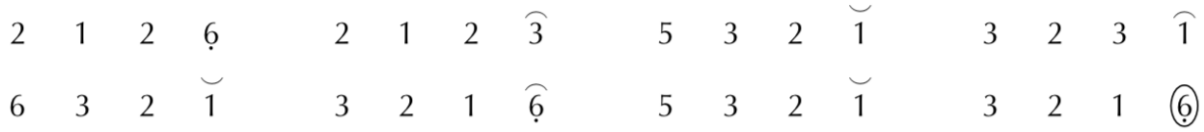
Irama Mabuk can be roughly translated from Javanese as drunken rhythm or drunken tempo. But the word *irama* does not translate directly to either rhythm or tempo. In fact, *irama* is not just a foreign word, but a foreign concept. It is related to rhythm and tempo in *karawitan*, the traditional gamelan music of central Java, but as Richard Pickvance explains in *A Gamelan Manual*, "the most important sense of the word *irama* is a more specialized one, referring to the system that allows the timescale of the melodic material to be expanded and contracted in a controlled manner." Additionally, Pickvance states that "the *irama* system also relates the basic tempo to the patterns played by the decorating instruments."[1] To better understand Pickvance's explanation, it is necessary to know something about *karawitan*.

The traditional music of central Java is guided by a basic melodic material called the *balungan*, derived from the Javanese word for skeleton, and in fact, the *balungan* is the skeleton of the composition. Today, a *balungan* is notated in cipher notation. Numbers indicate notes within one of the two scales - *pelog* or *slendro* - used in this musical tradition. It is similar to the Western idea of scale degree. If a Western musician were given the numbers 5-3-2-1 and told to play them in the key of C major, the musician would play G-E-D-C. However, in a gamelan, each instrument interprets this series of numbers differently, reinforcing that instrument's role within the music.

In a central Javanese gamelan there are roughly 3 groups of instruments - balungan instruments, punctuating instruments, and elaborating instruments. Balungan instruments play the notes of the *balungan* only, largely as written. Punctuating instruments play structurally important notes of the *balungan*. Elaborating instruments create decorated patterns based on the *balungan*. It is this last group of instruments that requires mastery. While *karawitan* cannot be described as improvised music, a player on an elaborating instrument does have significant discretion to determine how they interpret the *balungan* within stylistic standards.

The *balungan* for a section of *Ladrang Asmaradana* is given in cypher notation below.



The upward-facing and downward-facing arches indicate where punctuating instruments play. The circled number indicates that a *gong ageng* plays - *karawitan* is colotomic: it uses cycles of phrases, with the beginning and ending of each phrase articulated by a gong. The *balungan* above could be repeated several times before moving on to the next section.

Balungan instruments like the *saron* would play the *balungan* as written. The *saron panerus* - a small-sized *saron* an octave higher than the main *saron* - would play two notes for each note given, i.e., 22 11 22 66, etc. Elaborating instruments would each interpret this *balungan* differently. For instance, the *gender* might interpret the first four notes as follows:

```
5 6   5 1   5 6   1 6
1 5   6 5   6 1   2 6
```

Gender is played with a mallet in each hand. The upper numbers are for the right hand; the lower ones for the left.[2]

---

[1] Pickvance, Richard. *A Gamelan Manual: A Player's Guide to the Central Javanese Gamelan*. Jaman Mas Books, 2005.

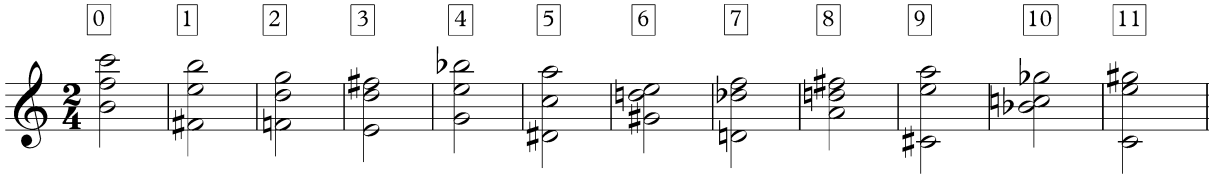
[2] I am indebted to Pickvance's examples of gender elaboration in my explanation here. More detail can be found in his book.

