MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 1995-96

Problem Set 7

Issued: Tuesday, April 2, 1996

Due: Solutions should be sent by email to your tutor by Wednesday, April 10 (even numbered recitation sections) or Friday April 12 (odd numbered recitation sections).

Reading: Sections 3.3 (through 3.33) and 3.5 of the notes

Assignment Problem Set 7 consists of Tutorial Exercises, Pre-Lab Exercises, and Laboratory Exercises. Tutorial Exercises are to be worked for presentation in tutorial, Pre-Lab Exercises are to be worked before going to Lab, and Laboratory Exercises are to be worked in Lab. Since many of the Laboratory Exercises test code you are to design in the Pre-Lab Exercises, it is important that you work the Pre-Lab Exercises before going to Lab.

Tutorial Exercises

Tutorial Exercise 7.1 Work Exercise 3.51 in the notes.

Tutorial Exercise 7.2 Work Exercise 3.12 in the notes.

Tutorial Exercise 7.3 Work Exercise 3.15 in the notes.

Tutorial Exercise 7.4 Work Exercise 3.16 in the notes.

Tutorial Exercises 7.5-7.7 are included in the laboratory assignment.

Laboratory: Streams

Streams are useful for modelling infinite mathematical objects, such as infinite power series. Power series can be used in many applications, including the representation of a local solution of a differential equation. As with the time sequences discussed in the notes, the coefficients of a power series can often be developed incrementally.

The file `ps7-series.scm` includes procedures that can be used to represent the series

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

by a stream of two-element lists {(0 a_0) (1 a_1) (2 a_2) (3 a_3) $\cdots$}. Each two-element list represents a term of the series, specifying the order and the coefficient of the term. To deal with terms we have the constructor `(make-term order coefficient)` and the selectors `(order term)`, and `(coefficient term)`.

To deal with series, we have the selectors `(first-term series)`, `(rest-terms series)`. To construct a series we use `cons-stream` directly.

Tutorial Exercise 7.5. Louis Reasoner is appalled at what he takes to be a lack of concern for data abstraction. He doesn't understand why `cons-stream` is used to construct series. Louis would like to use an abstract constructor `adjoin-term` like that used for polynomials in the Generic Arithmetic System of Problem Set 6. He tries to define `adjoin-term`

```
(define adjoin-term cons-stream)
```

but gets an error that he doesn't understand. Frustrated he tries again

```
(define (adjoin-term term series)
  (cons-stream term series))
```

Since this does not produce an error, he shows his definition to Alyssa P. Hacker, who says, "Louis, you really don't understand the differences between special forms and procedures, do you? Don't you remember the problem with `new-if` in Exercise 1.6 of the notes?

What would happen if Louis used his constructor to define the series whose coefficients are all 1:

```
(define (ones-series k)
  (adjoin-term (make-term k 1)
               (ones-series (+ k 1))))
```

and then attempted to evaluate `(stream-car (ones-series 0))`?

Series Arithmetic

In many power series the coefficients are rational numbers, and it is often convenient to operate on on these coefficients using rational number operations. In our system, `add`, `mul`, `sub`, `div`, and `equ?`, binary operators that are generic over Scheme numbers and rational numbers, are used to operate on coefficients. The system (in `ps7-rationals.scm`) uses Scheme's implicit typing to identify integers and numbers, but explicitly tags rational pairs. The procedure `(rational->real r)` is provided to converts a rational to a real; `(div n 1)` converts an integer to a rational. You do not have to understand how this system works in order to use it in this problem set.

Series arithmetic is a generalization of polynomial arithmetic. It is thus straightforward to add two series together

$$(a_0 + a_1 \times x + \cdots) + (b_0 + b_1 \times x + \cdots) = (a_0 + b_0) + (a_1 + b_1) \times x + \cdots$$

To perform this operation we provide the procedure `(add-series s1 s2)` that is defined like the analogous operation on polynomials in the Generic Arithmetic System of Problem Set 6. We also provide `(mul-series s1 s2)`) to multiply two series. Note that this is not term-by-term multiplication, as in Exercise 3.54 of the notes, rather it is a generalization of polynomial multiplication.

