A stream is an element and a “promise” to evaluate the rest of the stream. You’ve already seen multiple examples of this and its syntax in lecture and in the book, so I will not dwell on that. Suffice it to say, streams is one of the most mysterious topics in CS61A, but it’s also one of the coolest; mysterious, because defining a stream often seems like black magic (and requires MUCH more trust than whatever trust you worked up for recursion); cool, because things like infinite streams allows you to store an INFINITE amount of data in a FINITE amount of space/time!

How is that possible? We’re not going to be too concerned with the below-the-line implementations of streams, but it’s good to have an intuition. Recall that the body of a lambda is NOT executed until it is called. For example, typing into STk:

```
(define death (lambda () (/ 5 0)))
```

Scheme says “okay”, happily binding death to the lambda. But if you try to run it:

```
(death) ;; Scheme blows up
```

The crucial thing to notice is that, when you type the define statement, Scheme did NOT try to evaluate (/ 5 0) – otherwise, it would’ve died right on the spot. Instead, the evaluation of (/ 5 0) is delayed until the actual procedure call. Similarly, if we want to represent an infinite amount of information, we don’t have to calculate all of it at once; instead, we can simply calculate ONE piece of information (the stream-car), and leave instructions on how to calculate the NEXT piece of information (the “promise” to evaluate the rest).

It’s important to note, however, that Scheme doesn’t quite use plain lambda for streams. Instead, Scheme memorizes results of evaluating streams to maximize efficiency. This introduces some complications that we’ll visit later. The delay and force operators, therefore, are special operators with side effects.

QUESTIONS

1. Define a procedure (ones) that, when run with no arguments, returns a cons pair whose car is 1, and whose cdr is a procedure that, when run, does the same thing.

2. Define a procedure (integers-starting n) that takes in a number n and, when run, returns a cons pair whose car is n, and whose cdr is a procedure that, when run with no arguments, does the same thing for n+1.
Using Stream Operators

Here are some operators on streams that we’ll be using quite a bit:

- `(stream-map proc s ...)` – works just like `list-map`; can take in any number of streams
- `(stream-filter proc s)` – works just like `list filter`
- `(stream-append s1 s2)` – appends two finite streams together (why not infinite streams?)
- `(interleave s1 s2)` – interleave two streams into one, with alternating elements from `s1` and `s2`

Constructing Streams

This is the trickiest part of streams. I said that the topic of streams is a black art, and you’ll soon see why. The construction of streams is counter-intuitive with a heavy dose of that-can’t-possibly-work. So here are some rough guidelines:

1. **cons-stream is a special form!** `cons-stream` will NOT evaluate its second argument (the `stream-cdr`); obviously, this is desirable, since we’d like to delay that evaluation.

2. **Trust the, err, stream.** From the first day, we’ve been chanting “trust the recursion”. Well now that you’re (slightly more) comfortable with that idea, we need you to do something harder. When you’re defining a stream, you **have to think as if that stream is already defined**. It’s often very difficult to trace through how a stream is evaluated as you `stream-cdr` down it, so you have to work at the logical level. Therefore, the above definition of `integers` works. However, be careful that you don’t trust the stream too much. For example, this won’t work:

```
(define integers integers)
```

3. **Learn how to think about `stream-map`.** Consider this definition of `integers`, given the stream `ones`, a stream of ones, defined in SICP:

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

If the above definition of `integers` puzzles you a bit, here’s how to think about it:

```
1  2  3  4  5  6  ...  <= integers (as taken from the last line)
+  1  1  1  1  1  1  ...  <= ones
============================
1  2  3  4  5  6  7  ...  <= integers
```

If you’re ever confounded by a `stream-map` expression, write it out and all should be clear. For example, let’s try a harder one – `partial-sum`, whose `ith` element is the sum of the first `i` integers. It is defined thus:

```
(define partial-sum (cons-stream 0 (stream-map + partial-sum integers)))
```

If you find it odd to have `integers` or `partial-sum` as one of the things you’re adding up, refer to guideline #2.

4. **Specify the first element(s).** Recall that a stream is one element and a promise to evaluate more. Well, often, you have to specify that one element so that there’s a starting point. Therefore, unsurprisingly, when you define streams, it often looks like

```
(cons-stream [first element] [a stream of black magic]).
```
But there are many traps in this. In general, what you’re avoiding is an infinite loop when you try to look at some element of a stream. stream-cdr is usually the dangerous one here, as it may force evaluations that you want to delay. Note that Scheme stops evaluating a stream once it finds one element. So simply make sure that it’ll always find one element immediately. For example, consider this definition of fibs that produces a stream of Fibonacci numbers:

```
(define fibs (cons-stream 0 (stream-map + fibs (stream-cdr fibs))))
```

Its intentions are admirable enough; to construct the next fib number, we add the current one to the previous one. But let’s take a look at how it logically stacks up:

```
0           <= your stream-car
0 1 1 2 3 5 ...  <= fibs?
+ 1 1 2 3 5 8 ...  <= (stream-cdr fibs)
==============================
0 1 2 3 5 8 13 ...  <= not quite fibs...
```

