CS 61A
Summer 2019

Streams

Discussion 10: August 6, 2019 Solutions

1 Streams

In Python, we can use iterators to represent infinite sequences (for example, the generator for all natural numbers). However, Scheme does not support iterators. Let's see what happens when we try to use a Scheme list to represent an infinite sequence of natural numbers:

```
scm> (define (naturals n)
         (cons n (naturals (+ n 1))))
naturals
scm> (naturals 0)
Error: maximum recursion depth exceeded
```

Because `cons` is a regular procedure and both its operands must be evaluted before the pair is constructed, we cannot create an infinite sequence of integers using a Scheme list. Instead, our Scheme interpreter supports *streams*, which are *lazy* Scheme lists. The first element is represented explicitly, but the rest of the stream's elements are computed only when needed. Computing a value only when it's needed is also known as *lazy evaluation*.

```
scm> (define (naturals n)
         (cons-stream n (naturals (+ n 1))))
naturals
scm> (define nat (naturals 0))
nat
scm> (car nat)
0
scm> (cdr nat)
#[promise (not forced)]
scm> (car (cdr-stream nat))
1
scm> (car (cdr-stream (cdr-stream nat)))
2
```

We use the special form `cons-stream` to create a stream:

```
(cons-stream <operand1> <operand2>)
```

`cons-stream` is a special form because the second operand is not evaluated when evaluating the expression. To evaluate this expression, Scheme does the following:

1. Evaluate the first operand.
2. Construct a promise containing the second operand.
3. Return a pair containing the value of the first operand and the promise.

To actually get the rest of the stream, we must call `cdr-stream` on it to force the promise to be evaluated. Note that this argument is only evaluated once and is then stored in the promise; subsequent calls to `cdr-stream` returns the value without recomputing it. This allows us to efficiently work with infinite streams like the `naturals` example above. We can see this in action by using a non-pure function to compute the rest of the stream:

```
scm> (define (compute-rest n)
...>   (print 'evaluating!)
...>   (cons-stream n nil))
compute-rest
scm> (define s (cons-stream 0 (compute-rest 1)))
s
scm> (car (cdr-stream s))
evaluating!
1
scm> (car (cdr-stream s))
1
```

Here, the expression `compute-rest 1` is only evaluated the first time `cons-stream` is called, so the symbol `evaluating!` is only printed the first time.

When displaying a stream, the first element of the stream and the promise are displayed separated by a dot (this indicates that they are part of the same pair, with the promise as the `cdr`). If the value in the promise has not been evaluated by calling `cdr-stream`, we consider it to be not forced. Otherwise, we consider it forced.

```
scm> (define s (cons-stream 1 nil))
s
scm> s
(1 . #[promise (not forced)])
scm> (cdr-stream s) ; nil
()
scm> s
(1 . #[promise (forced)])
```

Streams are very similar to Scheme lists in that they are also recursive structures. Just like the `cdr` of a Scheme list is either another Scheme list or `nil`, the `cdr-stream` of a stream is either a stream or `nil`. The difference is that whereas both arguments to `cons` are evaluated upon calling `cons`, the second argument to `cons-stream` isn't evaluated until the first time that `cdr-stream` is called.

Here's a summary of what we just went over:

- `nil` is the empty stream
- `cons-stream` constructs a stream containing the value of the first operand and a promise to evaluate the second operand
- `car` returns the first element of the stream
- `cdr-stream` computes and returns the rest of stream

Video walkthrough

Questions

1.1 What would Scheme display?

As you work through these problems, remember that streams have two important components:

- Lazy evaluation – so the remainder of the stream isn't computed until explicitly requested.
- Memoization – so anything we compute won't be recomputed.

The examples here stretch these concepts to the limit. In most practical use cases, you may find you rarely need to redefine functions that compute the remainder of the stream.

```
scm> (define (has-even? s)
       (cond ((null? s) #f)
             ((even? (car s)) #t)
             (else (has-even? (cdr-stream s)))))
has-even?
scm> (define (f x) (* 3 x))
f
scm> (define nums (cons-stream 1 (cons-stream (f 3) (cons-stream (f 5) nil))))
nums
scm> nums
(1 . #[promise (not forced)])
scm> (cdr nums)
#[promise (not forced)]
scm> (cdr-stream nums)
(9 . #[promise (not forced)])
scm> nums
(1 . #[promise (forced)])
scm> (define (f x) (* 2 x))
f
scm> (cdr-stream nums)
(9 . #[promise (not forced)])
scm> (cdr-stream (cdr-stream nums))
(10 . #[promise (not forced)])
scm> (has-even? nums)
True
```

Video walkthrough

1.2 Using streams can be tricky! Compare the following two implementations of `filter-stream`, the first is a correct implementation whereas the second is wrong in some way. What's wrong with the second implementation?

```
; Correct
(define (filter-stream f s)
  (cond
    ((null? s) nil)
    ((f (car s)) (cons-stream (car s) (filter-stream f (cdr-stream s))))
    (else (filter-stream f (cdr-stream s)))))

; Incorrect
(define (filter-stream f s)
  (if (null? s) nil
    (let ((rest (filter-stream f (cdr-stream s))))
      (if (f (car s))
        (cons-stream (car s) rest)
        rest))))
```

Evaluating `rest` will result in infinite recursion if `s` is an infinite stream! In the body of `filter-stream`, `rest` is always computed before `cons-stream` can delay the evaluation.

