1/8 -1/8 -1/8 -1/8 1/8 1/8 -1/8 ...
Vc. (( 1/8 -1/8 1/8 1/8 1/8 1/8 1/8 1/8 -1/8 1/8 1/8 1/8 ...
```

;; see below for complete beat-space visualization of phrases 1 and 2

```
(setq r-val  
  (mapcar (function (lambda (x)  
                    (gen-random 0.2 x (gen-symbol-ratio nil '(8 6 2) '(1 2 4))))  
          (mapcar 'length r-len)))  
  
; (((2 1 1 4 2 1 2 1 2 1 2 1 1 2 1 2 1 2 2 2 2 2 2 1 1 2 1 2 2 2 2 1 2 1 1 2 2 1 1 2 2 1 2 1)  
; (1 1 1 1 1) (1 1 1 2) (2 2 2 2 1) - shows possible ornamentation type lists
```

This code creates an ornamentation list. For each list of symbols in *r-len* the values 1, 2 and 4 are distributed in a given ratio (8 6 2).

```
(setq r-symv1 (mapcar (function (lambda (x y) (symbol-swallow x y))) r-v1 r-sym)  
  r-symv2 (mapcar (function (lambda (x y) (symbol-swallow x y))) r-v2 r-sym)  
  r-symva (mapcar (function (lambda (x y) (symbol-swallow x y))) r-va r-sym)  
  r-symvc (mapcar (function (lambda (x y) (symbol-swallow x y))) r-vc r-sym)  
; ((= = j d g h m b = k f j = = b i = = = i = = i d = f  
)
```

Create a version of the nuclear melody (*r-sym*) for each instrument. In this instance *symbol-swallow* is used to turn notes corresponding to syncopations (e.g. $-1/8$) into rests. Therefore we get:

```
VnI. ((= = j d g h m b = k f j = = b i = = = i = = i d = f e c k b = l ...  
VnII. ((= = = = g = m b i k = j = = = = = = = = = = d = = = = = b = l ...  
Va. ((= = = = = = = = = k f = = = = i = j = = c = = = h f = = k = a l ...  
Vc. ((= = j d g h m b = k f j = = b i = = = i = = i d = f e c k b = l ...
```

; produces ornament symbol lists to match ornament lengths

```
(setq r-sym-orn1 (mapcar (function (lambda (x y) (gen-ornament-sym x y 0.1))) r-val r-symv1)  
  r-sym-orn2 (mapcar (function (lambda (x y) (gen-ornament-sym x y 0.1))) r-val r-symv2)  
  r-sym-orn3 (mapcar (function (lambda (x y) (gen-ornament-sym x y 0.1))) r-val r-symva)  
  r-sym-orn4 (mapcar (function (lambda (x y) (gen-ornament-sym x y 0.1))) r-val r-symvc)  
; ((= = = j a b c d g f h m l b = = k e f j = = = b h i
```

The symbols for each voice are now ornamented according to the list derived above.

```
(setq r-orn-v1 (mapcar (function (lambda (x y) (gen-ornament-len x y))) r-val r-len))
; ((1/16 1/16 1/8 1/8 1/32 1/32 1/32 1/32 1/16 1/16 1/8
```

```
;; score
```

```
(def-tonality
  default (activate-tonality (chromatic g 4))
)
```

```
(def-symbol
  vn1 r-sym-orn1
  vn2 r-sym-orn2
  va r-sym-orn3
  vc r-sym-orn4
)
```

```
(def-length
  default r-orn-v1
)
```


```
(def-duration
  default (do-section '(= x x = x = x = x x = x x x = x x = x x)
    '(change-length :divide 2 x :ratio) r-orn-v1)
)
```

```
(def-zone
  default (zone-ratio-sc (length-of vn1))
  ; (11/2 5/8 1/2 5/8 3/8 17/8 3/4 17/8 5/8 7/8 3/2 1/4 1/2 1/2 17/8 1/4 1/4 23/8 1/4 1/4)
)
```

```
(def-velocity
  default '((45) (80) (76) (100) (76) (46) (80) (55)(96)
    (100) (64) (40) (50) (60) (76) (100) (56) (45) (64) (96))
)
```

```
(def-channel
  vn1 1
  vn2 2
  va 3
  vc 4
)
```

The original (non-syncopated) length data is ornamented and used to create the final note-lengths for all the voices. The syncopations will still be present, since they have previously been turned into rest symbols by *symbol-swallow*. Therefore, the first violin part looks like this:

Symbols ((= = = j a b c d g f h = ...
 Lengths ((1/16 1/16 1/8 1/8 1/32 1/32 1/32 1/32 1/16 1/16 1/8 1/16 ...
 Notated  ...

Create variations in the duration of the notes by dividing the duration by two according to a template. Because length is the same, but the duration halved we get *staccato* notes (for example, see bar 25 of the score).

```
(def-program gm-sound-set
  vn1 violin
  vn2 violin
  va viola
  vc cello
)

(def-instrument-controller gm-controllers
  (vn1 panning '((110)))
  (vn2 panning '((86)))
  (va panning '((54)))
  (vc panning '((20)))
)

(def-tempo 75)

(compile-instrument-p "ccl;output:" "garden-1e"
  vn1
  vn2
  va
  vc
)
```

```
#|
```

```
;; possible tonality scheme
```

```
(transpose-tonality 7  
(patterns-to-scales (do-section :all '(delete '= x) r-sym)))
```

```
(g 4 g# 4 a 4 a# 4 b 4 c 5 c# 5 d 5 d# 5 e 5 f 5 f# 5 g 5)  
(g# 4 a# 4 d 5 f# 5)  
(b 4 f 5 g 5)  
(g# 4 a# 4 d 5 f# 5)  
(a# 4 d 5)  
(g 4 g# 4 a 4 a# 4 b 4 c 5 c# 5 d# 5 f# 5)  
(a# 4 c# 5 d 5 d# 5)  
(g 4 g# 4 a 4 a# 4 b 4 c 5 c# 5 d# 5 f# 5)  
(g# 4 a# 4 d 5 f# 5)  
(g 4 g# 4 a 4 b 4 f 5)  
(g# 4 a 4 b 4 c# 5 d# 5 f 5 f# 5 g 5)  
(b 4)  
(b 4 c# 5 d 5)  
(b 4 c# 5 d 5)  
(g 4 g# 4 a 4 a# 4 b 4 c 5 c# 5 d# 5 f# 5)  
(b 4)  
(b 4)  
(g# 4 a 4 a# 4 b 4 c 5 d 5 d# 5 e 5 f 5 f# 5 g 5)  
(b 4)  
(f# 5))
```

This is a conversion of the 20 patterns used in the above movement into scales, so that the tonalities derived from them can be used in the later movements.

```
(length-to-string (first r-v1))  
(length-to-string (first r-v2))  
(length-to-string (first r-va))  
(length-to-string (first r-vc))
```

```
v1 ----  
v2 - ---- -  
va -- - - - -  
vc - - - - - - - - - -
```

Below is a visualisation of the first phrase at a resolution of 1/8 notes. The second phrase is represented on the next page. Compare with bars 1-11 of the full score.

```
(length-to-string (second r-v1))  
(length-to-string (second r-v2))  
(length-to-string (second r-va))  
(length-to-string (second r-vc))
```

```
v1 - -  
v2 -  
va - -  
vc
```

```
|#
```