By Joshua Cantrell
jjc@cory.berkeley.edu

World of Streams

What is a stream?

A stream is best thought of as a list which is built algorithmically so that all of its elements don't have to exist immediately but can be built as needed. Our book, SICP, calls them delayed lists. To make them look even more similar to lists, the constructors used equate those found in lists:

`(cons-stream <car> <cdr>)` - The *car* is the element defined for this pair of the stream, and the *cdr* is the stream that represents the rest of the elements.
`(stream-car <stream-pair>)` - The *stream-pair* is a stream, and this gets the first element of the stream.
`(stream-cdr <stream-pair>)` - The *stream-pair* is a stream, and this gets the stream that represents the rest of the elements after the first.

What about delay and force?

To make streams, we need a way to delay an expression from being evaluated and a way to finish evaluating the delayed expression. Berkeley Scheme implements *delay* and *force* to help us do this.

Our implementation of *delay* acts like a syntactic sugar. When you type, `(delay <expression>)`, the interpreter really converts it to the expression, `(lambda () <expression>)`, before continuing to evaluate. Notice how putting the expression into the body of the procedure will cause it to only evaluate when the procedure is evaluated. This construct delays the evaluation of the expression. A delayed expression is commonly known as a *thunk* (you thunk about it, but never evaluated it).

Because the result of *delay* is a procedure, we know what *force* must do to finish evaluating the delayed expression. Our implementation of *force* simply evaluates the procedure, causing the body of the procedure to be evaluated as well.

`(delay <exp>)` - Causes the expression *exp* not to be evaluated until it is applied as an actual parameter to *force*.
`(force <delayed-exp>)` - Causes the delayed expression *delayed-exp* to be evaluated.

What are useful stream functions?

The book goes over how to write stream functions that can be extremely useful in practice. These parallel those functions written for lists (since streams are the same as lists, except for the delayed evaluation).

`(stream-ref <stream> <index>)` - Returns the element in the *index* position of the given *stream*.
`(stream-map <procedure> <stream>)` - Applies *procedure* to each element in *stream* and returns a stream of the results.
`(stream-for-each <procedure> <stream>)` - Applies *procedure* to each element in *stream*, but does not return any specified value.

By Joshua Cantrell
jjc@cory.berkeley.edu

(stream-filter <*predicate*> <*stream*>) - Returns a stream containing only those elements in *stream* for which the *predicate* is true.

(stream-map <*procedure*> <*stream*$_1$> <*stream*$_2$> ... <*stream*$_n$>) - Applies *procedure* to the elements at the same indexes in the given streams. The results are then returned in a stream[1]. If *stream*$_1$ is the *empty-stream*, stream-map stops mapping and returns the stream resulting up to that point.

(interleave <*stream*$_1$> <*stream*$_2$>) - Interleaves the elements of both streams together to make a single stream. The new stream starts with the 1st element of *stream*$_1$, followed by the 1st element of *stream*$_2$, followed by the 2nd element of *stream*$_1$, followed by the 2nd element of *stream*$_2$, and continuing.

Some techniques for developing streams.

You can make extremely simple streams. Naturally the simplest would look and be made like lists, but the more impressive streams are the infinite streams. Imagine if you tried to make a list of all integers above some number *n*.

```
(define (ints-gt-n n)
  (cons n (ints-gt-n (+ n 1))))
```

You should be leery about actually using the procedure. It doesn't have a base case, and will never stop adding a new element to the list. A sample substitutional method evaluation would be as follows:

1. `(ints-gt-n 0)`
2. `(#<(n) (cons n (ints-gt-n (+ n 1)))> 0)`
3. `(cons 0 (ints-gt-n (+ 0 1)))`
4. `(#<cons> 0 (#<(n) (cons n (ints-gt-n (+ n 1)))> (#<+> 0 1)))`
5. `(#<cons> 0 (#<(n) (cons n (ints-gt-n (+ n 1)))> 1))`
6. `(#<cons> 0 (cons 1 (ints-gt-n (+ 1 1))))`
7. `(#<cons> 0 (#<cons> 1 (#<(n) (cons n (ints-gt-n (+ n 1)))> (#<+> 1 1))))`
8. `(#<cons> 0 (#<cons> 1 (#<(n) (cons n (ints-gt-n (+ n 1)))> 2)))`
9. `(#<cons> 0 (#<cons> 1 (cons 2 (ints-gt-n (+ 2 1)))))`
10. And So On...