Close, but no cigar (and by the definition of Fibonacci numbers you really can’t just start with a single number). Actually, it’s even worse than that; if you type in the above definition of fibs, and call (stream-cdr fibs), you’ll send STk into a most unfortunate infinite loop. Why? Well, stream-cdr forces the evaluation of (stream-map + fibs (stream-cdr fibs)). stream-map is not a special form, so it’s going to evaluate both its arguments, fibs and (stream-cdr fibs). What’s fibs? Well, fibs is a stream starting with 0, so that’s fine. What’s (stream-cdr fibs)? Well, stream-cdr forces the evaluation of (stream-map + fibs (stream-cdr fibs)). stream-map is not a special form, so it’s going to evaluate both its arguments, fibs and (stream-cdr fibs). What’s fibs? Well, fibs is a stream starting with 0, so that’s fine. What’s (stream-cdr fibs)? You get the point.

How do we stop that horrid infinite loop? Well, it was asking for (stream-cdr fibs) that was giving us trouble – whenever we try to evaluate (stream-cdr fibs), it goes into an infinite loop. Thus, why don’t we just specify the stream-cdr?

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

So, then, let’s try it again. What’s (stream-cdr fibs)? Well, (stream-cdr fibs) is a stream starting with 1. There! Done! See? Now, it’s pretty magical that adding one more element fixes the stream-cdr problem for the whole stream. Convince yourself of this. As a general rule of thumb, if in the body of your definition you use the stream-cdr of what you’re defining, you probably need to specify two elements. Let’s check that it logically works out as well:

```
0 1           <= your stream-car
0 1 1 2 3 5 ...  <= fibs
+ 1 1 2 3 5 8 ...  <= (stream-cdr fibs)
==============================
0 1 1 2 3 5 8 13 ...  <= win!
```

QUESTIONS: Describe what the following expressions define.

1.  

```
(define s1 (add-stream (stream-map (lambda (x) (* x 2)) s1) s1))
```
2. (define s2
   (cons-stream 1
      (add-stream (stream-map (lambda (x) (* x 2)) s2) s2)))

3. (define s3
   (cons-stream 1
      (stream-filter (lambda (x) (not (= x 1))) s3)))

4. (define s4
   (cons-stream 1
      (cons-stream 2
         (stream-filter (lambda (x) (not (= x 1))) s4))))

5. (define s5
   (cons-stream 1
      (add-streams s5 integers)))

6. Define facts without defining any procedures; the stream should be a stream of 1!, 2!, 3!, 4!, etc. More specifically, it returns a stream with elements (1 2 6 24 ...)
   (define facts
      (cons-stream
       ...))

7. (HARD!) Define powers; the stream should be 1^1, 2^2, 3^3, ..., or, (1 4 16 64 ...). Of course, you cannot use the exponents procedure. I've given you a start, but you don't have to use it.
   (define powers (helper integers integers))
   (define (helper s t)
      (cons-stream
Constructing Streams Through Procedures

You'll find this the most natural way to construct streams, since it mirrors recursion so much. For example, to use a trite example,

```scheme
(define (integers-starting n) (cons-stream n (integers-starting (+ n 1))))
```

So `(integers-starting 1)` is a stream whose first element is 1, with a promise to evaluate `(integers-starting 2)`. The rules are similar to above; you still specify a first element, etc. Pretty simple? Let's try a few.

QUESTIONS

1. Define a procedure, `(list->stream ls)` that takes in a list and converts it into a stream.
   ```scheme
   (define (list->stream ls)
   ```

2. Define a procedure `(lists-starting n)` that takes in `n` and returns a stream containing `(n)`, `(n n+1)`, `(n n+1 n+2)`, etc. For example, `(lists-starting 1)` returns a stream containing `(1) (1 2) (1 2 3) (1 2 3 4) ...`
   ```scheme
   (define (lists-starting n)
   ```

3. Define a procedure `(chocolate name)` that takes in a name and returns a stream like so:
   ```scheme
   (chocolate 'chung) =>
   (chung really likes chocolate chung really really likes chocolate chung really really really likes chocolate ...)
   ```
   You'll want to use helper procedures.
   ```scheme
   (define (chocolate name)
   ```
Stream Processing

Sometimes you’ll be asked to write procedures that convert one given stream into another exerting a certain property.

