National University of Singapore
School of Computing
CS1101S: Programming Methodology (JavaScript)
Semester I, 2012/2013

Discussion Group Exercises 7

Problems:

Getting started

1. Describe the streams A and B produced by the following definitions. Assume that `integers` is the stream of positive integers (starting from 1):

```
function scale_stream(c, stream) {
    return stream_map(function (x){return c*x;}, stream);
}

var A = pair(1, function(){return scale_stream(2, A)});

function mul_streams(a,b) {
    return pair(head(a)*head(b),
        function() {
            return mul_streams(stream_tail(a), stream_tail(b));
        });
}

var B = pair(1, function(){return mul_streams(B, integers);});
```

Task Files

- lib/list.js
- lib/stream.js
- discussion_7_1.html
- **discussion_7_1.js**

Stream of pairs

2. Given a stream `s` the following function returns a stream of pairs of elements from `s`:

```
function stream_pairs(s) {
    if (is_empty_list(s)) {
        return [];
    } else {
        return stream_append(
                  stream_map(
                     function(sn){
                         return list(head(s), sn);
                     },
                     stream_tail(s)),
                  stream_pairs(stream_tail(s)));
    }
}
```

   (a) Suppose that `ints` is the (finite) stream 1, 2, 3, 4, 5. What is `stream_pairs(ints)`?

   (b) Give the clearest explanation that you can of how `stream_pairs` works.

   (c) Suppose that `integers` is the infinite stream of positive integers. What is the result of evaluating

```
var s2 = stream_pairs(integers);
```

   Hint: Note that the function `stream_append` is defined in `stream.js` a follows:

```
function stream_append(xs, ys) {
    if (is_empty_list(xs)) {
        return ys;
    } else {
        return pair(head(xs),
                    function() {
                        return stream_append(stream_tail(xs),
                                             ys);
                    });
    }
}
```

   (d) Consider the following variant of `stream_append`, called `stream_append_pickle` and the function `stream_pairs2` which makes use of it.

```
function stream_append_pickle(xs, ys) {
    if (is_empty_list(xs)) {
        return ys();
    } else {
        return pair(head(xs),
                    function() {
                        return stream_append_pickle(stream_tail(xs),
                                                    ys);
                    });
    }
}
```

```
function stream_pairs2(s) {
    if (is_empty_list(s)) {
        return [];
    } else {
        return stream_append_pickle(
                  stream_map(
                     function(sn){
                         return list(head(s), sn);
                     },
                     stream_tail(s)),
                  function() {
                      return stream_pairs(stream_tail(s));
                  });
    }
}

var s2 = stream_pairs2(integers);
```

Why does the function `stream_pair2` solve the problem that arose in the previous question?

(e) What are the first few elements of `stream_pairs2(integers)`? Can you suggest a modification of `stream_pairs2` that would be more appropriate in dealing with infinite streams?

Task Files

- lib/list.js
- lib/stream.js
- discussion_7_2.html
- **discussion_7_2.js**

Using streams to represent power series

3. The following power series

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

can be 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]

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

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:

```
function add_streams(s1, s2) {
    if (is_empty_list(s1)) {
        return s2;
    } else if (is_empty_list(s2)) {
        return s1;
    } else {
        return pair(head(s1) + head(s2),
                    function() {
                        return add_streams(stream_tail(s1),
                                           stream_tail(s2));
                    });
    }
}

function scale_stream(c, stream) {
    return stream_map(function(x){
                          return c * x;
                      },
                      stream);
}
```

```
var add_series = add_streams;

var scale_series = scale_stream;

function negate_series(s) {
    return scale_series(-1, s);
}

function subtract_series(s1, s2) {
    return add_series(s1, negate_series(s2));
}
```

We also provide two ways to construct series. The function `coeffs_to_series` takes a list of initial coefficients and pads it with zeroes to produce a power series. For example,

```
coeffs_to_series(list(1,3,4))
```

produces the power series $1 + 3x + 4x^2 + 0x^3 + 0x^4 + \ldots$.

```
function coeffs_to_series(list_of_coeffs) {
    var zeros = pair(0,
                     function(){
                         return zeros;
                     });
    function iter(list) {
        if (is_empty_list(list)) {
            return zeros;
        } else {
            return pair(head(list),
                        function() {
                            return iter(stream_tail(list));
                        });
        }
    }
    return iter(list_of_coeffs);
}
```

The function `fun_to_series` takes as argument a function 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: $0, 1, 2, 3, \ldots$.

```
function fun_to_series(fun) {
   return stream_map(fun, non_neg_integers);
}
```

To get some initial practice with streams, write definitions for each of the following:

- `alt_ones`: the stream $1, -1, 1, -1, \ldots$ in as many ways as you can think of.
- `zeros`: the infinite stream of 0's. Do this using `alt_ones` in as many ways as you can think of.

Now, show how to define the series:

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

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

Task Files

- lib/list.js
- lib/stream.js
- discussion_7_3.html
- **discussion_7_3.js**

Multiplying series

4. 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  12321
 ---------

11111
 22222
  33333
   22222
    11111
---------
136898631
```

   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 function, which multiplies two series:

```
function mul_series(s1, s2)
{
   return pair(<E1>,
      function(){  return add_series(<E2>, <E3>); });
}
```

   To test your function, demonstrate that the product of S_1 (from exercise 3) and S_1 is S_2. 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.)

Task Files

- lib/list.js
- lib/stream.js
- discussion_7_4.html
- **discussion_7_4.js**

Inverting a power series

5. 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.

   Use this idea to write a function `invert_unit_series` that computes $1/S$ for a unit power series S. To test your function, invert the series S_1 (from exercise 1) and show that you get the series $1 - x$. (Convince yourself that this is the correct answer.) Turn in a listing of your function. This is a very short function, 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 function does in fact work, and does not go into a loop.

Task Files

- lib/list.js
- lib/stream.js
- discussion_7_5.html
- **discussion_7_5.js**

6. Write a function `div_series` that divides two power series. The function `div_series` should work for any two series, provided that the denominator series begins with a non-zero constant term. Turn in a listing of your function along with three or four well-chosen test cases (and demonstrate why the answers given by your division are indeed the correct answers).

Task Files

- lib/list.js
- lib/stream.js
- discussion_7_6.html
- **discussion_7_6.js**