What if we wrote the same thing, except for using *cons-stream* instead of *cons*?

```
(define (ints-gt-n n)
  (cons-stream n (ints-gt-n (+ n 1))))
```

This doesn't have the same infinite recursive property because the *cdr* part of the *stream pair* is delayed until its value is needed. Of course, we could have a problem with a procedure that counted the number of elements in the resultant stream; it would never reach the end!

---

[1] This was defined in the homework. It's generally useful since it can take the place of the first defined *stream-map* and is far more flexible. It's defined on page 325 of the 2nd Edition SICP.

By Joshua Cantrell
jjc@cory.berkeley.edu

How does the delaying help make the problem tractable? Let's do a step by step comparison of the evaluation[2]:

```
1. (ints-gt-n 0)
2. (#<(n) (stream-cons n (ints-gt-n (+ n 1)))> 0)
3. (stream-cons 0 (ints-gt-n (+ 0 1)))
```
[3]

```
4. [ | ][▫]-->(ints-gt-n (+ 0 1))
     |
     v
    [0]
```

A value was actually returned by evaluating the expression! It returned a stream pair who's cdr never needed to be evaluated! Using the normal *cons* procedure, we needed to evaluate all of the expressions in the list because it wasn't a special form, but because *cons-stream* is a special form, the last expression didn't have to be evaluated.

How streams are used like lists.

Now how do streams work? It would be nice to see how cdring through a stream allows us to get to later values in the list even though they don't exist already! (Imagine that! Making up a list as we go! Isn't that called improvisation? An improvised list?) Let's start with the stream given above. What if we took three *stream-cdr*s of it?

```
1. (stream-cdr (stream-cdr (stream-cdr [ | ][▫]-->(ints-gt-n (+ 0 1)) )))
                                         |
                                         v
                                        [0]

   a. (ints-gt-n (+ 0 1))
   b. (#<(n) (stream-cons n (ints-gt-n (+ n 1)))> (#<+> 0 1))
   c. (#<(n) (stream-cons n (ints-gt-n (+ n 1)))> 1)
   d. (stream-cons 1 (ints-gt-n (+ 1 1)))

2. (stream-cdr (stream-cdr [ | ][▫]-->(ints-gt-n (+ 1 1)) ))
                             |
                             v
                            [1]

   a. (ints-gt-n (+ 1 1))
   b. (#<(n) (stream-cons n (ints-gt-n (+ n 1)))> (#<+> 1 1))
   c. (#<(n) (stream-cons n (ints-gt-n (+ n 1)))> 2)
   d. (stream-cons 2 (ints-gt-n (+ 2 1)))
```

---

[2] To help show that a *stream pair* is not a normal *pair*, I've double-boxed the cdr to indicate that what it points to is actually a delayed expression. I've also put the delayed expression in a rounded corner box to show that it was delayed.

[3] It may look like I've done something wrong, but don't forget the expression isn't evaluated and that *stream-cons* is a special form.

By Joshua Cantrell
jjc@cory.berkeley.edu

3. `(stream-cdr [2 | •]→(ints-gt-n (+ 2 1)) )`

```
2
```

a. `(ints-gt-n (+ 2 1))`
b. `(#<(n) (stream-cons n (ints-gt-n (+ n 1)))> (#<+> 2 1))`
c. `(#<(n) (stream-cons n (ints-gt-n (+ n 1)))> 3)`
d. `(stream-cons 3 (ints-gt-n (+ 3 1)))`

3. `[3 | •]→(ints-gt-n (+ 3 1))`

```
3
```

A more complex stream.

Using the *stream-map* that takes multiple streams, you can easily make an *add-streams* and then develop a stream by adding itself to its future elements. An example of this technique is the stream of Fibonacci numbers, `(0 1 1 2 3 5 8 ...)`.

```
(define fibs
  (cons-stream 0 (cons-stream 1 (add-stream fibs
                                            (stream-cdr fibs)))))
```

Notice that the only reason why this works is that the first two elements of the stream are predefined before the rest of the stream is evaluated. If this were not the case, we may have a problem with recursively evaluating the rest of the stream. This works by continuously building the stream by using previous elements in the stream. The equation for this is $a_{n+2}=a_n+a_{n+1}$ where $a_0=0$ and $a_1=1$.

Now let's trace through the first few cdrs of this stream. Before we go ahead and do this, I'll give the definition of *add-streams* because we'll need it.

```
(define (add-streams s1 s2)
  (stream-map + s1 s2))
```

Now we can start the step-by-step evaluation of four[4] *stream-cdr*s on the fibs stream.

