

University of Massachusetts Lowell

91.301: Organization of Programming Languages
Fall 2004

Problem Set 8

Heartbeat Monitoring: Using Streams

Issued: Section 201 (Dimock): Monday, 15-Nov-2004, Section 202 (Yanco): Tuesday, 16-Nov-2004
Due: Section 201 (Dimock): Wednesday, 24-Nov-2004, Section 202 (Yanco): Tuesday, 30-Nov-2004
Reading: Covers text (SICP 2nd Edition by Abelson & Sussman): Through 3.5

Streams and Delayed Evaluation

As opposed to the standard tools introduced so far in this course, streams provide a different way of reasoning about procedures and processes. In particular, streams are useful for modeling infinite objects, such as a sequence of samples of a continuous-time signal, or the sequence of coefficients of an infinite power series. In this problem set, we will mainly concern ourselves with the former, that is, modelling signals using streams. In particular, we are going to help Louis Reasoner to write an HRM System – Heart Rate Monitoring system.

The provided file `ps8-ans.ss` defines the procedures and macros for the stream data type. It loads in code from `heartbeat.ss` and `plotizing.ss` and provides a template for your answers. The code in `heartbeat.ss` provides some useful stream functions. `heartbeat.ss` also defines the simple streams `ones` and `integers` (which seems an inappropriate name for the strictly positive numbers), and a procedure that creates a relatively complex stream: `get-patient-heart-signal`. You do not need to understand the code of `get-patient-heart-signal` or the `make-signal` routine, which it calls.

This problem set also concentrates on the use of higher order procedures such as `map-stream`, `filter-stream`, and some others that you must to write. You should familiarize yourself with the stream procedures in `heartbeat.ss` before you start to write your code.

In the file `plotizing.ss`, you do not have to initially understand any code, but you will need to use the routine `plot-stream` which pops up a window and plots part of a stream in the window: read the comments for that code. You can also change two globals in `plotizing.ss` to set the correct window size for your system: `*viewport-height*` and `*viewport-width*` set the size of the viewport in pixels. Some later problems may benefit from drilling down a bit and looking at the lower level stream plotting routines.

A Short History on the HRM System

Louis Reasoner decided to team up with his friend Ben Bitdiddle to open up their own business: Massachusetts Medical Software Inc. Soon after the opening, they were approached by the Mess-General Hospital for developing a piece of software to handle HRM (Heart Rate Monitoring). The hospital has already acquired the necessary hardware. The software must be able to take a patient's heartbeat signal, represented

as a sequence of numbers, analyze it and determine whether the heart-beat of the patient is regular. While Louis has no idea how to approach this problem, Ben has already accepted the offer of the project.

“It shouldn’t be too difficult if we use streams!!” suggests Ben. “What are streams?” inquires Louis. Ben, unfortunately, has to be away for a month due to some emergency, so he has handed the project to Louis and you. In this problem set, your job is to help Louis complete the HRM project.

Problem 1:

To begin, it is useful to review some basic things about streams. Recall from the text that we could define an infinite stream of ones and integers using the following two expressions. Type them in and use the `print-stream` procedure to print part of them out.

```
(define ones (cons-stream 1 ones))

(define integers (cons-stream 1 (add-streams ones integers)))
```

- Using the same idea, use the stream `integers` to define the stream named `sums`, starting with 1, where the n 'th element of `sums` is the sum of the first n elements of `integers`.
- Redefine procedure `add-streams` in terms of `map-streams`. Name your resulting procedure `add-streams2`.
- use the stream `integers` to define the stream named `prods`, starting with 1, where the n 'th element of `prods` is the produce of the first n elements of `integers`. The first few elements of `prods`:

```
> (limited-print-stream 5 prods)
#STREAM(1 2 6 24 120 ...)
```

- Define a procedure `bounded-random-stream` which takes two integers as arguments, a lower bound L and an upper bound U , and returns a stream of random integers such that each number is greater than or equal to L , but strictly less than U . Use the builtin procedure `random`.

Hand in a listing of your definitions.

