MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1996

Problem Set 9

Streams

Issued: Thursday, November 7, 1996

Written solutions due: Friday, November 22, 1996

Tutorial preparation for: week of November 18, 1996

Reading: Finish chapter 4, through the end of section 4.3; review section 3.5; code file `series.scm, mceval.scm` (attached)

Part 1: Tutorial exercises

Prepare the following exercises for oral presentation in tutorial:

Tutorial exercise 1: Do exercise 3.51 from the book. Make sure that you can explain what is going on—don't just say what the computer prints.

Tutorial exercise 2: Describe the streams produced by the following definitions. Assume that `integers` is the stream of non-negative integers (starting from 1):

```
(define A (cons-stream 1 (scale-stream 2 A)))

(define (mul-streams a b)
  (cons-stream
   (* (stream-car a) (stream-car b))
   (mul-streams (stream-cdr a)
                (stream-cdr b))))

(define B (cons-stream 1 (mul-stream B integers)))
```

Tutorial exercise 3: Given a stream `s` the following procedure returns the stream of all pairs of elements from `s`:

```
(define (stream-pairs s)
  (if (stream-null? s)
      the-empty-stream
      (stream-append
       (stream-map
        (lambda (sn) (list (stream-car s) sn))
        (stream-cdr s))
       (stream-pairs (stream-cdr s)))))

(define (stream-append s1 s2)
  (if (stream-null? s1)
      s2
      (cons-stream (stream-car s1)
                   (stream-append (stream-cdr s1) s2))))
```

(a) Suppose that `integers` is the (finite) stream 1, 2, 3, 4, 5. What is `(stream-pairs s)`? (b) Give the clearest explanation that you can of how `stream-pairs` works. (c) Suppose that `s` is the stream of positive integers. What are the first few elements of `(stream-pairs s)`? Can you suggest a modification of `stream-pairs` that would be more appropriate in dealing with infinite streams?

Tutorial exercise 4: Do exercise 3.52 from the book. Pay careful attention to how `memo-proc` may impact results when side-effects (procedures with state) are mixed with streams.

Part 2: Laboratory–adding streams to an evaluator

The file `mceval.scm` contains a simple version of the evaluator – one without lazy parameters, or memoization. We are going to install streams directly into this evaluator, by both modifying `eval` and adding some supporting infrastructure. These additions will also require you to think very carefully about the distinction between evaluator code and code in the `M-eval` language.

To get used to this evaluator, let's start by experimenting with it. To initialize this evaluator, evaluate the following expressions in the normal Scheme evaluator:

```
(define the-global-environment (setup-environment))

(driver-loop)
```

The first expression sets up a global environment with some predefined procedures in it. By looking at the file `mceval.scm`, you can see that we are relying on some underlying Scheme procedures as our basic environment. If you decide to add more inherent Scheme procedures to your global environment, you should add them to the list of `primitive-procedures`, and reset the global environment. The second expression starts up an evaluator loop, so that you can now try evaluating some simple expressions. Notice, by the way, that if you hit a bug, you will be thrown back into the underlying Scheme evaluator, and you will have to reinitialize your state to return to this new evaluator. This means you will need to reset the global environment, restart the driver loop, and re-evaluate any expressions you had evaluated earlier.

Exercise 1 Try this evaluator out on some simple expressions. Turn in a transcript of your tests.

To add streams directly into this evaluator, we need to do several things. First, we need to add a method for handling `cons-stream` expressions. As noted in the text, such expressions should be treated as a special form, so you will need to modify the evaluator in the corresponding spot. In this case, we will use a simplified notion of a "thunk" – we will wrap delayed experessions in a lambda of no arguments, which is an effective and convenient way to remember both the expression and the environment for later forcing.

The basic idea when evaluating a `cons-stream` expression, then, is to `cons` together the value of the first argument with a procedure of no arguments whose body is the second argument to `cons-stream`, i.e. evaluating `(cons-stream a b)` should be equivalent to evaluating

```
(cons a (lambda () b))
```

In this way, the second argument to the stream will be delayed, and will not be evaluated as part of the construction of the stream.

Exercise 2 Make whatever changes are appropriate to incorporate `cons-stream` as a special form in the evaluator, including whatever supporting procedures are needed. Turn in a listing of your additions, and an example of using this special form.