This elaboration would only suffice in Irama I. There are 6 levels of irama. The *balungan* in Irama II moves at roughly half or two-thirds the speed of Irama I. Irama III moves roughly half of Irama II. While it is very uncommon to use a large range of *irama* in a piece, the possible ratio between the slowest and fastest *irama* speeds is approximately 10:1. However, elaborating instruments maintain roughly the same playing speed as the *irama* changes, so that the ratio of elaborating instrument notes per *balungan* note increases as the *irama* increases. Hence, with elaborating instruments, patterns must change with the *irama* because more notes are needed to fill the space given. In Irama IV, for instance, the following is a possible *gender* (G) pattern for just the first 4 notes of the balungan (B):

```
B: 2
G: .5.6.5.3 .1.6...3 .1.6...3 .6.5.6.3
   ..12612. 6121612. 6121612. 6.653.3.

B: 1
G: .6.5.6.1 .6.1.6.3 .6.5.6.1 .6.5.6.1
   6.56356. 2.12612. 6.56356. 1.621621

B: 2
G: ...5.6.1 ...6...1 ...6...1 .5.6.1.6
   .23.3.3. 3212612. 3212612. 3212.212

B: 6
G: .5.3.5.3 .5.6.5.1 .5.6.5.1 .5.6.1.6
   .1.6.5.6 .1.5.6.3 ...5.3.5 .6.16216
```

Karawitan is not just a system of pitches and tempi, it is an ecology. Elaborating instruments are quiet: in faster *irama* their sound is covered by the *balungan* instruments. But as the *irama* increases and the *balungan* slows, there is more acoustical space between the notes played by louder instruments - space that gets taken up by elaborating instruments. Hence, a slowing *irama* reveals previously hidden instrumental sounds, and these instruments reveal another dimension of the *balungan*.

It is this ecological scenario, negotiated through *irama*, that is fundamental to my *Irama Mabuk*. My replacement for a *balungan* is a series of 12 chords.[3]

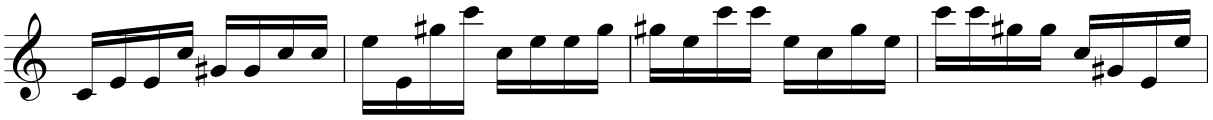


Each is a trichord. Some reference chords with tonal functions. For example, chord 4 is a first inversion diminished triad, chord 6 suggests a first inversion dominant seventh chord, and chord 8 is a second inversion major triad. Other trichords are built from atonal collections. Chord 0 is a (016) collection, chord 1 is a (027) collection, and chord 10 is a (026) collection.

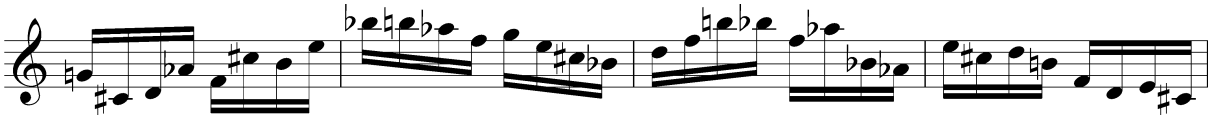
[3] The use of 0 rather than 1 as the base for numbering the chords and irama levels is due to the fact that the programming language used in the development of this work utilizes zero-based indexing, meaning the first value in a list is retrieved by requesting the 0th element.

But looks can be deceiving. These chords represent a *balungan* at *irama* 0, and at this level do not necessarily present all permitted pitches. Through the other 5 irama levels, the harmonic potential of each chord is revealed. As more notes are added, the chord changes into a melody.

