Code Katas in Scheme with Wisp

Dr. Arne Babenhauserheide

<2020-07-20 Mo>

Code Katas are a way to hone your coding skills. I've long tiptoed around them; now it is time to change that and do some katas.

I'm starting with the code katas from codekata.com, because that site is simple and clear: Just descriptions of Katas. I will focus on the coding parts, because I want to work them with Wisp.

For more descriptions of code katas, just read codekata.com. There are other sites, but many of the older ones got lost in the domain churn that is still haunting the internet.[1]

Feel free to follow me along here or do the Katas yourself.

Check the RSS-Feed to get informed when I add new katas-work.

Contents

Kata02: Karate Chop

Create 5 different implementations of binary search, one per day.

Note: In the first version I had accumulating rounding errors, because I tried to get the simplest version — that isn't actually simple. I already knew that once, but had forgotten it. To hit such things outside production is exactly why I do these Katas. Also: training is useful.

Goals:

- Take notes of subtle errors.

[1] To secure this page against domain churn, it also exists within Freenet, so it will only fall to link churn if people don't visit it. Also you can find its full source on hg.sr.ht.

- check for merits of the different approaches: which was most fun, hardest to get working, best for production. Why?
- How did you find 5 unique approaches?

Requirements:

- Efficient enough with up to 100.000 elements that I don't kill it out of boredom and that it does not exhaust all memory.
- Returns the index on success and -1 on failure.

Test:

```
import : srfi srfi-64
define : check chop
    test-begin "check-chop"
    test-equal -1 : chop 3 #()
    test-equal -1 : chop 3 #(1)
    test-equal  0 : chop 1 #(1)
    test-equal  0 : chop 1 #(1 3 5)
    test-equal  1 : chop 3 #(1 3 5)
    test-equal  2 : chop 5 #(1 3 5)
    test-equal -1 : chop 0 #(1 3 5)
    test-equal -1 : chop 2 #(1 3 5)
    test-equal -1 : chop 4 #(1 3 5)
    test-equal -1 : chop 6 #(1 3 5)
    test-equal  0 : chop 1 #(1 3 5 7)
    test-equal  1 : chop 3 #(1 3 5 7)
    test-equal  2 : chop 5 #(1 3 5 7)
    test-equal  3 : chop 7 #(1 3 5 7)
    test-equal -1 : chop 0 #(1 3 5 7)
    test-equal -1 : chop 2 #(1 3 5 7)
    test-equal -1 : chop 4 #(1 3 5 7)
    test-equal -1 : chop 6 #(1 3 5 7)
    test-equal -1 : chop 8 #(1 3 5 7)
    ;; performance-requirement:
    test-equal -1 : chop -1 : list->vector : iota 100000
    test-equal 1 : chop 1 : list->vector : iota 100000
    test-equal 50 : chop 50 : list->vector : iota 100000
```

```
    test-equal 99995 : chop 99995 : list->vector : iota 100000
    test-end "check-chop"
```

First day: Default upper + lower implementation

```
define : chop-index target vec
  . "Simple index-based binary search."
  define len : vector-length vec
  if : = 0 len
     . -1
     let loop
       : lower 0
 upper len
       define mid : + lower : floor/ {upper - lower} 2
       define value : vector-ref vec mid
       ;; used debugging here:
       format #t "target ~a len ~a upper ~a lower ~a mid ~a value ~a vec ~a\n" tar
       cond
 {target = value}
   . mid
 ;; error: terminator-condition wasn't clear. Must be lower=mid due to floor/
 {lower = mid}
   . -1
 {target < value}
     loop lower mid
 {target > value}
     loop {mid + 1} upper

{{{tests}}}
check chop-index
```

Second day: Split vector in recursion

```
define : chop-split target vec
  . "Split the vector in recursion."
  let loop : (vec vec) (offset 0)
    ;; Error: (out-of-range "vector-ref" "Argument 2 out of range: ~S" (0) (0))
    define len : vector-length vec
```

```
    define half : floor/ len 2
    ;; Error: missed the zero-length case
    define value : if {len = 0} #f : vector-ref vec half
    format #t "target ~a len ~a half ~a offset ~a value ~a vec ~a\n" target len ha
    cond
      : = len 0
. -1
      : = target value
+ offset half
      : = len 1
. -1
      : > target value
let : : v2 : make-vector : - len half
  vector-move-left! vec half len v2 0
  loop v2 : + half offset
      : = half 0 ;; fallthrough: skip vector operation
. -1
      : < target value
;; Error: got the half wrong (used half + 1)
let : : v2 : make-vector half
  vector-move-left! vec 0 half v2 0
  loop v2 offset

{{{tests}}}
check chop-split
```

Fibonacci-sequence: optimize by golden ratios

This is an experimental implementation. It works for the tests, but I'm not sure whether it is general enough.