Another way of thinking about this is that everything in the body of the `let` doesn't matter. All we will be doing is repeatedly doing the recursive call on `filter-stream`. Video walkthrough

1.3 Write a function `map-stream`, which takes a function `f` and a stream `s`. It returns a new stream which has all the elements from `s`, but with `f` applied to each one.

```
(define (map-stream f s)

  (if (null? s)
    nil
    (cons-stream (f (car s)) (map-stream f (cdr-stream s))))
```

It might help to also compare this to the version of `map` for regular (non-stream) Scheme lists:

```
(define (map f s)
  (if (null? s)
    nil
    (cons (f (car s)) (map f (cdr s)))))
```

Not too different, eh? The main change we've made is indicating we want to lazily evaluate the rest of our stream by using `cons-stream` instead of `cons`, and recognizing is a stream rather than a regular list by using `cdr-stream`.

```
scm> (define evens (map-stream (lambda (x) (* x 2)) nat))
evens
scm> (car (cdr-stream evens))
2
```

1.4 Write a function `slice` which takes in a stream `s`, a `start`, and an `end`. It should return a Scheme list that contains the elements of `s` between index `start` and `end`,

not including `end`. If the stream ends before `end`, you can return `nil`.

```
(define (slice s start end)

  (cond
    ((or (null? s) (= end 0)) nil)
    ((> start 0)
      (slice (cdr-stream s) (- start 1) (- end 1)))
    (else
      (cons (car s)
        (slice (cdr-stream s) (- start 1) (- end 1))))))
```

```
scm> (define nat (naturals 0)) ; See naturals procedure defined earlier
nat
scm> (slice nat 4 12)
(4 5 6 7 8 9 10 11)
```

1.5 We can even represent the sequence of all prime numbers as an infinite stream! Define a function `sieve`, which takes in a stream of increasing numbers and returns a stream containing only those numbers which are not multiples of an earlier number in the stream. We can define `primes` by sifting all natural numbers starting at 2. Look online for the **Sieve of Eratosthenes** if you need some inspiration.

```
(define (sieve s)

  (cons-stream
    (car s)
    (sieve (sift (car s) (cdr-stream s)))))
(define (sift prime s)
  (filter-stream
    (lambda (x) (not (= 0 (modulo x prime))))
    s))
(define primes
  (sieve (naturals 2)))
scm> (slice primes 0 10)
(2 3 5 7 11 13 17 19 23 29)
```

1.6 Since streams only evaluate the next element when they are needed, we can combine infinite streams together for interesting results! Use it to define a few of our favorite sequences. We've defined the function `combine-with` for you below, as well as an example of how to use it to define the stream of even numbers.

```
(define (combine-with f xs ys)
  (if (or (null? xs) (null? ys))
    nil
    (cons-stream
      (f (car xs) (car ys))
      (combine-with f (cdr-stream xs) (cdr-stream ys)))))
scm> (define evens (combine-with + (naturals 0) (naturals 0)))
evens
scm> (slice evens 0 10)
(0 2 4 6 8 10 12 14 16 18)
```

For these questions, you may use the `naturals` stream in addition to `combine-with`.

i.
```
(define factorials

  (cons-stream 1 (combine-with * (naturals 1) factorials)))
```
```
scm> (slice factorials 0 10)
(1 1 2 6 24 120 720 5040 40320 362880)
```

ii.
```
(define fibs

  (cons-stream 0
    (cons-stream 1
      (combine-with + fibs (cdr-stream fibs)))))
```
```
scm> (slice fibs 0 10)
(0 1 1 2 3 5 8 13 21 34)
```

iii. (Extra for practice) Write `exp`, which returns a stream where the nth term represents the degree-n polynomial expantion for e^x, which is $\sum_{i=0}^{n} x^i/i!$.

You may use `factorials` in addition to `combine-with` and `naturals` in your solution.

```
(define (exp x)
  (let ((terms (combine-with (lambda (a b) (/ (expt x a) b))
                             (cdr-stream (naturals 0))
                             (cdr-stream factorials))))
    (cons-stream 1 (combine-with + terms (exp x)))))
```

```
scm> (slice (exp 2) 0 5)
(1 3 5 6.333333333 7)
```

Extra Questions

1.1 In Discussion 9, we saw how to write a macro that creates a lambda function given an expression. While creating a parameter-less function might not seem that useful at first, it can be helpful in many cases when we don't want to immediately evaluate an expression.

Using the `make-lambda` macro you defined in Discussion 9, define `make-stream`, a macro which returns a pair of elements, where the second element is not evaluated until `cdr-stream` is called on it. Also define the procedure `cdr-stream`, which takes in a stream returned by `make-stream` and returns the result of evaluating the second element in the stream pair.

Unlike the streams we've seen in lecture and earlier in discussion, if you repeatedly call `cdr-stream` on a stream returned by `make-stream`, you may evaluate an expression multiple times.

```
(define-macro (make-stream first second)


(define-macro (make-stream first second)
        `(list ,first (make-lambda ,second)))

(define (cdr-stream stream)


(define (cdr-stream stream)
        ((car (cdr stream))))

scm> (define a (make-stream (print 1) (make-stream (print 2) nil)))
1
a
scm> (define b (cdr-stream a))
2
b
scm> (cdr-stream b)
()
```