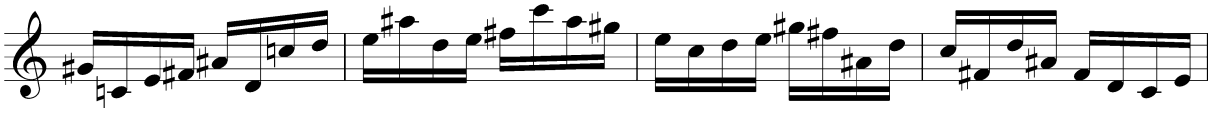
Some chords add no new pitches. Chord 11 is an example. Although notes from different registers are incorporated, only pitches from the original trichord are used.



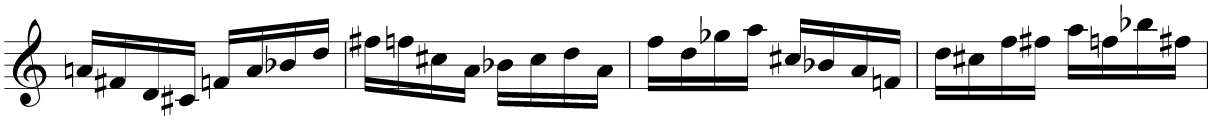
Chord 4, however, expands from the initial E-G-Bb to E-G-Bb-C#-B-D-F-Ab. Essentially, this is the combination of two fully diminished seventh chords.



Some harmonic expansion reveals that presumptions about the nature of the initial trichord are incorrect. At Irama 0, chord 6 suggests a dominant seventh chord, but in Irama 5 it is clear that the harmonic content is a whole tone scale.



Chord 8 is a major triad at Irama 0, but evolves into an octatonic scale by Irama 5.



Realizations of different *irama* levels are fixed. Hence, the 12 chords and their 6 *irama* level projections form all the material of the piece. Since the work cycles through the 12 chords continually, the main component of the compositional process is selecting the *irama* for each chord. In *karawitan*, shifting between *irama* is a gradual process. It occurs during the last repetition of a cycle. Generally, *irama* shifts up or down by one level, e.g., from Irama II into Irama I. In my work, changing to any *irama* level can happen with each new chord. This is where drunkenness enters.

The drunkenness I'm referencing in my work's title is statistical drunkenness. A drunken walk, or a random walk, is a stochastic process. It is found in fields as diverse as finance, behavioral economics, and physic. It is the same type of randomness that Burton G. Malkiel addressed in his classic book on stock market investing, *A Random Walk Down Wall Street*. Specifically, I use a Bernoulli random walk, in which a series of trials that can have only 2 outcomes is carried out. A coin toss is a good example of a Bernoulli trial. In the case of *Irama Mabuk*, getting heads would increase the *irama* level while getting tails would decrease it. The collection of these Bernoulli trials forms a Bernoulli random walk.

I used a recursive function to generate this. Defining a function **`bern-drunk`** that takes a minimum value, maximum value, desired length of results, and a starting value, I recursively call the function, randomly choose a value that is one greater or less than its predecessor with each call, staying within the boundaries defined by the minimum and maximum values.


```
(defun bern-drunk (min-val max-val length start &optional lor)
  (cond ((equal lor nil)
         (bern-drunk min-val max-val length start (list start)))
        ((= (length lor) length) lor)
        ((= min-val (first (last lor)))
         (bern-drunk min-val max-val length start (append lor (list (+ min-val 1)))))
        ((= max-val (first (last lor)))
         (bern-drunk min-val max-val length start (append lor (list (- max-val 1)))))
        (t
         (bern-drunk min-val max-val length start (append lor (list (+ (first (last lor))
          (rnd-pick '(-1 1))))))))
```

Code Explanation

```
(defun bern-drunk (min-val max-val length start &optional lor)
```

We define a function, that is, a small algorithmic process, that takes the arguments `min-val`, `max-val`, `length`, `start`, and optionally `lor`, which stands for list of results. This list of results will be collected over successive calls to the function, and it will be returned once the length of `lor` is equal to the `length` argument.

```
(cond ((equal lor nil)
        (bern-drunk min-val max-val length start (list start)))
```

`Cond` allows for a list of conditions to be given. The program checks these conditions in the order written. First, it asks if the argument `lor` is equal to nil, that is, it checks to see if there is a value for `lor`. If false - and it will be false at the first run of the function since no value for `lor` will be supplied - then, it calls the function again using all of the arguments already given and creates a list with the `start` argument to serve as the `lor` argument.

```
((= (length lor) length) lor)
```

