CS61A Lecture 31

Wednesday, November 13, 2019

Announcements

- Homework 8 is due tomorrow, 11/14.
- Scheme project due 11/20, submit by 11/19 for early submission bonus point.
- Guerilla section on Saturday 11/16.
- Scheme Recursive Art contest due 12/2.

Sequence Operations

Map, filter and reduce are all higher-order functions that express sequence manipulation using compact expressions:

Example: sum all primes in an interval from a to b.

Example 1	**Example 2**
db65592a.png	`def sum_primes(a,b):` `    return sum(filter(is_prime, range(a,b)))`
Constant space	Constant space

The reason the latter runs in constant space as well is that Python's built-in sequence processing functions are very space efficient. `filter` returns a generator, not a list, and `sum` doesn't try to do the entire operation at once, but instead takes one value at a time, which negates the need for `filter` to return the entire list.

What about Scheme? Scheme doesn't have a range function, so let's build one.

```
(define (range a b)
    (if (>= a b)
        nil
        (cons a (range (+ a 1) b))
    )
)
```

Scheme also doesn't have a sum function:

```
(define (sum s)
    (reduce + s 0)
)
```

And to build the prime identifier (which isn't built into Python either, but we've built it in class before):

```
(define (prime? x)
    (and
        (> x 1)
        (null?
            (filter (lambda (y) (= 0 (remainder x y)))
            (range 2 x))
        )
    )
)
```

And finally:

```
(define (sum-primes a b)
    (sum (filter prime? (range a b)))
)
```

This Scheme version does the same thing as the Python one, but it’s both slower, and also uses more space: linear space. Speed can be chalked up to the fact that we are literally translating Scheme to Python and back, but the space issue is caused by the fact our range function is built at runtime, not at evaluation.

Python uses a feature called **lazy evaluation**. We could build this same feature into Scheme with a feature called **streams**.

Streams

Streams are lazy Scheme lists. They are lists, but the rest of the list is only computed only when needed:

```
> (car (cons 1 nil))
1
> (cdr (cons 1 nil))
()
```

Streams are functionally identical:

```
> (car (cons-stream 1 nil))
1
> (cdr-stream (cons-stream 1 nil))
()
```

The rest of the list is built but not evaluated until it is needed:

```
> (cons 1 (cons (/ 1 0) nil))
Error
> (cons-stream 1 (cons-stream (/ 1 0) nil))
(1 . #[promise (not forced)])
> (car (cons-stream 1 (cons-stream (/ 1 0) nil)))
1
> (cdr-stream (cons-stream 1 (cons-stream (/ 1 0) nil)))
Error
```

With this knowledge, we can redefine our `range` procedure:

```
(define (range a b)
    (if (>= a b)
        nil
        (cons-stream a (range (+ a 1) b))
    )
)
```

Integer Streams

An integer stream is a stream of consecutive integers. The rest of the stream is not yet computed when the stream is created.

```
(define (int-stream start)
    (cons-stream start (int-stream (+ start 1)))
)
```

We can use this in many ways:

```
> (prefix (int-stream 1) 5)
(1 2 3 4 5)
```

This is an example of an **infinite stream**. It goes on and on without stopping. We could've asked for the prefix until infinity!

This all may seem very similar to iterators, but the stream is not quite the same. It doesn't get "used up" like an iterator is.

Stream Processing

What can you do with a stream? You can square all the elements in a stream:

```
> (define (square-stream s)
.     (cons-stream (* (car s) (car s))
.     (square-stream (cdr-stream s)))
. )
> (int-stream 5)
(5 . #[promise (not forced)])
> (prefix (square-stream (int-stream 5)) 9)
(25 36 49 64)
```

Streams are by definition, recursively defined. We use this in many ways:

The rest of a constant stream is the constant stream:

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

Combining two streams by separating each into `car` and `cdr` :