Similarly, it is straightforward to scale a series by a constant.

$$c \times (a_0 + a_1 \times x + a_2 \times x^2 + \cdots) = c \times a_0 + c \times a_1 \times x + c \times a_2 \times x^2 + \cdots$$

We provide a scaling operation `(scale-series k series)` to scale (using `mul`) all the term coefficients by the factor `k`. This defined in terms of the mapping operation `(stream-map proc stream)` that maps a procedure on the elements of a stream.

To convert a stream of numbers that represents the coefficients of the power series (starting with the coefficient of the term of order 0) to our representation of the corresponding series terms we provide the procedure `numbers->series`. Thus, if we have defined an infinite stream of ones as explained in section 3.5.2 of the notes,

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

we can evaluate

```
(define geometric-series (numbers->series ones))
```

to represent the series: $1 + x + x^2 + x^3 + \cdots$

You can use `(print-rat-terms n series)` to examine the coefficients of the first `n` terms of a series as rational numbers. To examine the equivalent decimal representation use `(print-real-terms n series)`. These procedures are in `ps7-utilities.scm`

PreLab Exercise 7.1a. To subtract two series we have the operation `sub-series`

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

Define `negate-series`.

Lab Exercise 7.1b. Verify that your definition works by examining the first 10 terms of

- `(negate-series geometric-series)`
- `(sub-series geometric-series *geometric-series)`.

where we define `*geometric-series` as

```
(define *geometric-series
  (cons-stream (make-term 0 0) (rest-terms geometric-series)))
```

Raising a series to a integral power is an operation of interest. For example squaring the series

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

produces the series

$$a_0^2 + 2a_0a_1x + (a_1^2 + 2a_0a_2) \times x^2 + \cdots$$

PreLab Exercise 7.2a. Define the procedure `(series-power n series)` that given a representation of a series returns a representation of that series raised to the n-th power, where n is a non-negative integer. Pay special attention to what `(series-power 0 series)` should return.

Lab Exercise 7.2b. Verify that your definition of `series-power` works by examining the first 10 terms in the series that corresponds to squaring `geometric-series`.

PreLab Exercise 7.3a. Work exercises 3.60 and 3.61 in the notes. Be sure to use the selectors and constructors appropriate for the implementation of series used in this Problem Set rather than that used in the notes.

Lab Exercise 7.3b. Test your definitions by examining the first 10 terms of the series corresponding to

$$\frac{1}{1 + x + x^2 + x^3 + \cdots}$$

Integration of Series

Given a representation of an infinite series, we can produce a representation of the indefinite integral of that series. The indefinite integral of

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

is

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

This corresponds to the stream $\{$`(1` a_0`) (2` $\frac{a_1}{2}$`) (3` $\frac{a_2}{3}$`) (4` $\frac{a_3}{4}a$`)` $\cdots\}$.

Note that this series does not include a constant term, corresponding to the constant of integration. An explicit `cons-stream` must be used in procedures that create series via integration to provide definite integrals that contain this constant term, as in the case of the procedures used to create the `sin-series` and `cos-series` below.

PreLab Exercise 7.4a. Complete the following definition of procedure `indefinite-integral` that takes a representation of a power series and returns a representation of the integral of that series (missing the constant term).

```
(define (indefinite-integral series) ... )
```

Also define a procedure, `derivative`, that takes a series representation as input and produces the series representation of the derivative of that series.

Lab Exercise 7.4b. Show that your definitions work. Use `print-rat-terms` to examine the first 10 terms of `(derivative (indefinite-integral geometric-series))`.

Given addition, scaling, and integration of series, one can write some remarkably concise definitions of series. For example, the Taylor series for the cos and sin functions are just series that start with 1 and 0 respectively and that have the property that each is the integral of the other (with a scale factor of -1 in one case.[1]

PreLab Exercise 7.5a. Complete the following definitions of `cos-series` and `sin-series`:

```
(define cos-series (cons-stream (make-term 0 1) ... ))

(define sin-series (cons-stream (make-term 0 0) ... ))
```

PreLab Exercise 7.5b. Demonstrate that your series for $\sin x$ and $\cos x$ work by examining the first 10 terms of the corresponding series.