QUESTIONS:

1. Define a procedure, \((\text{stream-censor } s \text{ replacements})\) that takes in a stream \(s\) and a table \(\text{replacements}\) and returns a stream with all the instances of all the \(\text{car}\) of entries in \(\text{replacements}\) replaced with the \(\text{cadr}\) of entries in \(\text{replacements}\):

\[
\text{stream-censor} \ (\text{hello you weirdo ...}) \ ((\text{you I-am}) \ (\text{weirdo an-idiot}))) \Rightarrow \\
(\text{hello I-am an-idiot ...})
\]

\[\text{(define (stream-censor s replacements)}\]

2. Define a procedure \((\text{make-alternating } s)\) that takes in a stream of positive numbers and alternates their signs. So \(\text{make-alternating ones} \Rightarrow (1 \ -1 \ 1 \ -1 \ ... )\) and \(\text{make-alternating integers} \Rightarrow (1 \ -2 \ 3 \ -4 \ ... )\). Assume \(s\) is an infinite stream.

\[\text{(define (make-alternating s)}\]

My Body's Floating Down the Muddy Stream

Now, more fun with streams!

MORE QUESTIONS

1. Given streams \(\text{ones}, \text{twos}, \text{threes},\) and \(\text{fours}\), write down the first ten elements of:

\(\text{(interleave ones (interleave twos (interleave threes fours)))}\)

2. Construct a stream \(\text{all-integers}\) that includes 0 and both the negative and positive integers.

\[\text{(define all-integers}\]

CS61A Summer 2010 – George Wang, Jonathan Kotker, Seshadri Mahalingam, Eric Tzeng, Steven Tang
3. Suppose we were foolish enough to try to implement `stream-accumulate`:

```scheme
(define (stream-accumulate combiner null-value s)
  (cond ((stream-null? s) null-value)
        (else (combiner
                (stream-car s)
                (stream-accumulate combiner null-value (stream-cdr s))))))
```

What happens when we do:

a. `(define foo (stream-accumulate + 0 integers))`

b. `(define bar (cons-stream 1 (stream-accumulate + 0 integers)))`

c. `(define baz (stream-accumulate
                  (lambda (x y) (cons-stream x y))
                  the-empty-stream integers))`

4. *SICP* ex. 3.68, page 341. If you understand this, you'll be fine.

MIDTERM 2 REVIEW

QUESTION 1. (What will Scheme print?)

What will Scheme print? If it will cause an error, simply write ERROR.

(a)

```scheme
> (equal? ((lambda (x) (x x x)) 7) '(7 7 7))
```

(b)

```scheme
> (define x (cons 1 'x))
> (define y x)
> (set! x 1)
> y
```
**QUESTION 2. (Box-'n'-pointers)**

Draw a box-and-pointer diagram for the following (the number of pairs in your final answer MATTERS). Also, fill in any blanks with the return value.

```
> (define a (list (list 3) 5))
> (define b (append a a))
> (set-car! (cdr b) (caddr b))
> (set-car! a (cons 3 4))
> a
__________
> b
__________
```

**QUESTION 3.** What are all the possible values of \(x\) after running the following Scheme code? If there can be deadlock, write DEADLOCK.

```
> (define x 8)
> (parallel-execute (lambda () (set! x (+ x 1)))

               (lambda () (set! x (if (even? x)
                   (set! x (+ x 5))
                   (+ x 50))))))
```

**QUESTION 4. (Metacircular evaluator)**

Suppose we type the following into mc-eval:

```
MCE> (define 'x (* x x))
```

This expression evaluates without error. What would be returned by the following expressions? (Relevant code is on page 10.)

(a) MCE> 'x

(b) MCE> x

(c) MCE> quote

(d) MCE> (quote x)
QUESTION 5. (Environment diagrams)

Draw an environment diagram for the following Scheme code. Also, fill in any blanks with the return value.

```
> (define foo
   (let ((x 3))
     (lambda ()
       (if (= x 1)
         x
       (* x (begin (set! x (- x 1)) (foo)))))))
> (foo)

_________
> (foo)

_________
> (foo)
```

---

CS61A Summer 2010 – George Wang, Jonathan Kotker, Seshadri Mahalingam, Eric Tzeng, Steven Tang
(define (mc-eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
          (make-procedure (lambda-parameters exp)
                          (lambda-body exp)
                          env))
        ((begin? exp) (eval-sequence (begin-actions exp) env))
        ((cond? exp) (mc-eval (cond->if exp) env))
        ((application? exp)
          (mc-apply (mc-eval (operator exp) env)
                    (list-of-values (operands exp) env)))
        (else
          (error "Unknown expression type -- EVAL" exp))))

(define (mc-apply procedure arguments)
  (cond ((primitive-procedure? procedure)
          (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
          (eval-sequence
            (procedure-body procedure)
            (extend-environment
              (procedure-parameters procedure)
              arguments
              (procedure-environment procedure))))
        (else
          ...))
(error
 "Unknown procedure type -- APPLY" procedure)))

(define (definition? exp)
  (tagged-list? exp 'define))

(define (definition-variable exp)
  (if (symbol? (cadr exp))
      (cadr exp)
      (caadr exp)))

(define (definition-value exp)
  (if (symbol? (cadr exp))
      (caddr exp)
      (make-lambda (cdadr exp)
                    (cddr exp)))))

(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
      (mc-eval (definition-value exp) env)
      env)
  'ok)