On this subsequent call, there will be a value for `lor`, so the first condition will be false and the the program will test the next condition. This checks whether the length of `lor` is equal to the value given for the argument `length`. This is the condition that will terminate the recursive process. If this is true, then the function returns `lor`, the drunken walk.

```
((= min-val (first (last lor)))
  (bern-drunk min-val max-val length start (append lor (list (+ min-val 1)))))
((= max-val (first (last lor)))
  (bern-drunk min-val max-val length start (append lor (list (- max-val 1)))))
```

The next two conditions check if the last value in `lor` is equal to the minimum value or maximum value established. If so, the result of (`min-val` + 1) or (`max-val` - 1), respectively, is appended to the end of `lor` and the function is called again.

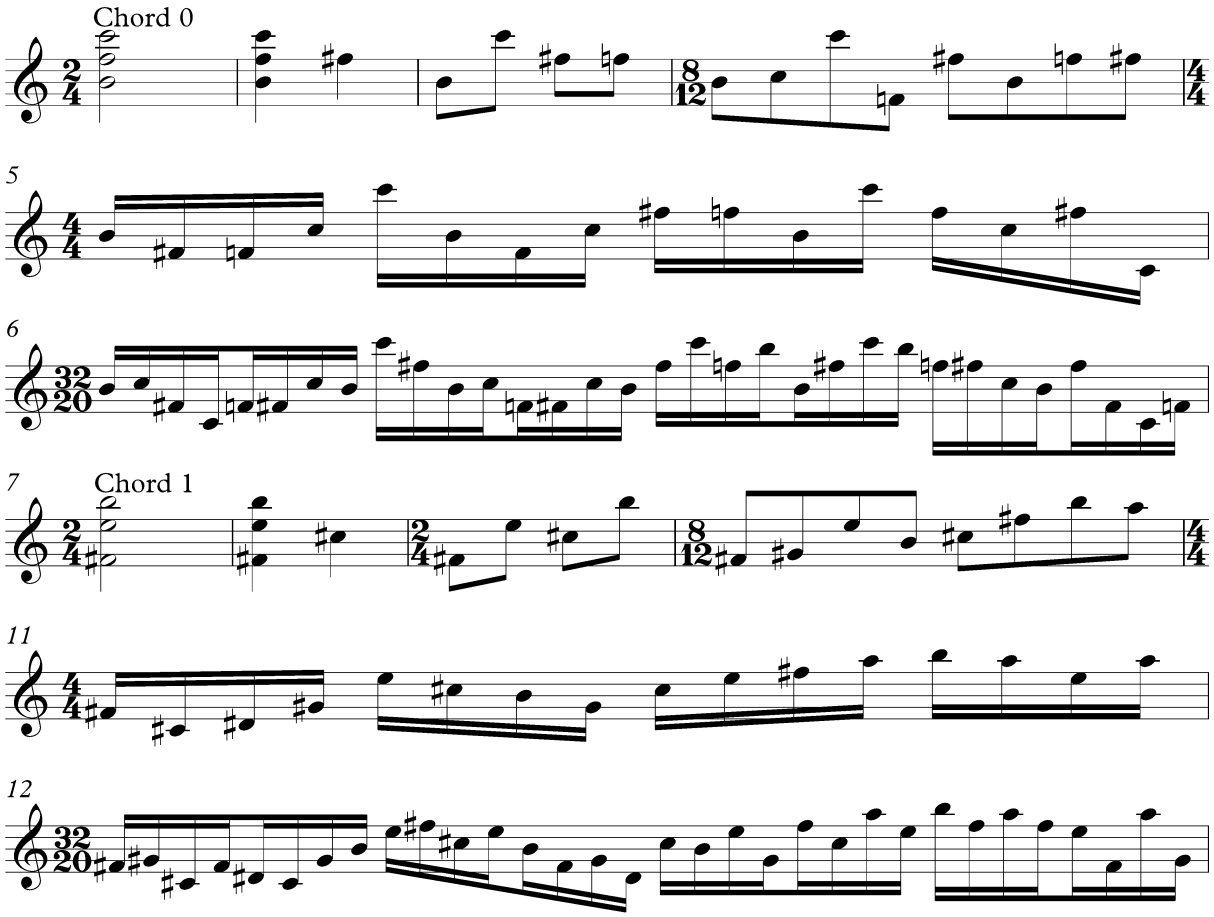
```
(t
  (bern-drunk min-val max-val length start (append lor (list (+ (first (last lor))
   (rnd-pick '(-1 1)))))))
```

If all other conditions are false, the function is called again and 1 is added or subtracted from the last value in `lor`.