Exercise 3 In this implementation of streams, the first element is extracted using `stream-car`, which you can implement by

```
(define stream-car car)
```

You can add this to your system by evaluating the above expression inside of `M-eval` (Note – be sure you evaluate it in this evaluator, if you do it instead within the inherent Scheme evaluator you will not be able to access this definition).

Implementing `stream-cdr` is a bit more work, however, since it needs to deal with the delayed evaluation. In particular, evaluating a `stream-cdr` expression should get the value of the `cdr` of the argument, then apply this procedure to a list of no arguments to force the evaluation of the delayed expression.

Again, turn in a listing of what changes you make to the evaluator to implement this, and whatever supporting procedures you use.

Exercise 4: Test out your modifications by writing and evaluating definitions for each of the following:

- `ones`: the infinite stream of 1's.
- `non-neg-integers`: the stream of integers, $1, 2, 3, 4, \ldots$
- `alt-ones`: the stream $1, -1, 1, -1, \ldots$
- `zeros`: the infinite stream of 0's. Do this using `alt-ones`.

The following procedure will allow you to access the nth element of a stream:

```
(define (nth-stream n st)
  (if (= n 0)
      (stream-car st)
      (nth-stream (- n 1) (stream-cdr st))))
```

Evaluate this definition in your evaluator and try accessing elements of your stream.

Exercise 5 As noted in the text and in lecture, we can avoid a lot of redundant computation by memoizing streams. This means that the first time an element is accessed, it is evaluated, but in all future times the value is simply looked up in a local state variable. The procedure:

```
(define (memo-proc proc)
  ((lambda (already-run? result)
    (lambda ()
      (if (not already-run?)
          (begin (set! result (proc))
                 (set! already-run? true)
                 result)
          result)))
   false false))
```

will take as input a `lambda` expression and return a memoized version of that procedure.

Using your implementation of `cons-stream` as a guideline, implement a new stream constructor called `cons-stream-memo` with the property that the second element of the stream is a memoized lambda expression. You must be sure to type in and evaluate the definition of `memo-proc` in `M-eval` in order to be able to use it. You will note that this `memo-proc` is different than the version in section 3.5 of the book: it does not use `let`. Explain why.

Complete your implementation of `cons-stream-memo`. You may also need to create a new selector for `stream-cdr`. Turn in a listing of your modifications. Show examples where you generate some infinite streams (as in Exercise 4) and access elements of these streams – you should notice a difference in speed, especially as you move further down stream.

Part 3: Laboratory—Using streams to represent power series

Now we want to explore the *use* of streams for capturing computational processes. If you want, you can do this part of the problem set within your modified `M-eval` evaluator. However, since there is no debugger there, you may prefer to do this part of the problem set directly in the standard Scheme evaluator, which has built in stream primitives (without memoization).

We described in lecture a few weeks ago how to represent polynomials as lists of terms. In a similar way, we can work with *power series*, such as

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{3 \cdot 2} + \frac{x^4}{4 \cdot 3 \cdot 2} + \cdots$$

$$\begin{aligned}\cos x &= 1 - \frac{x^2}{2} + \frac{x^4}{4\cdot 3\cdot 2} - \cdots \\ \sin x &= x - \frac{x^3}{3\cdot 2} + \frac{x^5}{5\cdot 4\cdot 3\cdot 2} - \cdots\end{aligned}$$

represented as streams of infinitely many terms. That is, the power series

$$a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots$$

will be represented as the infinite stream whose elements are $a_0, a_1, a_2, a_3, \ldots$.[1]

Why would we want such a method? Well, let's separate the idea of a series representation from the idea of evaluating a function. For example, suppose we let $f(x) = \sin x$. We can separate the idea of evaluating f, e.g., $f(0) = 0, f(.1) = 0.0998334$, from the means we use to compute the value of f. This is where the series representation is used, as a way of storing information sufficient to determine values of the function. In particular, by substituting a value for x into the series, and computing more and more terms in the sum, we get better and better estimates of the value of the function for that argument. This is shown in the table, where $\sin \frac{1}{10}$ is considered.