Problem 2:

Louis and you then interview the hospital to collect more data about the interface between the software and the patient’s signal. The patient’s signal is represented by a stream of values, representing a sampling of the heartbeat at regular time intervals. The procedure `get-patient-heart-signal` in the given code simulates such a signal. Take a look at what those streams look like by either printing them out, or using the `plot-stream` procedure to plot the stream (or signal) on your screen. `plot-stream` takes three inputs – a stream, a maximum y value ym and a number n , and plots the first n values of the stream. The plot is scaled so that the vertical distance on the screen ranges from $-ym$ to $+ym$, and the horizontal distance on the screen is the range 0 through n . (The procedure will truncate values so that they lie within the range of $-ym$ and $+ym$, that is, values outside of this range will be plotted as asterisks “*” at the top and bottom edges of the screen.)

To see how this works, try

```
(define square-roots (map-stream sqrt integers))

(plot-stream square-roots 10 100)

(plot-stream (get-patient-heart-signal) 20 200)
```

Repeat the latter expression a number of times to get a feel for what the generated heart signals look like.

There is nothing to hand in for this problem.

Problem 3:

Now you are ready to design the software. Louis, although not a very good programmer, is very good at wishful thinking. He suggests that since the signals have a lot of noise (as shown by the ruggedness of the signals in the plots), one should do some more processing on the signal before feeding them into a regularity-detecting procedure. His idea is to separate out the valid stream elements from the invalid ones. He claims that the spikes in the plots are valid signals, and that in general the spikes have a value greater than a certain threshold value. The idea is to construct a new signal consisting only of the digits 0 and 1, based on whether the value of a stream element is above (\geq) or below ($<$) the threshold. In other words, whenever the original signal value is greater than the threshold, the new signal should contain a 1, and otherwise the new signal should contain a 0. (This process is frequently called "binarization" in signal processing.)

Write a procedure called `binarize` that, when given a signal and a threshold, returns a new signal consisting of zeros and ones at the appropriate places.

Hand in a listing of your procedure, and a transcript of its execution.

Problem 4:

Next, Louis wants to transform a signal in 0's and 1's in the following way:

input stream: 0 0 0 1 0 0 1 0 0 0 1 0 0 0 0 1 0.....

output stream: 4 7 11 16

In other words, the output stream has the positions of all the 1's in the input stream, counting left to right, starting from 1.

- a. Write a procedure called `peak-positions` that returns such an output stream given such an input stream.

(Hint: you might find it easier to write a supporting procedure for the main procedure.)

Hand in a listing of your procedure(s) and a demonstration of how it works.

- b. `peak-positions` has a problem for the signals for heartbeat monitoring: If a patient dies, or if the hardware malfunctions, there could be an input stream that is all 0s or all 1s after a certain point.

What is the behavior of `peak-positions` on a stream of all 0s?

The rest of this subproblem is optional!

Write `peak-positions2` that has two parameters: a stream `S` and a number `n`. If it finds more than `n` zeros in a sequence, then it terminates. (So applying `peak-positions2` to an infinite input stream may produce a finite output stream.)

Describe your design for `peak-positions2` and submit the code.

Problem 5:

There is one more step of processing, according to Louis. "If we take the output signal from `peak-positions` and somehow generate yet another new signal which consists of the difference between the adjacent elements in the stream, and if all numbers in this new signal are close to a constant, that means the patient's heartbeat is regular!" To complete the pre-processing of a signal, write a procedure called `peak-spacing` that takes a signal and returns a new signal whose elements are the differences between adjacent elements in the input signal.

Hand in a listing of your procedure and a transcript of its execution.

Problem 6:

"Now, we are done with writing all the basic building blocks, but we need a test which tells us whether a patient's heartbeat is regular," said Louis. Write a procedure called `check-regularity`, which takes four arguments: a signal, a tolerance-margin, a number n , and a procedure called `receiver`. `check-regularity` should assume that its input signal has already been passed through `peak-spacing`. It should examine the first n elements in the signal and call the procedure `receiver` with two arguments:

```
(receiver <boolean-value> <stream>)
```

where `<boolean-value>` is false iff (if and only if) the heartbeat is NOT regular, and `<stream>` is the stream of signals not tested by `check-regularity` (in other words, the elements of the input stream starting with the $(n + 1)$ -st).