An example with minimum of 0, maximum of 5, length of 10, and start value of 2 is given below. Note the result starts with 2, is 10 values in length, and shifts by +1/-1 in each successive value.

```
(bern-drunk 0 5 10 2) => '(2 3 2 1 0 1 0 1 2 3)
```

The resulting values can then be used to access *irama* levels for successive chords. In order to do that, I created a list of lists as a kind of database, each sublist represents a measure of music containing a chord at a particular irama level. The lists are written in OMN, the notational scripting language for the programming environment OpusModus. Below are the first 2 chords with all *irama* levels. Following that are these same measures in OMN.




```
'((h b4f5c6) ;Chord 0, Irama 0
  (q b4f5c6 fs5)
  (e b4f5c6 c6 fs5 f5)
  (3q b4f5c6 c5 c6 f4 fs5 b4 f5 fs5)
  (s b4f5c6 fs4 f4 c5 c6 b4 f4 c5 fs5 f5 b4 c6 f5 c5 fs5 c4)
  (5q b4f5c6 c5 fs4 c4 f4 fs4 c5 b4 c6 fs5 b4 c5 f4 fs4 c5 b4 fs5 c6 f5 b5 b4 fs5
c6
   b5 f5 fs5 c5 b4 fs5 fs4 c4 f4)
  (h fs4e5b5) ;Chord 1, Irama 0
  (q fs4e5b5 cs5)
  (e fs4e5b5 e5 cs5 b5)
  (3q fs4e5b5 gs4 e5 b4 cs5 fs5 b5 a5) ;Chord 1, Irama 3
  (s fs4e5b5 cs4 eb4 gs4 e5 cs5 b4 gs4 cs5 e5 fs5 a5 b5 a5 e5 a5)
  (5q fs4e5b5 gs4 cs4 fs4 eb4 cs4 gs4 b4 e5 fs5 cs5 e5 b4 fs4 gs4 eb4 cs5 b4 e5
gs4
   fs5 cs5 a5 e5 b5 fs5 a5 fs5 e5 fs4 a5 gs4))
```

Each sublist begins with a rhythmic value - h indicates half note, q indicates quarter note, e is used for eighth, s for sixteenth, 3q for triple eighth, 5q for sixteenth quintuplet. Notes are expressed by pitch and register. For example, c4 is middle c. Notes without space between them indicate chords. Otherwise, notes are individual pitches.

The complete list includes 72 sublists, each chord and irama level available. To access these in a logical way, I created a function called **get-passage** that takes the chord number and irama number and references the list of lists - named score in the function - to return the correct nth value of the list.

```
(defun get-passage (chord irama &optional (the-score score))
    (nth (+ (* 6 chord) irama) the-score))
```

For example, given the arguments 1 and 3, **get-passage** returns Irama 3 for Chord 1. This is measure 10 in the traditional notation given above. The sublist retrieved is also commented in the list of lists and shown below as an outcome.

```
(get-passage 1 3) => '(3q fs4e5b5 gs4 e5 b4 cs5 fs5 b5 a5)
```

Using **get-passage** it is straightforward to iteratively generate musical passages. The first 36 measures of *Irama Mabuk* are generated with the following code.

```
(loop for chord in (gen-repeat 3 '(0 1 2 3 4 5 6 7 8 9 10 11))
    collecting
     (get-passage chord (rnd-pick '(0 1))))
```

A list of 36 values that cycles through the numbers 0 to 11 is iteratively looped through. Each value from this list is used as the chord argument in a call to **get-passage**. The *irama* level is randomly chosen: it can be either 0 or 1. The resulting music is a slightly drunken cycling through the *balungan* chords. These 36 measures open the work to establish the *balungan* chord series.

