CSC324: Principles of Programming Languages

Lecture 0xB

- I will hold a quiz walkthrough on Monday during lab hours. It will be recorded if you are unable to make it, but feel free to join and ask questions.

- My office hours will remain through the break.

In previous classes...

- We saw how **lazy evaluation** of expressions lets us only evaluate the subexpressions that are ultimately used in the final computation
- We saw how an eager evaluation can be transformed into a lazy one by lifting the expression into a **thunk**

Today, we will extend the notion of laziness from expression evaluation to data structure design

A list!!!

By now, you know the data definition for a list like the back of your hand.

```
; a list of elements of type 'a' is either:
; - the empty list 'empty, OR
; - (cons x xs): where:
;   - x is of type 'a', and
;   - xs is a list of elements of type 'a'.
;
; Such a list is strict in its evaluation.
```

A list!!!

By now, you know the data definition for a list like the back of your hand.

```
; a list of elements of type 'a' is either:
; - the empty list 'empty, OR
; - (cons x xs): where:
;   - x is of type 'a', and
;   - xs is a list of elements of type 'a'.
;
; Such a list is strict in its evaluation.

; the input list needs to be fully constructed before it can be
; consumed by the map HOF.  This might be okay for this call to map...
(map add1 (cons 1 (cons 2 (cons 3 (... cons 99 (cons 100 'empty))))))
```

A list!!!

By now, you know the data definition for a list like the back of your hand.

```
; a list of elements of type 'a' is either:
; - the empty list 'empty, OR
; - (cons x xs): where:
;   - x is of type 'a', and
;   - xs is a list of elements of type 'a'.
;
; Such a list is strict in its evaluation.

; the input list needs to be fully constructed before it can be
; consumed by the map HOF.  This might be okay for this call to map...
(map add1 (cons 1 (cons 2 (cons 3 (... cons 99 (cons 100 'empty))))))

; ...but here, we construct the entire list even when we only want
; to get the first two elements out of it.
(take (cons 1 (cons 2 (cons 3 (... cons 99 (cons 100 'empty)))))
      2)
```

- We saw how to delay the evaluation of an expression by lifting it into a thunk.
- Given a (cons x xs), can we delay the evaluation of the tail of the list by similarly lifting it into a thunk?

```
;; A list of type 'a' is:
;; - 'empty, or
;; - (cons x xs), where:
;;   - x is an element of type 'a'
;;   - xs is a list of type 'a'
```

- We saw how to delay the evaluation of an expression by lifting it into a thunk.
- Given a (cons x xs), can we delay the evaluation of the tail of the list by similarly lifting it into a thunk?

```
;; A list of type 'a' is:
;; - 'empty, or
;; - (cons x xs), where:
;;   - x is an element of type 'a'
;;   - xs is a list of type 'a'
```

```
;; a stream of type 'a' is:
;; - a thunk producing 'empty, or:
;; - (scons x xs): where:
;;   - x is a thunk that produces an 'a'
;;   - xs is a thunk that produces a stream
```

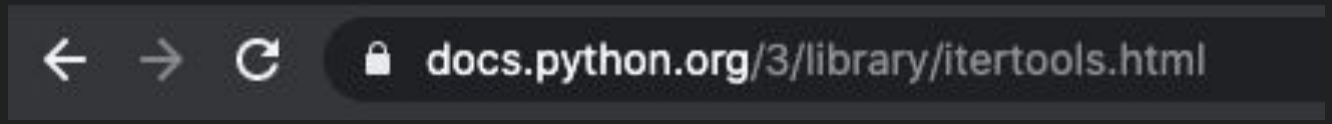
Infinite streams

- Unlike with lists, there's no restriction on when the thunk stored in xs needs to produce the 'empty value...in fact, there's no restriction that it need ever do so

- Such a stream, that will generate arbitrarily many elements if asked, is denoted an **infinite stream**.

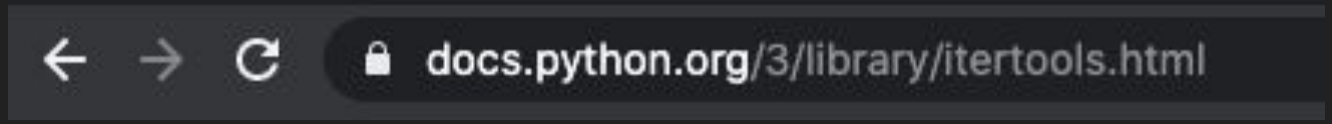
- Example in Haskell: `[1..]` is a lazy list that will produce an arbitrarily long sequence of ints...if you keep evaluating the tail!

Infinite streams in Haskell in terms of `x : xs`



Iterator	Arguments	Results	Example
`count()`	start, [step]	start, start+step, start+2*step, …	`count(10) --> 10 11 12 13 14 ...`
`cycle()`	p	p0, p1, … plast, p0, p1, …	`cycle('ABCD') --> A B C D A B C D ...`
`repeat()`	elem [,n]	elem, elem, elem, … endlessly or up to n times	`repeat(10, 3) --> 10 10 10`

Infinite streams in Racket in terms of (scons x xs)



Iterator	Arguments	Results	Example
`count()`	start, [step]	start, start+step, start+2*step, …	`count(10) --> 10 11 12 13 14 ...`
`cycle()`	p	p0, p1, … plast, p0, p1, …	`cycle('ABCD') --> A B C D A B C D ...`
`repeat()`	elem [,n]	elem, elem, elem, … endlessly or up to n times	`repeat(10, 3) --> 10 10 10`

Higher-order functions on sconses

```
(define (scons-map f s)
  (match s
    ['empty 'empty]
    [(cons x-thunk xs-thunk)
     (scons (f (x-thunk)) (scons-map f (xs-thunk)))]))

(define (scons-filter p s)
  (match s
    ['empty 'empty]
    [(cons x-thunk xs-thunk)
     (if (p (x-thunk))
         (scons (x-thunk) (scons-filter p (xs-thunk)))
         (scons-filter p (xs-thunk)))]))
```

Producing multiples of 3

```
(scons-filter (λ (i) (= (modulo i 3) 0)) (nats 2))
```

```
> (scons-take (scons-filter (λ (i) (= (modulo i 3) 0)) (nats 2)) 10)
'(3 6 9 12 15 18 21 24 27 30)
```

Filtering out multiples of i

```
(scons-filter (λ (i) (> (modulo i 3) 0)) (nats 2))
```

```
> (scons-take (scons-filter (λ (i) (> (modulo i 3) 0)) (nats 2)) 10)
'(2 4 5 7 8 10 11 13 14 16)
>
```

A prime sieve

- When we are passed a number, we want to filter out all "downstream" numbers that are not divisible by that number

- We receive 2? No even numbers allowed.
- We receive 3? No multiples of three allowed.
- Do we ever receive 4? *No, because we filtered out even numbers already!*

Go forth and break cryptography!

```
(define/match (sieve l)
  (('empty) empty)
  (((cons xt xst))
   (scons (xt) (sieve (scons-filter (λ (i) (> (modulo i (xt)) 0)) (xst))))))
```

```
> (scons-take (sieve (nats 2)) 10)
'(2 3 5 7 11 13 17 19 23 29)
>
```

Have a great break!

Course break: No classes 17, 19, 24 June		
Week 6 (26 June) *Mini-week: no class on 24 July*	Lecture slides	Assignment 1: Due by Friday, 26 June

- The ambiguous operator and local mutation