A Modularity Problem

Consider infinite sequences of integers, such as:
- powers of 2: 1, 2, 4, 8, 16, 32, 64, ...
- factorials: 1, 1, 2, 6, 24, 120, 720, ...
- Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
- primes: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 ...

Suppose we want answers to questions like the following:
- What are the first n elements?
- What is the first element greater than 100?
- What is the (0-based) index of the first element greater than 100?
- What is the first consecutive pair whose difference is more than 25?
- For which index i is the sum of elements 0 through i more than 1000?

Challenge: can we answer these questions in a modular way?
Non-Modular Haskell Solutions

```haskell
fibsPrefix :: Integer -> [Integer]
fibsPrefix num = gen 0 0 1
  where gen n a b =
    if n >= num then []
    else a : (gen (n + 1) b (a + b))

leastFibGt :: Integer -> Integer
leastFibGt lim = least 0 1
  where least a b = if a > lim then a
    else least b (a + b))

fibSumIndex :: Integer -> Integer
fibSumIndex lim = index 0 0 0 1
  where index i sum a b =
    if sum > lim then i
    else index (i+1) (sum+a) b (a + b)
```

A More Modular Approach: Infinite Lists

*Idea:* Separate the generation of the sequence elements from subsequent processing. Since we don’t know how many elements we’ll need, generate *all* of them — * lazily! *

```haskell
nats = genNats 0 where genNats n = n : genNats (n + 1)
  -- Can also be written: nats = [0..]
poss = tail nats -- the positive integers
  -- Can also be written: poss = [1..]
powers n = genPowers 1
  where genPowers x = x : (genPowers (n * x))
facts = genFacts 1 1
  where genFacts ans n = ans : (genFacts (n*ans) (n + 1))
fibs = genFibs 0 1
  where genFibs a b = a : (genFibs b (a + b))
```
### Processing Infinite Lists

*Note:* We assume the following functions are invoked only on infinite lists. This allows us to ignore the empty list as a base case! Each function *can* be extended to handle the empty list as well.

-- Returns a list of the first n elements of a given list.
\[
\text{take } n \ (x:xs) = \begin{cases} 
[] & \text{if } (n == 0) \\ x : (\text{take } (n-1) \ xs) & \text{else}
\end{cases}
\]

-- Returns first element satisfying predicate p
\[
\text{firstElem } p \ (x:xs) = \begin{cases} 
(x) & \text{if } (p \ x) \\ \text{firstElem } p \ xs & \text{else}
\end{cases}
\]

-- Returns first contiguous pair satisfying predicate p
\[
\text{firstPair } p \ (x:y:zs) = \\
\text{if } (p(x,y)) \text{ then } (x,y) \text{ else firstPair } p \ (y:zs)
\]

-- Returns (0-based) index of first elt satisfying pred p
\[
\text{index } p \ xs = \text{ind } 0 \ xs \\
\text{where ind } i \ (x : xs) = \\
\begin{cases} 
0 & \text{if } (p \ x) \\
(i+1) & \text{else}
\end{cases} \ (\text{ind } (i+1) \ xs)
\]

### Examples

\[
\text{take } 10 \ \text{fibs}
\]

\[
\text{firstElem } (\ \lambda \ x \rightarrow x > 100) \ \text{powers } 2
\]

\[
\text{index } (\ \lambda \ x \rightarrow x > 1000) \ \text{facts}
\]

\[
\text{firstPair } (\ (x,y) \rightarrow (y - x) > 25) \ \text{fibs}
\]
Scanning

Scanning accumulates the partial results of a foldl into a list.

\[
\text{scanl} :: (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow [a]
\]

\[
\text{scanl} \ f \ \text{ans} \ (x:xs) = \text{ans} : \text{scanl} \ f \ (f \ \text{ans} \ x) \ xs
\]

\[
\text{scanl} \ (+) \ 0 \ \text{powers} \ 2 \ -- \ be \ careful \ of \ initial \ zero!
\]

-- alternative definition of facts
\[
\text{facts} = \text{scanl} \ (*) \ 1 \ \text{ints}
\]

-- Like scanl, but uses first elt as initial answer
\[
\text{scanl1} :: (a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow [a]
\]

\[
\text{scanl1} \ f \ (x:xs) = \text{scanl} \ f \ x \ xs
\]

\[
\text{index} \ (\text{fn} \ s \rightarrow s > 1000) \ (\text{scanl1} \ (+) \ 0 \ \text{fibs})
\]