1. `(stream-cdr (stream-cdr (stream-cdr (stream-cdr`

```
[0 | •]→(cons-stream 1 (add-stream fibs
                                   (stream-cdr fibs)))  ))))
```

a. `(cons-stream 1 (add-stream fibs (stream-cdr fibs)))`
b. `(cons-stream 1 (add-stream fibs (stream-cdr fibs)))`

[4] If you still don't see the pattern, then you are invited to continue the hand analysis beyond four.

By Joshua Cantrell
jjc@cory.berkeley.edu

2.
```
(stream-cdr (stream-cdr (stream-cdr
   [ | ]->(add-stream fibs
    |                (stream-cdr fibs))   )))
    v
   [1]
```

```
a. (add-stream fibs (stream-cdr fibs)))
b. (#<(s1 s2) (stream-map + s1 s2> :fibs scdr:fibs)[5]
c. (stream-map + :fibs scdr:fibs)
d. (#<stream-map> #<+> :fibs scdr:fibs)[6]
e. (stream-cons (#<+> (stream-car :fibs) (stream-car scdr:fibs))
          (stream-map #<+> (stream-cdr :fibs) (stream-cdr scdr:fibs)))
f. (stream-cons (#<+> 0 1)
          (stream-map #<+> (stream-cdr :fibs) (stream-cdr scdr:fibs)))
g. (stream-cons 1
          (stream-map #<+> (stream-cdr :fibs) (stream-cdr scdr:fibs)))
```

3.
```
(stream-cdr
 (stream-cdr [ | ]->(stream-map #<+> (stream-cdr :fibs)
              |                      (stream-cdr scdr:fibs))   ))
              v
             [1]
```

```
a. (stream-map #<+> (stream-cdr :fibs) (stream-cdr scdr:fibs))
b. (#<stream-map> #<+> scdr:fibs scdr:scdr:fibs)
c. (stream-cons (#<+> (stream-car scdr:fibs) (stream-car scdr:scdr:fibs))
      (stream-map #<+> (stream-cdr scdr:fibs) (stream-cdr scdr:scdr:fibs)))
d. (stream-cons (#<+> 1 1)
      (stream-map #<+> (stream-cdr scdr:fibs) (stream-cdr scdr:scdr:fibs)))
e. (stream-cons 2
      (stream-map #<+> (stream-cdr scdr:fibs) (stream-cdr scdr:scdr:fibs)))
```

4.
```
(stream-cdr [ | ]->(stream-map #<+> (stream-cdr scdr:fibs)
             |                      (stream-cdr scdr:scdr:fibs))   )
             v
            [2]
```

```
a. (stream-map #<+> (stream-cdr scdr:fibs) (stream-cdr scdr:scdr:fibs))
b. (#<stream-map> #<+> scdr:scdr:fibs scdr:scdr:scdr:fibs)
c. (stream-cons (#<+> (stream-car srdr:scdr:fibs)
                      (stream-car scdr:scdr:scdr:fibs))
                (stream-map #<+> (stream-cdr scdr:scdr:fibs)
                                 (stream-cdr scdr:scdr:scdr:fibs)))
```

---

[5] You'll have to excuse me for this strange notation, but it was becoming too cluttered and tedious putting the streams directly into the expressions. *:fibs* should be replaced with what the symbol *fibs* is bound to, and *scdr:fibs* should be replaced with the *stream-cdr* of what the symbol *fibs* is bound to. I'll let you guess what *scdr:scdr:fibs* stands for.

[6] I'll be substituting in the logical answer that stream-map evaluates to. I don't want to give away the homework.

By Joshua Cantrell
jjc@cory.berkeley.edu

d.
```
(stream-cons (#<+> 1 2)
             (stream-map #<+> (stream-cdr scdr:scdr:fibs)
                              (stream-cdr scdr:scdr:scdr:fibs)))
```
e.
```
(stream-cons 3
             (stream-map #<+> (stream-cdr scdr:scdr:fibs)
                              (stream-cdr scdr:scdr:scdr:fibs)))
```

4.
```
(stream-cdr [ | ]-->(stream-map #<+> (stream-cdr scdr:scdr:fibs)          )
             |                       (stream-cdr scdr:scdr:scdr:fibs))
             v
            [2]
```

