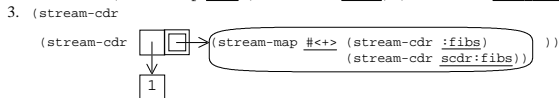
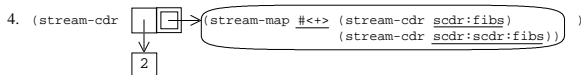


- a. (add-stream fibs (stream-cdr fibs))
- b. (#(s1 s2) (stream-map + s1 s2) :fibs scdr:fibs)⁵
- c. (stream-map + :fibs scdr:fibs)
- d. (#stream-map #<+> :fibs scdr:fibs)⁶
- e. (stream-cons #<+> (stream-car :fibs) (stream-car scdr:fibs))
(stream-map #<+> (stream-cdr :fibs) (stream-cdr scdr:fibs))
- f.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



- 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)))

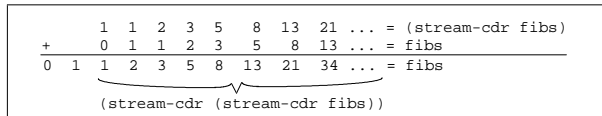


- a. (stream-map #<+> (stream-cdr scdr:fibs) (stream-cdr scdr:scdr:fibs))
- b. (#stream-map #<+> scdr:scdr:fibs scdr:scdr:scdr:fibs)
- c. (stream-cons #<+> (stream-car scdr:scdr:fibs) (stream-car scdr:scdr:scdr:fibs))
(stream-map #<+> (stream-cdr scdr:scdr:fibs) (stream-cdr scdr:scdr:scdr:fibs))

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.

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

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 SICP⁷ illustrating this:



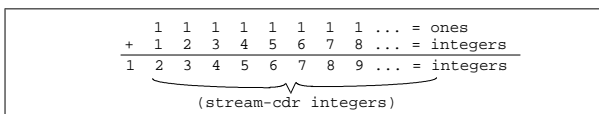
Making Streams out of Multiple Streams

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

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:



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:

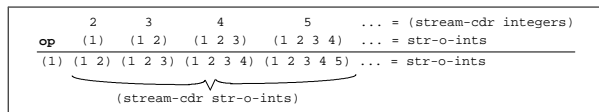
```
(define integers (cons-stream 1 (stream-map + ones integers)))
```

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.



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.

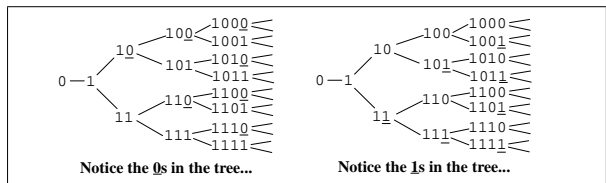
```
(define str-o-ints (cons-stream '(1) (map op (stream-cdr integers) str-o-ints)))
```

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))
```

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 numbers⁸. 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.



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` 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` procedure¹⁰?

```
(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` 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

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 → (1**0** ...)
(stream-cdr binary) w/ appended 1 → (1**1** ...)
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...