Composition of Functions Represented by Series

In the remainder of this problem set we will be particularly interested in series of the form

$$
\begin{aligned}
p_1(x) &= 0 + a_1 \times x + a_2 \times x^2 + a_3 \times x^3 + \cdots \\
p_2(y) &= 0 + b_1 \times y + b_2 \times y^2 + b_3 \times y^3 + \cdots
\end{aligned}
$$

that is, series in which the coefficient of the constant term is zero. Consider the operation of composing two functions represented by series, i.e. computing $p_1\,(p_2\,(y))$. In particular, given the series representations of $p_1(x)$ and $p_2(y)$ `(compose-series p1 p2)` should return the series corresponding to

[1] This can be seen by examining the series themselves. It reflects the fact that

$$\frac{d\ \cos x}{dx} = -\ \sin x \tag{1}$$

$$\frac{d\ \sin x}{dx} = \cos x \tag{2}$$

$$
\begin{aligned}
p1(p2(y)) &= a_1 \times p2(y) + a_2 \times [p2(y)]^2 + a_3 \times [p2(y)]^3 + \cdots && (3)\\
&= c_1 \times y + c_2 \times y^2 + c_3 \times y^3 + \cdots && (4)
\end{aligned}
$$

Tutorial Exercise 7.6. The constant term in the series representation of composition of p_1 and p_2 has a value equal to $p_1(p_2(0))$. Why must the coefficient of the constant term of p_2 be zero when using Eq. 3 to compute $p_1(p_2(0))$?

Tutorial Exercise 7.7. It would seem straightforward to implement the `compose-series` operation as

```
(define (compose-series p1 p2)
  (let ((t1 (first-term p1)))
    (add-series
     (scale-series (coefficient t1) (series-power (order t1) p2))
     (compose-series (rest-terms p1) p2))))
```

Unfortunately this does not work. Why? Can the problem be corrected by skillfully using `delay` and `force`?

An implementation of `compose-series` that does work is based on the following observation: we can compute c_m in Eq. 4 without having to compute terms in Eq. 3 in which $p_2(y)$ is raised to a power higher than m, because $[p_2(y)]^n$ only includes terms whose order is greater than m when $n > m$. This observation suggests that `compose-series` be defined as:

```
(define (compose-series p1 p2)
  (define (get-first-nonzero-term s)
    (let ((term (first-term s)))
      (if (equ? (coefficient term) 0)
          (get-first-nonzero-term (rest-terms s))
          term)))

  (define (f t st)
    (scale-series (coefficient t) (series-power (order t) st)))

  (define (helper1 n)
    (define (loop1 k s result)
      (if (> k n)
          result
          (loop1
           (+ k 1)
           (rest-terms s)
           (add-series
            result
            (f (first-term s) p2)))))
    (loop1 2 (rest-terms p1) (f (first-term p1) p2)))

  (define (helper2 i)
    (define (loop2 ser)
      (let ((term (first-term ser)))
        (if (= (order term) i)
            term
            (loop2 (rest-terms ser)))))
    (cons-stream (loop2 (helper1 i))
                 (helper2 (+ i 1))))

  (let ((term (first-nonzero-term p2)))
    (if (= (order term) 0)
        (error "COMPOSE-SERIES series P2 can't start with " term)
  (helper2 (* (order (first-term p1)) (order term))))))


(define (self-compose-series x) (compose-series x x))
```

where `self-compose-series` composes a series with itself.

PreLab Exercise 7.6a. Briefly explain what procedures `helper1`, `helper2`, `loop1`, and `loop2` do, in terms of series operations.

Lab Exercise 7.6b. This definition of `compose-series` (available in `ps7-compose.sch` does work, as you can show by examining the terms of `(self-compose-series *geometric-series)`. The result is expected to be a series whose coefficients are the successive non-negative powers of 2 (as you can show). What are the coefficients of the first 10 terms of the series that represents $\sin(\sin y)$?

LabExercise 7.7. Examine the first 20 terms of the series produced by evaluating

```
(compose-series
  (series-power 5 geometric-series)
  (series-power 7 *geometric-series)).
```

The process of evaluation takes longer and longer as higher and higher order terms in the series are produced. (If the process seems to be taking *forever*, interrupt the interpreter.) Why is this so? Suggest how the code should be changed to speed up the computation.

Turning in your answers

The section on "Feedback" in the *Don't Panic* manual describes how to send Email from the 6.001 lab. Send your answer buffer to your tutor. Do this by starting a mail message and using the Edwin `M-x insert-buffer` command. Don't forget to include a reply email address (or else put your name in the message) so your tutor will know that the message is coming from you.