Another way to visualize how we actually came up with this definition of Fibonacci numbers is by looking at the equations, $a_{n+2}=a_n+a_{n+1}$ where $a_0=0$ and $a_1=1$. Notice that the first two elements are given to us, $a_0=0$ and $a_1=1$. Those elements that follow will just be related to the previous elements in the stream, $a_n=a_{n-2}+a_{n-1}$. We start the generalization of stream at a_2, so we can see that adding all of the elements starting from beginning of the Fibonacci stream with those elements starting at the cdr of the Fibonacci stream will give us the rest of the elements, from a_2 into infinity. Here's a diagram modified from SICP[7] illustrating this:

```
        1   1   2   3   5    8   13   21  ...  = (stream-cdr fibs)
+       0   1   1   2   3    5    8   13  ...  = fibs
----------------------------------------------------------------
0   1   1   2   3   5   8   13   21   34  ...  = fibs
        \______________________________/
        (stream-cdr (stream-cdr fibs))
```

Making Streams out of Multiple Streams

Using *stream-map* tends to be useful for taking one or more streams, performing some operation on each of the elements of the stream(s), and returning a new stream. The *fibs* stream only shows the combination of a stream with part of itself to create the rest of itself. By combining a stream with a goal stream, it is also possible to create the rest of the goal stream. I'll go through some steps to show how useful this technique can be. Let's start with a stream of ones.

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

[7] Abelson & Sussman, *SICP 2nd Edition, pg 329*

By Joshua Cantrell
jjc@cory.berkeley.edu

Now we'd like to define a stream of all integers equal to or greater than one, and name it *integers*. Let's try to draw a picture that shows two streams that could be combined to create the wanted stream:

```
       1   1   1   1   1   1   1   1 ... = ones
   +   1   2   3   4   5   6   7   8 ... = integers
   ----------------------------------------------
   1   2   3   4   5   6   7   8   9 ... = integers
       \______________________________/
           (stream-cdr integers)
```

This seems easy enough. As long as we add the streams together to find the rest of one of the given streams, we can produce a complete stream. The code to produce the above stream is:

```
(define integers (cons-stream 1 (stream-map + ones integers)))
```

Compare this to the *ints-gt-n* stream constructor. Notice that both add one to each previous element of the stream, but this one does it using two streams, *map* and the + operator.

Now let's make a stream containing all lists that contain all integers from 1 to any number *n* in ascending order. The stream would look like:

```
str-o-ints → ((1) (1 2) (1 2 3) (1 2 3 4) (1 2 3 4 5) ...)
```

How should we approach this problem? You certainly can't *add* any of our previous streams to come up with this stream. Let's start with the diagram.

```
        2      3        4           5       ... = (stream-cdr integers)
 op    (1)   (1 2)   (1 2 3)    (1 2 3 4)   ... = str-o-ints
 ------------------------------------------------------------
 (1) (1 2) (1 2 3) (1 2 3 4) (1 2 3 4 5)    ... = str-o-ints
     \___________________________________/
          (stream-cdr str-o-ints)
```

If we can find a procedure *op* that will take each element of both streams to form the needed elements, then we've solved our problem. So what procedure *op* will take the number and attach it to the end of the previous list of numbers?

```
(define (op num lst) (append lst (list num)))
```

Now we can create the *str-o-ints* stream.

```
(define str-o-ints (cons-stream '(1)
                      (map op (stream-cdr integers) str-o-ints)))
```

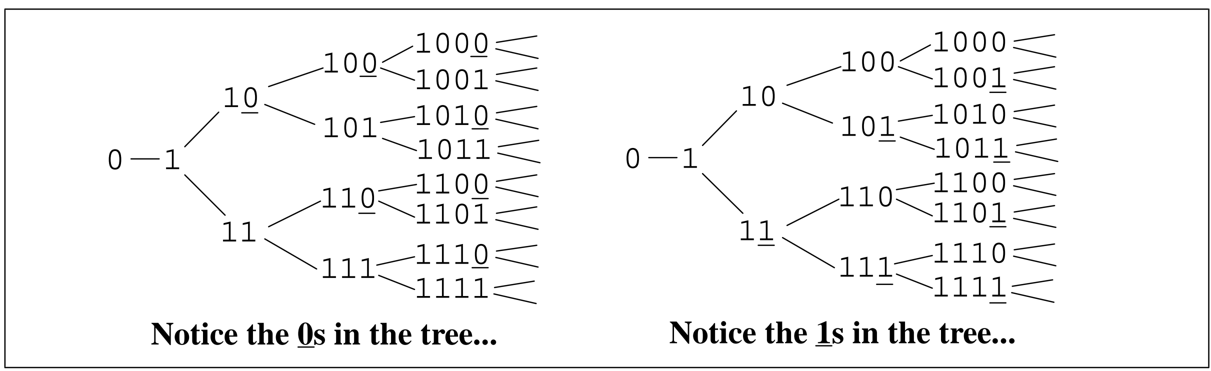
Another useful method of creating new streams is by using *stream-filter*. We can create the stream of odd integers starting from 1 by the following expression:

```
(define odd-ints (stream-filter odd? integers))
```