It took me a while to find that I have to allow for index-clamping at the upper end.

```
define : chop-fib target vec
  . "Use golden secquence steps."
  define len : vector-length vec
  define : fib n
    if {n = 0} 1
```

```
      let loop : (f1 1) (f2 1) (step 0)
 if {step >= n} f2
  loop f2 {f1 + f2} {step + 1}
 define : fib-steps len
   . "the number of steps needed in a fibonacci-sequence to hit at least the give
   if {len <= 1} 1
     let loop : (f1 1) (f2 1) (step 1)
if {f2 >= len} step
 loop f2 {f1 + f2} {step + 1}
 define max-steps : fib-steps len
 if : = 0 len
    . -1
    let loop
      : index : min {len - 1} : fib {max-steps - 2}
 step 1
 direction 'right
      cond
 {step > { 2 * max-steps }}  ;; error: limited steps, but the staggered step-sizes
  . -1
 {index < 0}
  . -1
 {index >= len}
  . -1
 else
  let : : value : vector-ref vec index
   ;; used debugging here:
   format #t "target ~a step ~a max-steps ~a len ~a index ~a value ~a vec ~a\n" ta
   cond
     {target = value}
       . index
     {step > max-steps}  ;; error: limited steps, but rounding of diff can break t
       . -1
     {index < 0}
       . -1
     {target > value}
       ;; move to right
       if : = index {len - 1}
  . -1
```

```
  if : equal? direction 'right
      let : : diff : fib {max-steps - step - 3}
  ;; format #t "right smaller diff ~a\n" diff
  loop ;
    min {len - 1} : + index diff ;; error: missed that I have to clamp the index w
    + step 2
    . 'right
      let : : diff : fib {max-steps - step - 2}
  ;; format #t "right larger diff ~a\n" diff
  loop ;
    min {len - 1} : + index diff
    + step 1
    . 'right
     {target < value}
       ;; move to left
       if : equal? direction 'left
   let : : diff : fib {max-steps - step - 3}
       ;; format #t "left smaller diff ~a\n" diff
       loop ;
 - index diff
 + step 2
 . 'left
   let : : diff : fib {max-steps - step - 2}
       ;; format #t "left larger diff ~a\n" diff
       loop ;
 - index diff
 + step 1
 . 'left


{{{tests}}}
 check chop-fib
```

Kata04: Data Munging

Part One: Weather Data

Output the day number (column one) with the smallest temperature spread (second column minus third column).

```
import : only (srfi srfi-1) first second
 only (ice-9 rdelim) read-line

define : read-columns port
    let loop : (columns '()) (index 0)
define column
  let linereader : (word-index index) (heading '()) (in-word #f) (ch (read-char po
    cond
      : and in-word : or (equal? ch #\space) (equal? ch #\newline)
cons index : cons word-index : apply string : reverse heading
      in-word
linereader (+ 1 word-index) (cons ch heading) in-word : read-char port
      : not : equal? #\space ch
linereader (+ 1 word-index) (cons ch heading) #t : read-char port
      else
linereader (+ 1 word-index) heading in-word : read-char port
if : equal? #\newline : peek-char port
    . columns
    loop
cons column columns
car : cdr column

define : read-one-line-by-column-names port column-info names
  define : get-name column
     cdr : cdr column
  define : index-starts-named-column? index
    filter : λ(x) : equal? index : first x
filter : λ(x) : member (get-name x) names
   . column-info
  let skip-to-column : (data '()) (index 0) (ch (read-char port))
      define relevant-columns : index-starts-named-column? index
```

```
      cond
  : or (equal? ch #\newline) (eof-object? ch)
    . data
  : null? relevant-columns
    skip-to-column data (+ index 1) (read-char port)
  else
    let :
      define column : first relevant-columns
      define column-name : get-name column
      define column-end : second column
      let read-column : (column-content (list)) (index index) (ch ch)
if : equal? column-end index
   skip-to-column
       cons (cons column-name (apply string (reverse column-content))) data
       . index
       read-char port
   read-column
     cons ch column-content
     + index 1
     read-char port

define : find-day-with-minimal-spread port columns
    . "Find the day (Dy) with minimal spread (MxT - MnT)"
    let loop : (min-day #f) (min-spread #f)
      if : eof-object? : peek-char port
 . min-day
 let :
   define line
     read-one-line-by-column-names port columns '("Dy" "MxT" "MnT")
   define : column-value name
     string->number : string-trim : assoc-ref line name
   define day : column-value "Dy"
   define MxT : column-value "MxT"
   define MnT : column-value "MnT"
   define spread
     if : and MxT MnT
- MxT MnT
. #f
```

```
    cond
      : not min-day
        loop day spread
      : or (not day) (not spread) ;; error condition: not parseable
        loop min-day min-spread
      {spread < min-spread}
        loop day spread
      else
        loop min-day min-spread


define : read-weather
    call-with-input-file "weather.dat"
      λ (port)
define columns : read-columns port
newline
read-line port ; strip empty line
display : find-day-with-minimal-spread port columns
newline
```

Let's add a little scaffolding to make this efficient[2] to run as script.

```
#!/usr/bin/env bash
exec guile -L $(dirname $(realpath "$0")) -x .w --language=wisp -e '(weather)' -c
; !#
define-module : weather
   . #:export : main

{{{weather}}}

define : main args
  read-weather
```

[2]The bash-indirection is pretty efficient, since it uses the compile-cache of Guile that can be mmapped directly. To test its performance again, I just ran the script 100 times in a simple for loop and divided the time by 100. Parsing the weather takes around 33ms on my machine. Running a module where the main just returns `#f` takes around 25ms.