# 9
# *Etudes*, Op. 3: Iteration, Rows and Sets

In this chapter we work through examples that use iteration to implement some basic twelve-tone and set theory functions.

## Op. 3, No. 1: Converting MIDI Key Numbers to Pitch Classes

In the first exercise we implement `list->pcs`, a function that converts a list of MIDI key numbers into pitch classes. A pitch class can be thought of as the “remainder” of a key number after all octave information has been “subtracted out”. Since there are only 12 divisions per octave in the standard scale there are only 12 pitch classes. We will represent pitch classes using the integers 0-11. For example, all C key numbers will be pitch class 0, all F-sharps and G-flats are pitch class 6, and so on.

The Lisp function `mod` can be used to convert a key number into a pitch class. The `mod` function returns the absolute value of the remainder from the division of two integers. We can use `loop` to see the effects of `mod` on a range of numbers:

***Interaction 9-1.*** *Using loop to collect numbers mod 12.*

```
cm> (loop for i from 60 to 72 collect (mod i 12))
(0 1 2 3 4 5 6 7 8 9 10 11 0)
cm> (loop for k from 0 downto -12 collect (mod k 12))
(0 1 2 3 4 5 6 7 8 9 10 11 0)
cm>
```

Given the `mod` mod operator our first exercise is easy to implement. We split our programming task into two small functions. The first function uses `mod` to convert a single key number into a pitch class. The second function converts a list of key numbers into a list of pitch classes by mapping the first function over each element in the list.

***Example 9-1.*** *Converting key numbers to pitch classes.*

```
(define (keynum->pc k)
  (mod k 12))

(define (list->pcs knums)
  (loop for k in knums collect (keynum->pc k)))
```

`list->pcs` uses loop to iterate a variable *k* over every key number in a list of key numbers input into the function. The `collect` clause collects the results of `keynum->pc` into a new list and returns that list as the value of the loop. Since the loop expression is the last expression in the body of the function definition, the value returned by loop becomes the value of the function.

***Interaction 9-2.*** *Pitch classes from measure 15 of Berg's Violinkonzert.*

```
cm> (define albans-notes '(55 58 62 66 69 72 76 80 83 85 87 89))

cm> (list->pcs albans-notes)
(7 10 2 6 9 0 4 8 11 1 3 5)
cm>
```

# Op. 3, No. 2: Normalizing Pitch Classes

Note that in Interaction 9-2 the series of pitch classes returned by `list->pcs` is close to, but not the same as, a twelve-tone row. To form a twelve-tone row we would need to convert the list of pitch classes into a list of intervals, where the first interval in the row is 0, the “root” for row transpositions. We will take this opportunity to define a new function that normalizes a list of pitch classes or key numbers into a list of zero-based intervals:

***Example 9-2.*** *Normalizing a list of pitch classes or key numbers.*

```
(define (normalize-pcs knums)
  (loop with root = (first knums)
        for k in knums
        collect (keynum->pc (- k root))))
```

In this function a loop “normalizes” the input list by subtracting the first number in the list from all the numbers in the list. Recall that the `with` clause is used to initialize, rather than step, a looping variable. The expression `with root = (first knums)` directs loop to bind the variable *root* to the first element in the input list. The key feature to remember about a `with` clause is that it happens just one time, immediately before the iteration starts. (A `for` clause, on the other hand, sets variables each time through the loop.)

***Interaction 9-3.*** *Normalizing pitch classes.*

```
cm> (define albans-notes
      '(55 58 62 66 69 72 76 80 83 85 87 89))

cm> (normalize-pcs (list->pcs albans-notes))
(0 3 7 11 2 5 9 1 4 6 8 10)
cm>
```

## Op. 3, No. 3: Matrix Operations

In Example 9-3 we implement four functions that perform the core manipulations for twelve-tone composition. The function `retrograde-row` returns the retrograde version (reversal) of a row. The function `transpose-row` shifts the row to a new pitch class. The function `invert-row` inverts a twelve-tone row. The function `row->matrix` returns a list of lists that represents the twelve rows of a “Prime by Inversion” matrix. The utility function `print-matrix` prints a matrix so that each row appears on its own line.