By Joshua Cantrell
jjc@cory.berkeley.edu

An even more complex stream.

There are times when we want to represent streams best calculated using tree like computations. An example of this would be making a stream of all binary numbers[8]. The stream of binary numbers would look something like: `(0 1 10 11 100 101 110 111 ...)`. Now how should we approach solving this problem? One hint would be that it is best seen looking at it in a tree containing all the binary numbers.



Notice the 0s in the tree... **Notice the 1s in the tree...**

Going down one branch adds a *0* to the end, and going down the other branch adds a *1* to the end. Assuming that we make all the numbers at each division into more branches, we should simply be able to use the *word*[9] procedure to add a *0* and a *1* to each number. The question is, what's the best way to add them to the end, one after another? How about the *interleave* procedure[10]?

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

This *interleave* procedure will disperse the two streams equally into a new stream. So what if we made a stream by mapping a *0* to the end of each element in the *binary* stream, and another stream by mapping a *1* to the end of each element in the *binary* stream, and finally *interleave*d them together? This would produce our necessary nodes in the tree at the first split, which would mean we'd produce the nodes for the second splits, leading to the correct nodes at the third splits, and so on...

```
(define binary
  (cons-stream 0 (cons-stream 1 (interleave
                                     (stream-map (lambda (x) (word x 0))
                                                 (stream-cdr binary))
                                     (stream-map (lambda (x) (word x 1))
                                                 (stream-cdr binary)) ))))
```

---

[8] This document is not trying to cover different bases of numbers. If you don't know about binary, it's about time you found out. Ask someone, or look it up.
[9] Don't forget, *word* is only in Berkeley Scheme, not in normal Scheme.
[10] *SICP 2nd Edition*, pg. 341

By Joshua Cantrell
jjc@cory.berkeley.edu

Let's try a new method of tracing how this works. We will write out what we can assume the resultant streams will be given what we already know. Notice that at first, all we know is that the first two values in the *binary* stream are 0 and 1. The bold numbers are new things in the stream. The underlined numbers are those elements in the stream currently being used to calculate the new value in the stream. Notice how the interleaving produces one value with a *0* attached and then follows that with a value with a *1* attached.

1. binary → (**0 1** ...)
   (stream-cdr binary) w/ appended 0 → (**10** ...)
   (stream-cdr binary) w/ appended 1 → (**11** ...)
2. binary → (0 1 **10** ...)
   (stream-cdr binary) w/ appended 0 → (10 **100** ...)
   (stream-cdr binary) w/ appended 1 → (11 **101** ...)
3. binary → (0 1 10 **11** ...)
   (stream-cdr binary) w/ appended 0 → (10 100 **110** ...)
   (stream-cdr binary) w/ appended 1 → (11 101 **111** ...)
4. binary → (0 1 10 11 **100** ...)
   (stream-cdr binary) w/ appended 0 → (10 100 110 **1000** ...)
   (stream-cdr binary) w/ appended 1 → (11 101 111 **1001** ...)
5. binary → (0 1 10 11 100 **101** ...)
   (stream-cdr binary) w/ appended 0 → (10 100 110 1000 **1010** ...)
   (stream-cdr binary) w/ appended 1 → (11 101 111 1001 **1011** ...)
6. binary → (0 1 10 11 100 101 **110** ...)
   (stream-cdr binary) w/ appended 0 → (10 100 110 1000 1010 **1100** ...)
   (stream-cdr binary) w/ appended 1 → (11 101 111 1001 1011 **1101** ...)
7. binary → (0 1 10 11 100 101 110 **111** ...)
   (stream-cdr binary) w/ appended 0 → (10 100 110 1000 1010 1100 **1110** ...)
   (stream-cdr binary) w/ appended 1 → (11 101 111 1001 1011 1101 **1111** ...)
8. And So On...