6.001 Tutorial 11 Notes

TA: Gerald Dalley
02–03 May 2005

1 Announcements

- Project 5 due Friday.
- Lectures clarification: there will be at least one question from each of the guest lectures on the exam.

2 Lazy Evaluation

The whole point of lazy evaluation is to put off doing work as long as you can. There are two ways of doing that. One is to be explicit:

```
(delay exp)
; Returns a promise to evaluate exp
; in current environment

(force promise)
; Returns the value of exp in the
; original environment
```

It's easier if everything is implicit. In this case, all arguments to functions are delayed, and values are forced when needed:

- Printing the value to screen.
- Used as an argument to a primitive procedure.
- Used in a conditional.
- Used as an operator.

2.1 Streams

Lazy evaluation has a bunch of interesting applications, but the one that we're most interested in is constructing streams, (usually) infinite data structures. A stream is just like a list, but the `cdr` of each cons cell is lazy-memoized.

```
(define (cons-stream x (y lazy-memo))
  (cons x y))

(define stream-car car)

(define stream-cdr cdr)
```

Some streams are very easy to construct. For example, the stream of infinite 1's:

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

For more complicated streams, we need to actually write new procedures:

```
;;; Adds two infinite streams element-wise
(define (add-two-streams s1 s2)
  (cons-stream
```

```
;;; Returns a new stream with each element
;;; multiplied by the scale factor
(define (stream-scale stream scale)
  (if (null? stream)
```

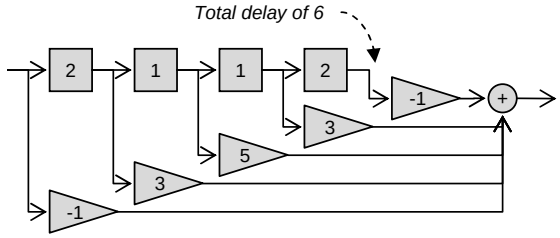
```
;;; For a positive delay, returns a stream
;;; with "delay" zeros prepended.  For
;;; negative delays, removes "delay" items
;;; from the input.
(define (stream-delay stream delay)
```

We can generalize `add-two-streams` a bit to allow for handling of any number of streams. Hint: make sure you keep track of when we're dealing with a regular list of streams and when we're dealing with the streams themselves.

```
;;; Adds all of the streams element-wise.
;;; Assumes each stream contains only
;;; numbers.  Returns the empty list if
;;; any of the streams has terminated.
(define (stream-add . streams)
```

2.2 Signal and Image Processing

We can now do some interesting signal processing applications *à la* 6.003 , 18.03, and 6.344. In signal processing, we can do things like smoothing of signals or edge detection by creating filter banks where rectangles represent delays, triangles represent scaling, and circled plus signs represent stream addition. An example filter bank for smoothing is as follows:

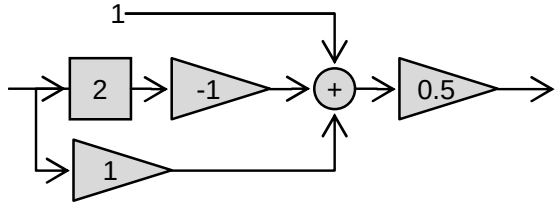


Now let's write some code to implement this filter bank:

```
;;; Smooths out the values in stream s1
;;; by applying the filter bank shown in
;;; the figure.
;;; Hint: add (using the right procedure)
;;;   a set of independently delayed and
;;;   scaled streams instead of using a let
;;;   statement to store a bunch of temporary
;;;   streams.
(define (stream-smooth s1)
```

If we're a little clever, we can extend these ideas to actually smooth the rows of images. Let's see it in action... (see `image-processing.scm` in the solution).

We can also use filter banks for things like edge detection:



(see `image-processing.scm` in the solution for source).