The main body of the work begins only in m. 111. This component of the work is determined by 10 cycles of the balungan. Shifts in irama level are determined by a process I called streaming. Five behaviors for the irama changes are established, each a list of 120 values. These are named as follows:

1. irama-null, e.g., (0 0 0 0 0 …)

2. irama-low-random, e.g., (0 1 1 0 0 …)
3. irama-wave, e.g., (0 1 2 3 4 5 4 3 2 1 0 1 …)
4. irama-random, e.g., (2 1 0 5 3 4 0 …)
5. irama-drunk, e.g., (2 3 4 5 4 3 4 3 2 1 2 1 0 1 0 …)

In streaming, a start value is given. The first value of each of the 5 lists is evaluated. Any containing the start value are possible streams to switch to. The computer randomly picks one. The first values of each list are removed, and the new first value of the selected stream is recorded. This value serves like the original start value. Any streams containing it as their first value become streams that the computer can switch to. The process is continued until the lists are exhausted. A path of 120 values results, and these are mapped to *irama* levels for the 10 cycles of chords.

The result is less random that the process may seem. While in some situations there are 2 or 3 streams to select from, in other cases, there is only 1. In the example below, such a scenario occurs already with the third trial. Here, irama-wave's value is rarely matched by another stream for another 4 trials. Only irama-drunk offers an alternative path in trial 5. In trials 7-10, only irama-wave and irama-drunk have the same values. Regardless of which is chosen, the outcome will be the same for this stretch. Hence, if irama-wave is chosen in trial 3, it will likely determine the outcome values until trial 10. While not a drunken walk, this process does create unpredictable behavior in *irama* changes for the body of the work.

```
Trial         1 2 3 4 5 6 7 8 9 t e …
null:        (0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
low-random:  (0 1 1 0 0 1 0 1 1 1 0 1 0 0 0)
wave:        (0 1 2 3 4 5 4 3 2 1 0 1 2 3 4)
random:      (2 1 0 5 3 4 0 1 0 5 4 5 1 3 3)
drunk:       (2 3 4 5 4 3 4 3 2 1 2 1 0 1 0)
```

Between the opening 36 measures and the main body of the work, starting at m. 111, there is the first of a number of drunken inserts. Using the last 13 measures of the open (m. 24-36) as series, an algorithm generates a collection of subsets of this series. This algorithm has two components. First, a list of starting points is generated. Then, a length of 3, 4, or 5 measures to extracted using the starting point is randomly chosen. The function `gen-subseries` first generates a length of starting points using the `gen-start-points` function as a helper function as well as a series of sub-length values. Then, it iterates through these two lists using them to generate a list that is itself iterated through to extract the desired measures from the series. For example, with a starting point of 2 and a length of 3, the list '(2 3 4) is generated and iterated through to collect the sublists at indices 2, 3, and 4 in the series.

```
(defun gen-start-points (start len list-len &optional losp)
  (let* ((shift (rnd-pick '(-1 -2 0 1 2)))
         (new-start-point (mod (+ start shift) list-len)))
  (cond ((= len 0) losp)
        (t (gen-start-points new-start-point (- len 1) list-len
            (append losp (list new-start-point)))))))

(defun gen-subseries (start len series)
  (let* ((start-points (gen-start-points start len (length series)))
         (sub-lengths (rnd-repeat len '(3 4 5))))
    (loop for x in start-points
      for y in sub-lengths collecting
       (loop for z in (gen-integer-seq (list (list x (+ x y)))) collecting
         (nth (mod z (length series)) series)))))
```

Code Explanation

`Gen-start-points` is a helper function. It gets called within the `get-subseries` function in order to enable that function's algorithm.

```
(defun gen-start-points (start len list-len &optional losp)
```

Define a function called `gen-start-points`, which accepts the arguments `start`, `len` (for length), `list-len` (for list length), and optionally `losp` (list of starting points). All arguments, except `losp`, are integers.

```
(let* ((shift (rnd-pick '(-1 -2 0 1 2)))
       (new-start-point (mod (+ start shift) list-len)))
```