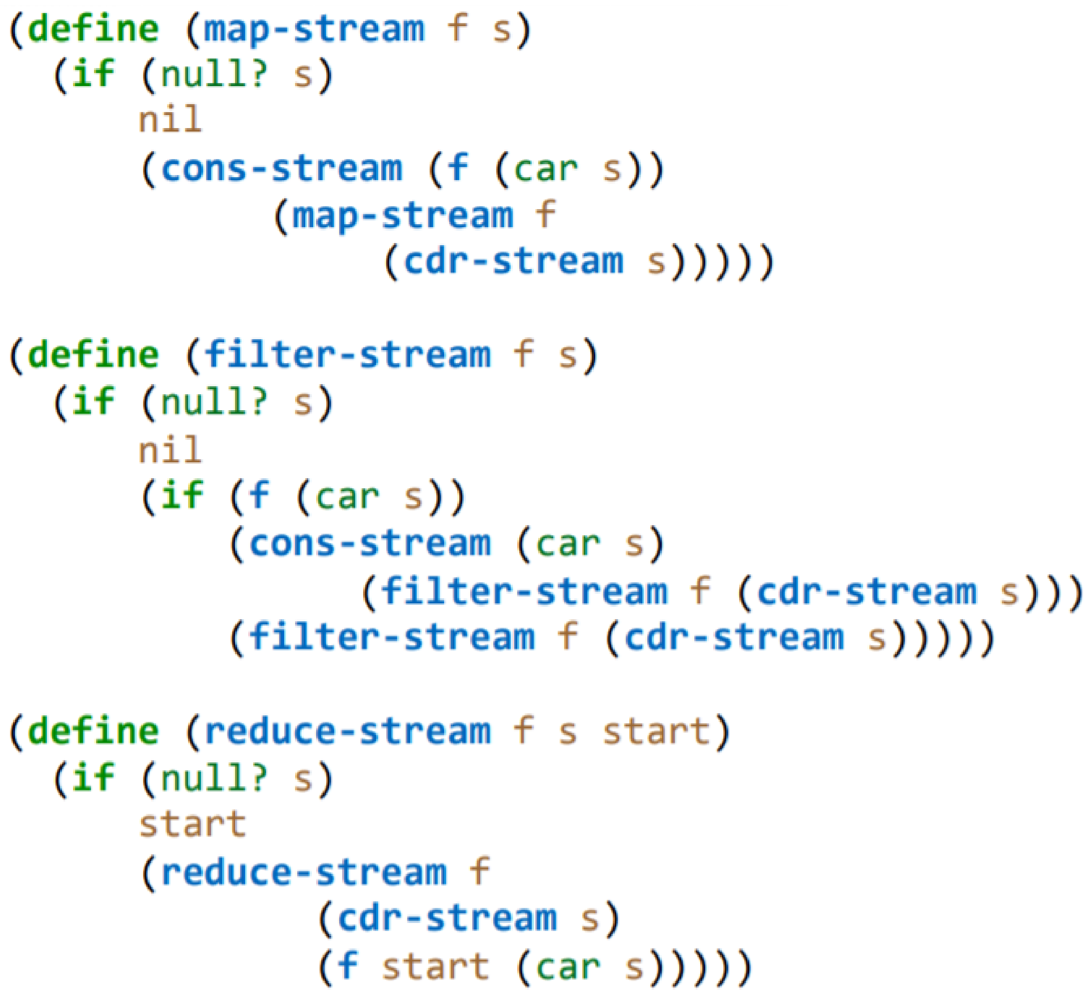
```
(define (add-streams s t)
    (cons-stream
        (+ (car s) (car t))
        (add-streams (cdr-stream s) (cdr-stream t))
    )
)
```

We can combine the `ints` stream with the `ones` stream:

```
(define ints (cons-stream 1 (add-streams ones ints)))
```

Higher-Order Functions on Streams

Anything you can do with a list, you can do with a stream. We can even implement `map` , `filter` and `reduce` with streams:

List	Stream
e8a9f437.png	

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

(define (filter-stream f s)
  (if (null? s)
      nil
      (if (f (car s))
          (cons-stream (car s)
                (filter-stream f (cdr-stream s)))
          (filter-stream f (cdr-stream s)))))

(define (reduce-stream f s start)
  (if (null? s)
      start
      (reduce-stream f
              (cdr-stream s)
              (f start (car s)))))
```

Prime Streams

For any prime kk, any larger prime must not be divisible by kk.

Thus, the stream of integers not divisible by any k \leq nk ≤ n is:

- The stream of integers not divisible by any k < nk < n.
- Filtered to remove any element divisible by nn.

This recurrence is called the Sieve of Eratosthenes. Let’s build it in Scheme.

```
(define (sieve s)
    (cons-stream
        (car s)
        (sieve
            (filter-stream
                (lambda (x) (< 0 (remainder x (car s) ) ) )
            )
        )
    )
)

(define primes (sieve (int-stream 2))
```