An optional case: If there are fewer than $n + 1$ elements in the stream, `check-regularity` should call `receiver` with `#f` and `the-empty-stream`. This case will only occur with our inputs if `peak-positions2` is used rather than `peak-positions`.

`check-regularity` should consider a heartbeat regular iff the difference between any two adjacent elements of the input signal is less than (`<`) the value of the tolerance-margin. Remember that we are assuming that the signal has already been processed by the procedure `peak-spacing`. (Notice how the `receiver` argument can be used just as a method of returning more than one argument to the caller.)

Hand in a listing of `check-regularity` and a transcript showing it working. For testing your procedure, use 2 as the argument for the tolerance-margin and 7 as the argument for the number n .

Here is a possible test:

```
(define heartbeat (get-patient-heart-signal))
(define start-of-monitoring
  (peak-spacing (peak-positions2 1000
                          (binarize heartbeat 5))))

; get some visual idea of irregularity
(plot-stream
  (map-stream (lambda (x) (- x (stream-car start-of-monitoring)))
              start-of-monitoring)
  2 141)

; get a printout
(limited-print-stream 141 start-of-monitoring)

; now get the automated report
(define tolerance 2) ; durations can be at most 1 off
(define runlength 7)
(define (check-some s n)
  (if (= n 0)
      'done
      (check-regularity
       s
       tolerance
       runlength
       (lambda (s regular)
         (display (if regular 'regular 'irregular))
         (display " ")
         (check-some s (- n 1)))))))

(check-some start-of-monitoring 20) ; depends on heartbeat...
(check-some ones 20) ; should be all regular
(check-some (bounded-random-stream 3 6) 20) ; all regular with high probability
(set! runlength 2)
(check-some (bounded-random-stream 3 6) 20) ; mix of regular, irregular
```

Problem 7:

Now, Mess-General states that since the hardware for processing the signals is very expensive, they intend to have only one machine, and therefore the HRM system must be time-shared among all the patients in the hospital. To deal with multiple signals, one from each patient, Louis and you decide to write the sequence of patients' signals again as a stream, i.e., the software will be processing a stream of streams.

- a. Write *finite* stream called the `patient-stream`, which is simply a stream of signals from all the patients. Use the `get-patient-heart-signal` procedure to simulate the signals for the patients.
You might find `enumerate-interval` and `map-stream` useful.
- b. Write a procedure `stream-nth` that finds the n 'th element of a stream (starting with element 0).
- c. Write a procedure `stream-nth-tail` that returns the n 'th `stream-cdr` of a stream (0 returns the whole stream).
- d. Both `stream-nth` and `stream-nth-tail` have the same pattern. (`define (stream-nth-action f) ...`) that captures this pattern (returning a procedure taking `n` and `s`.
Now (`define stream-nth (stream-nth-action (lambda (??) ??))`) fill in for the `??`.
(`define stream-nth-tail (stream-nth-action (lambda (??) ??))`) fill in for the `??`.
- e. Write a procedure `plot-heartbeats` that takes `get-patient-heart-signal` stream as parameter `s` and a parameter n , and plots the first n heartbeats.
You might find `peak-positions` (`binarize`) and `stream-nth` useful.
Try your routine with `for-each-stream` over your `patient-stream`. You should start with (`init-plotting`) and invoke `reset-plotting` between each `plot-stream-no-init`.
- f. Write a procedure called the `monitor`, which takes three arguments: the `patient-stream`, a tolerance-margin, and a number n . This procedure should time-share in the following way:
For each patient, first plot the original signal of his/her heartbeat on the screen. Then, check whether the heartbeat of the patient is regular, using the tolerance-margin and n as arguments to the `check-regularity` procedure. If the patient's heartbeat is irregular, print a message of the form "Heartbeat of patient number n is irregular!"
When the last patient has been checked, go back to the first patient and go through the whole process again, now with `monitor` starting at the $(n + 1)$ -st heartbeat of every patient, and so on.

Hand in a listing of your procedures and a transcript showing them being run on test cases.

Problem 8:

Describe in a short paragraph what would have to be changed if the patient stream is an infinite stream. (This is perhaps not realistic, but interesting to think about.)