Define two variables called `shift` and `new-start-point`. For `shift`, randomly select from the values -1, -2, 0, 1, 2. To generate `new-start-point` add the values for `shift` and `start`. Then, apply a modulus of the list length to that. For instance, if the list length is 10 and the sum of `shift` and `start` yield 12, then the `new-start-point value` will be 2.

```
(cond ((= len 0) losp)
       (t (gen-start-points new-start-point (- len 1) list-len
           (append losp (list new-start-point)))))))
```

This function is also recursive. But different from `bern-drunk`, in `gen-start-points` the `len` value is decreased by 1 with each recursive call. Hence, the conditional check to see if `len` is equal to 0. If so, the function returns `losp` and terminates the process. Otherwise, `gen-start-points` is called again, now with `new-start-point` replacing the originally given `start` value, `len` reduced by 1, and the `new-start-point` value appended to the end of `losp`.

```
(defun gen-subseries (start len series)
  (let* ((start-points (gen-start-points start len (length series)))
         (sub-lengths (rnd-repeat len '(3 4 5))))
```

Within the `gen-subseries` function, the `gen-start-points` function is used to create the first variable called `start-points`. `Gen-subseries` takes 3 arguments: `start`, `len`, and `series`. `Start` and `len` must be integers; `series` must be a list, or a list of lists, for example, measures 24-36 in OMN. Note that the length of the `series` is used as the `list-len` argument in `gen-start-points`. Lengths for each sub-length are needed. Note that using len for both `gen-start-points` and `rnd-repeat` ensures that the resulting lists will be the same length. For every starting point, there will also be a value for the number of measures to extract from that starting point. Note that a `sub-length` of 3 will generate a 4-measure result: the starting measure and 3 additional ones.

```
(loop for x in start-points
      for y in sub-lengths collecting
       (loop for z in (gen-integer-seq (list (list x (+ x y)))) collecting
         (nth (mod z (length series)) series)))))
```

The macro `loop` is used for iteration, most commonly encountered when a list needs to be read through and acted upon. I use a nested loop here. There are 3 variables called `x`, `y`, and `z`. `X` represents the starting points, `y` represents the sub-lengths. These are iteratively called and used to create a list. The `gen-integer-seq` takes a list of 2 values and generates an integer series with them. For instance, `(gen-integer-seq '((3 7)))` will return `'(3 4 5 6 7)`. The resulting list is then iterated through, each value being represented by the variable `z`, extracting the nth value within `series` and yield the sub-series desired.

This technique is applied in mm. 223-242 to the *irama* level 4 pitches of chord 9 as well as to chord selection and *irama* level 3 pitches in mm. 228-405. At other locations in which the chord gets stuck - mm. 158-182 and mm. 247-280 - a random selection of the *irama* level is being applied to the given chord. The process using the `gen-subseries` function is not used in these sections.

A final word on the use of meter in the work. Meter is commonly used in music to state the number of beats (numerator) in a measure as well as the value of the beat (denominator). A change in meter, however, can also articulate a tempo change. Shifting from 5/4 to 5/8 is actually the same as keeping the meter at 5/4 and doubling the tempo, e.g., from 60BPM to 120BPM. In my work, I needed a way to quickly shift between tempos at each measure. I chose to use meter for this.

The denominator of a meter states the number of beats that fit in the time of a whole note. 4 quarter notes, 8 eight note, 16 sixteenth notes, etc. Using this, denominators for triplet eighths (12) or quintuplet sixteenths (20) can be deduced. This approach allows for numerator values that are not simple multiples of 3 or 5. For instance, 8/12 fits 8 notes at the speed of triplet eighth notes within a measure. Because groupings of triplets do not need to be by threes and groupings of quintuplets do not need to be by fives, the sense that tempo has changed rather than a different subdivision of the beat has occurred is reinforced.

With *irama* tempo changes expressed through meter, tempo can be used for gradual changes in speed. The base tempo for the work is 144BPM. The tempos within the work also reinforce ratios expressed within the meter changes. Below is a table of the tempo ratios. When simplified, it is clear that the ratios are those between common subdivisions of a beat.

Tempo Ratio	Simplified Ratio
144:96	3:2
144:120	6:5
120:96	5:4
128:96	4:3