How much time did you spend on this homework assignment? Report separately time spent before going to lab and time spent in the 6.001 lab.

If you cooperated with other students working this problem set please indicate their names in the space below. As you should know, as long as the guidelines described in the *6.001 Policy on Cooperation* handout are followed, such cooperation is allowed and encouraged.

```
;;; ps7-series.scm ---- series representaion and operations
; represent the series g(x) = a0 + a1 x + a2 x^2 + ···
;   as the stream { (0 a0) (1 a1) (2 a2) ··· }
(define (order term) (car term))
(define (coefficient term) (cadr term))
(define (make-term order coefficient) (list order coefficient))
(define first-term stream-car)
(define rest-terms stream-cdr)
;;; numbers is the stream { a0 a1 a2 a3 ··· }
(define (numbers->series numbers)
  (define (helper n s)
    (cons-stream (make-term n (stream-car s))
                 (helper (+ n 1) (stream-cdr s))))
  (helper 0 numbers))
(define (constant-series k) ;;; k + k x + k x^2 + k x^3 + ···
  (numbers->series (stream-map (lambda (x) (* k x)) ones)))
(define geometric-series (constant-series 1)) ;;; 1 + x + x^2 + x^3 + ···
(define (scale-series k s)
  (stream-map (lambda (x) (make-term (order x) (mul k (coefficient x)))) s))
(define (sum-terms t1 t2)
  (if (= (order t1) (order t2))
      (make-term (order t1) (add (coefficient t1) (coefficient t2)))
      (error "SUM-TERMS terms of different orders" (list t1 t2))))
(define (add-series s1 s2)
  (let ((h1 (first-term s1)) (h2 (first-term s2)))
    (cond ((< (order h1) (order h2))
           (cons-stream h1 (add-series (rest-terms s1) s2)))
          ((< (order h2) (order h1))
           (cons-stream h2 (add-series s1 (rest-terms s2))))
          (else (cons-stream (sum-terms h1 h2)
                             (add-series (rest-terms s1) (rest-terms s2)))))))
(define (sub-series s1 s2) (add-series s1 (negate-series s2)))
(define (mul-terms t1 t2)
  (make-term (+ (order t1) (order t2))
             (mul (coefficient t1) (coefficient t2))))
(define (mul-term-series t s)
  (cons-stream (mul-terms t (first-term s))
               (mul-term-series t (rest-terms s))))
(define (mul-series s1 s2)
  (let ((h1 (first-term s1)) (h2 (first-term s2)))
    (cons-stream
     (mul-terms h1 h2)
     (add-series (mul-term-series h1 (rest-terms s2))
                 (add-series (mul-term-series h2 (rest-terms s1))
                             (mul-series (rest-terms s1) (rest-terms s2)))))))
```

```
;;; compose-streams
(define (compose-series p1 p2)
  (define (get-first-nonzero-term s)
    (let ((term (first-term s)))
      (if (equ? (coefficient term) 0)
          (get-first-nonzero-term (rest-terms s))
          term)))

  (define (f t st)
    (scale-series (coefficient t) (series-power (order t) st)))

  (define (helper1 n)
    (define (loop1 k s result)
      (if (> k n)
          result
          (loop1
           (+ k 1)
           (rest-terms s)
           (add-series
            result
            (f (first-term s) p2)))))
    (loop1 2 (rest-terms p1) (f (first-term p1) p2)))

  (define (helper2 i)
    (define (loop2 ser)
      (let ((term (first-term ser)))
        (if (= (order term) i)
            term
            (loop2 (rest-terms ser)))))
    (cons-stream (loop2 (helper1 i))
                 (helper2 (+ i 1))))

  (let ((term (first-nonzero-term p2)))
    (if (= (order term) 0)
        (error "COMPOSE-SERIES series P2 can´t start with " term)
  (helper2 (* (order (first-term p1)) (order term))))))


(define (self-compose-series x) (compose-series x x))
```

```
;;;; PS7-RATIONALS.SCM -- Rational number arithmetic package
;;; The following procedures try to remove common factors as early as
;;;  possible to reduce the cost of manipulation of big integers.
(define (+rational x y)
  (let ((dx (denominator x))
        (dy (denominator y)))
    (let ((d1 (gcd dx dy)))
      (let ((dx/d1 (quotient dx d1))