Coefficient	x^n	term	sum	value
0	1	0	0	0
1	$\frac{1}{10}$	$\frac{1}{10}$	$\frac{1}{10}$	.1
0	$\frac{1}{100}$	0	$\frac{1}{10}$	.1
$-\frac{1}{6}$	$\frac{1}{1000}$	$-\frac{1}{6000}$	$\frac{599}{6000}$	.099833333333
0	$\frac{1}{10000}$	0	$\frac{599}{6000}$	.099833333333
$\frac{1}{120}$	$\frac{1}{100000}$	$\frac{1}{12000000}$	$\frac{1198001}{12000000}$	.09983341666

The first column shows the terms from the series representation for sine. This is the infinite series with which we will be dealing. The second column shows values for the associated powers of $\frac{1}{10}$. The third column is the product of the first two, and represents the next term in the series evaluation. The fourth column represents the sum of the terms to that point, and the last column is the decimal approximation to the sum.

With this representation of functions as streams of coefficients, series operations such as addition and scaling (multiplying by a constant) are identical to the basic stream operations. We provide series operations, though, in order to implement a complete power series data abstraction:

[1] In this representation, all streams are infinite: a finite polynomial will be represented as a stream with an infinite number of trailing zeroes.

```
(define (add-streams s1 s2)
  (cond ((stream-null? s1) s2)
        ((stream-null? s2) s1)
        (else
         (cons-stream (+ (stream-car s1) (stream-car s2))
                      (add-streams (stream-cdr s1)
                                   (stream-cdr s2))))))

(define (scale-stream c stream)
  (stream-map (lambda (x) (* x c)) stream))

(define add-series add-streams)

(define scale-series scale-stream)

(define (negate-series s)
  (scale-series -1 s))

(define (subtract-series s1 s2)
  (add-series s1 (negate-series s2)))
```

You can use the following procedure to examine the series you will generate in this problem set:

```
(define (show-series s nterms)
  (if (= nterms 0)
      'done
      (begin (write-line (stream-car s))
             (show-series (stream-cdr s) (- nterms 1)))))
```

You can also examine an individual coefficient (of x^n) in a series using `series-coeff`:

```
(define (series-coeff s n)
  (stream-ref s n))
```

We also provide two ways to construct series. `Coeffs->series` takes an list of initial coefficients and pads it with zeroes to produce a power series. For example, `(coeff->series '(1 3 4))` produces the power series $1 + 3x + 4x^2 + 0x^3 + 0x^4 + \ldots$.

```
(define (coeffs->series list-of-coeffs)
  (define zeros (cons-stream 0 zeros))
  (define (iter list)
    (if (null? list)
        zeros
        (cons-stream (car list)
                     (iter (cdr list)))))
  (iter list-of-coeffs))
```

`Proc->series` takes as argument a procedure p of one numeric argument and returns the series

$$p(0) + p(1)x + p(2)x^2 + p(3)x^3 + \cdots$$

The definition requires the stream `non-neg-integers` to be the stream of non-negative integers: $0, 1, 2, 3, \ldots$.

```
(define (proc->series proc)
  (stream-map proc non-neg-integers))
```

Note: Loading the code for this problem set will change Scheme's basic arithmetic operations `+`, `-`, `*`, and `/` so that they will work with rational numbers. For instance, `(/ 3 4)` will produce 3/4 rather than .75. You'll find this useful in doing the exercises below.

Exercise 6: Show how to define the series:

$$
\begin{aligned}
S_1 &= 1 + x + x^2 + x^3 + \cdots \\
S_2 &= 1 + \frac{x}{2} + \frac{x^2}{3} + \frac{x^3}{4} + \cdots
\end{aligned}
$$

Turn in your definitions and a couple of coefficient printouts to demonstrate that they work.

Exercise 7: Multiplying two series is a lot like multiplying two multi-digit numbers, but starting with the left-most digit, instead of the right-most.

For example:

```
      11111
   x  12345
  ---------

   11111
    22222
     33333
      44444
       55555
 -----------
   137165295
```

Now imagine that there can be an infinite number of digits, i.e., each of these is a (possibly infinite) series. (Remember that because each "digit" is in fact a term in the series, it can become arbitrarily large, without carrying, as in ordinary multiplication.)

Using this idea, complete the definition of the following procedure, which multiplies two series:

```
(define (mul-series s1 s2)
  (cons-stream ⟨ E1 ⟩
               (add-series ⟨ E2 ⟩
                           ⟨ E3 ⟩)))
```

To test your procedure, demonstrate that the product of S_1 (from Exercise 6) and S_1 is the series $1 + 2x + 3x^2 + 4x^3 + \cdots$. What is the coefficient of x^{10} in the product of S_2 and S_2? Turn in your definition of `mul-series`. (Optional: Give a general formula for the coefficient of x^n in the product of S_2 and S_2.)