***Example 9-3.*** *Twelve-Tone Functions.*

```
(define (retrograde-row row)
  (reverse row))

(define (transpose-row row to)
  (loop for pc in row collect (keynum->pc (+ pc to))))

(define (invert-row row)
  (loop for pc in row collect (keynum->pc (- 12 pc))))

(define (retrograde-invert-row row)
  (retrograde-row (invert-row row)))

(define (row->matrix row)
  (loop for i in (invert-row row)
        collect (transpose-row row i)))

(define (print-matrix matrix)
  (loop for row in matrix
        do (print row)))
```

The function `retrograde-row` is the easiest to implement, it simply calls the core function `reverse` to reverse the order of the input list. Notice that by defining this function we have essentially *renamed* `reverse` so that it better reflects our application's use of it. The function `transpose-row` iterates over an input row and adds an offset to each element. Notice that we filter this value through our `keynum->pc` function so that we our result list is guaranteed to remain pitch classes even after the addition of the offset. The function `invert-row` is similar to `transpose-row` except that it subtracts rather than adds an offset from each pitch-class in the input list. The function `retrograde-invert-row` can be implemented simply by calling two functions we have already defined in our implementation. The last two functions operate on a matrix of twelve-tone rows. The function `row->matrix` computes the prime-by-inversion matrix for given a row by transposing the row to each interval in the row's inversion. Note that this function returns a list of twelve lists: the outer list represents the matrix and each sublist represents one transposition of the original row. The function `print-matix` can be used to display the matrix so that each row is printed on a single line. Note that #8212; unlike all the other functions in our example #8212; this last function is called for its *effect* (displaying the matrix) rather than to calculate and return a value. Interaction 9-4 demonstrates using our functions on the row from Alban Berg's Violin Concerto.

***Interaction 9-4.*** *Twelve-tone matrix operations.*

```
cm> (define albans-row '(0 3 7 11 2 5 9 1 4 6 8 10))

cm> (retrograde-row albans-row)
(10 8 6 4 1 9 5 2 11 7 3 0)
cm> (transpose-row albans-row 3)
(3 6 10 2 5 8 0 4 7 9 11 1)
cm> (invert-row albans-row)
(0 9 5 1 10 7 3 11 8 6 4 2)
cm> (retrograde-invert-row albans-row)
(2 4 6 8 11 3 7 10 1 5 9 0)
cm> (print-matrix (row->matrix albans-row))
(0 3 7 11 2 5 9 1 4 6 8 10)
(9 0 4 8 11 2 6 10 1 3 5 7)
(5 8 0 4 7 10 2 6 9 11 1 3)
(1 4 8 0 3 6 10 2 5 7 9 11)
(10 1 5 9 0 3 7 11 2 4 6 8)
(7 10 2 6 9 0 4 8 11 1 3 5)
(3 6 10 2 5 8 0 4 7 9 11 1)
(11 2 6 10 1 4 8 0 3 5 7 9)
(8 11 3 7 10 1 5 9 0 2 4 6)
(6 9 1 5 8 11 3 7 10 0 2 4)
(4 7 11 3 6 9 1 5 8 10 0 2)
(2 5 9 1 4 7 11 3 6 8 10 0)
cm>
```

## Op. 3, No. 4: Determining Normal Order

The *normal order* of a group of notes is the most “tightl packed” permutation of its interval content. The most tightly packed permutation is the version of the set with the smallest total span. In the case of a “tie”, the permutation with smaller intervals leftward is the winner. For example there are four permutations of a dominant seventh chord:

***Table 9-1.*** *Inversions of a dominant seventh chord.*

| inversion | intervals |
|---|---|
| root | (0 4 7 10) |
| first | (0 3 6 8) |
| second | (0 3 5 9) |
| third | (0 2 6 9) |