            (dy/d1 (quotient dy d1)))
        (if (= 1 d1)
            (make-rational
             (+ (* (numerator x) dy)
                (* dx (numerator y)))
             (* dx dy))
            (let ((tem (+ (* dy/d1 (numerator x))
                          (* dx/d1 (numerator y)))))
              (if (= tem 0)
                  0
                  (let ((d2 (gcd tem d1)))
                    (make-rational
                     (quotient tem d2)
                     (* dx/d1
                        (quotient dy d2)))))))))))
(define (*rational x y)
  (let ((nx (numerator x))
        (ny (numerator y))
        (dx (denominator x))
        (dy (denominator y)))
    (let ((gnxdy (gcd nx dy))
          (gnydx (gcd ny dx)))
      (let ((nx/gcd (quotient nx gnxdy))
            (dy/gcd (quotient dy gnxdy))
            (ny/gcd (quotient ny gnydx))
            (dx/gcd (quotient dx gnydx)))
        (make-rational (* nx/gcd ny/gcd)
                       (* dx/gcd dy/gcd))))))
(define (negate-rational x)
  (make-rational (- (numerator x))
 (denominator x)))
(define (-rational x y)
  (+rational x (negate-rational y)))
(define (recip-rational x)
  (make-rational (denominator x) (numerator x)))
(define (/rational x y)
  (*rational x (recip-rational y)))
(define (=rational x y)
  (= (* (denominator y) (numerator x))
     (* (denominator x) (numerator y))))
```

```
;;; Representation of rationals
(define (make-rational n d) ;gcd already done
  (cond ((= d 0) (error "zero denominator -- rational"))
        ((= n 0) 0)
        ((= d 1) n)
        ((< d 0) (make-rational (- n) (- d)))
        (else (construct-rational n d))))
(define (rational? q)
  (cond ((integer? q) true)
        ((pair? q) (eq? (car q) '*rat*))
        (else false)))
(define (construct-rational n d)
  (list '*rat* n d))
;;; number? is used instead of integer? in the following so that
;;; reals can be coerced to reals.
(define (numerator q)
  (cond ((number? q) q)
        ((eq? '*rat* (car q)) (cadr q))
        (else (error "bad rational"))))
(define (denominator q)
  (cond ((number? q) 1)
        ((eq? '*rat* (car q)) (caddr q))
        (else (error "bad rational"))))
(define (rat->real q)
  (/ (numerator q)
     (denominator q)))
;;;; Generic arithmetic with scheme numbers and rationals.
(define (with-coercions real-op rational-op)
  (lambda (x y)
    (if (and (rational? x) (rational? y))
        (rational-op x y)
        (real-op (rat->real x) (rat->real y)))))
(define add (with-coercions + +rational))
(define sub (with-coercions - -rational))
(define mul (with-coercions * *rational))
(define div (with-coercions / /rational))
(define equ? (with-coercions = =rational))
```

```
;;; ps7-streams.scm --- stream operations
(define (stream-filter pred st)
  (let ((elt (stream-car st)))
  (if (pred elt)
      (cons-stream elt (stream-filter pred (stream-cdr st)))
      (stream-filter pred (stream-cdr st)))))
(define (stream-map f st)
  (cons-stream (f (stream-car st))
       (stream-map f (stream-cdr st))))
(define (stream-for-each proc k st)
  (define (loop n stream)
    (cond ((<= n 0) 'done)
          (else (proc (stream-car stream))
                (loop (- n 1) (stream-cdr stream)))))
  (loop k st))
(define ones (cons-stream 1 ones))
;;; ps7-utilities.scm --- General series printers
(define (print-term x)
  (lambda (term)
    (display '<)
    (display (order term))
    (display " ")
    (x (coefficient term))
    (display '>)))
(define (print-rat r)
  (cond ((integer? r) (display r))
        ((= (denominator r) 1)
         (display (numerator r)))
        (else (display (list (numerator r) '/ (denominator r))))))
(define (print-rat-term t)
  ((print-term print-rat) t))
(define (print-rat-terms n s)
  (stream-for-each (lambda (x) (print-rat-term x) (newline)) n s) )
(define (print-real r)
  (display (rat->real r)))
(define (print-real-term t)
  ((print-term print-real) t))
(define (print-real-terms n s)
  (stream-for-each (lambda (x) (print-real-term x) (newline)) n s))
```