Inverting a power series

Let S be a power series whose constant term is 1. We'll call such a power series a "unit power series." Suppose we want to find the *inverse* of S, namely, the power series X such that $S \cdot X = 1$. To see how to do this, write $S = 1 + S_R$ where S_R is the rest of S after the constant term. Then we want to solve the equation $S \cdot X = 1$ for S and we can do this as follows:

$$\begin{aligned} S \cdot X &= 1 \\ (1 + S_R) \cdot X &= 1 \\ X + S_R \cdot X &= 1 \\ X &= 1 - S_R \cdot X \end{aligned}$$

In other words, X is the power series whose constant term is 1 and whose rest is given by the negative of S_R times X.

Exercise 8: Use this idea to write a procedure `invert-unit-series` that computes $1/S$ for a unit power series S. To test your procedure, invert the series S_1 (from exercise 6) and show that you get the series $1 - x$. (Convince yourself that this is the correct answer.) Turn in a listing of your procedure. This is a very short procedure, but it is very clever. In fact, to someone looking at it for the first time, it may seem that it can't work—that it must go into an infinite loop. Write a few sentences of explanation explaining why the procedure does in fact work, and does not go into a loop.

Exercise 9: Use your answer from exercise 8 to produce a procedure `div-series` that divides two power series. `Div-series` should work for any two series, provided that the denominator series begins with a non-zero constant term. (If the denominator has a zero constant term, then `div-series` should signal an error.) Turn in a listing of your procedure along with three or four well-chosen test cases (and demonstrate why the answers given by your division are indeed the correct answers).

Exercise 10: Now suppose that we want to integrate a series representation. By this, we mean that we want to perform symbolic integration, thus, for example, given a series

$$a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots$$

we want to return the integral of the series (except for the constant term)

$$a_0 x + \frac{1}{2} a_1 x^2 + \frac{1}{3} a_2 x^3 + \frac{1}{4} a_3 x^4 + \cdots$$

Define a procedure `integrate-series-tail` that will do this. Note that all you need to do is transform the series

$$a_0 \quad a_1 \quad a_2 \quad a_3 \quad a_4 \quad a_5 \quad \cdots$$

into the series

$$a_0 \quad \frac{a_1}{2} \quad \frac{a_2}{3} \quad \frac{a_3}{4} \quad \frac{a_4}{5} \quad \frac{a_5}{6} \quad \cdots$$

Note that this means that the procedure generates the coefficients of a series starting with the first order coefficient, not that the zeroth order coefficient is 0.

Turn in a listing of your procedure and demonstrate that it works by computing `integrate-series-tail` of the series S_1 from exercise 6. How does this differ from the series S_2?

Exercise 11: Demonstrate that you can generate the series for e^x as

```
(define exp-series
  (cons-stream 1 (integrate-series-tail exp-series)))
```

Explain the reasoning behind this definition. Show how to generate the series for sine and cosine, in a similar way, as a pair of mutually recursive definitions. It may help to recall that the integral

$$\int \sin x = -\cos x$$

and that the integral

$$\int \cos x = \sin x$$

Exercise 12: Louis Reasoner is unhappy with the idea of using `integrate-series-tail` separately. "After all," he says, "if we know what the constant term of the integral is supposed to be, we should just be able to incorporate that into a procedure." Louis consequently writes the following procedure, using `integrate-series-tail`:

```
(define (integrate-series series constant-term)
  (cons-stream constant-term (integrate-series-tail series)))
```

He would prefer to define the exponential series as

```
(define exp-series
  (integrate-series exp-series 1))
```

Write a two or three sentence clear explanation of why this won't work, while the definition in exercise 11 does work.

Exercise 13: Write a procedure that produces the derivative of a power series. Turn in a definition of your procedure and some examples demonstrating that it works.

Exercise 14: Generate the power series for tangent, and secant. List the first ten or so coefficients of each series. Demonstrate that the derivative of the tangent is the square of the secant.

Exercise 15: We can also generate power series for inverse trigonometric functions. For example:

$$\tan^{-1}(x) = \int_0^x \frac{dz}{1+z^2}$$

Use this equation, plus methods that you have already created, to generate a power series for arctan. Note that $1+z^2$ can be viewed as a finite series. Turn in your definition, and a printout of the first few coefficients of the series.