Higher-order Generation of Infinite Sequences

\[
\text{iterate} :: (a \rightarrow a) \rightarrow a \rightarrow [a]
\]

\[
\text{iterate} \ f \ x = x : \text{iterate} \ f \ (f \ x)
\]

-- another way to generate the nats
\[
\text{nats} = \text{iterate} \ (1 +) \ 0
\]

\[
\text{iterate2} :: (a \rightarrow a \rightarrow a) \rightarrow a \rightarrow a \rightarrow [a]
\]

\[
\text{iterate2} \ f \ x1 \ x2 = x1 : \text{iterate2} \ f \ x2 \ (f \ x1 \ x2)
\]

-- another way to generate the fibs
\[
\text{fibs} = \text{iterate2} \ (+) \ 0 \ 1
\]

\[
\text{iteratei} :: (\text{Integer} \rightarrow a \rightarrow a) \rightarrow \text{Integer} \rightarrow a \rightarrow [a]
\]

\[
\text{iteratei} \ f \ n \ x = x : \text{iteratei} \ f \ (n + 1) \ (f \ n \ x)
\]

-- a third way to generate the facts
\[
\text{facts} = \text{iteratei} \ (*) \ 1 \ 1
\]
Cyclic Definitions of Infinite Sequences

```haskell
ones = 1 : ones

-- a third way to generate the nats
nats = 0 : (map (1 +) nats)

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = (f x y) : (zipWith f xs ys)

-- a fourth way to generate the nats
nats = 0 : (zipWith (+) ones nats)

-- a fourth way to generate the facts
facts = 1 : (zipWith (*) poss facts)

-- a third way to generate the fibs
fibs = 0 : 1 : (zipWith (+) fibs (tail fibs))
```

Generating Primes

Idea: use the “sieve of Eratosthenes”

```haskell
sieve (x:xs) =
  x : (sieve (filter (\ y -> (rem y x) /= 0) xs))

primes = sieve (tail (tail nats)) -- start sieving at 2
```

Not only does this give an infinite list of primes, it does so efficiently by avoiding unnecessary divisions.

For more examples of lazy lists in Haskell, see Chapter 17 of Simon Thompson’s book *Haskell: The Craft of Functional Programming.*
Lazy Trees

Can use laziness to perform a two-pass tree walk in a single pass:

```haskell
data Tree a = Leaf | Node (Tree a) a (Tree a)

addMax tr = tr'
  where (tr', m) = walk tr
    walk Leaf = (Leaf, 0)
    walk (Node l v r) = (Node l' (m + v) r',
     max3 ml v mr)
      where (l',ml) = walk l
      (r',mr) = walk r

max3 a b c = max a (max b c)
```

See Hughes’s paper “Why Functional Programming Matters” for compelling lazy game tree example.

Streams : Lazy Lists for Scheme

```scheme
(cons-stream Ehead Etail)
Return a (potentially infinite) stream whose head is the value of Ehead and whose tail is the value of Etail. The evaluation of Etail is delayed until it is needed.

(head Estream)
Return the head element of the stream value of Estream.

(tail Estream)
Return the tail of the stream value of Estream. This forces the computation of the delayed tail expression.

(stream-null? Estream)
Return true if Estream is the empty stream and false otherwise.
```

the-empty-stream
The empty stream
Stream Examples I

(define ints-from
  (lambda (n)
    (cons-stream n (ints-from (+ n 1)))))) ; No base case!

;; Converts first n elements of infinite stream to a list
(define take
  (lambda (n str)
    (if (= n 0)
        '()
        (cons (head str) (take (- n 1) (tail str))))))

(define ones (cons-stream 1 ones))

(define map-stream
  (lambda (f str)
    (cons-stream (f (head str))
                 (map-stream f (tail str))))))

(define nats (cons-stream 0 (map-stream (lambda (x) (+ x 1)) nats)))

Stream Examples II

(define map2-streams
  (lambda (f str1 str2)
    (cons-stream (f (head str1) (head str2))
                 (map2-streams f (tail str1) (tail str2)))))

(define fibs
  (cons-stream 0
               (cons-stream 1
                            (map2-streams + fibs (tail fibs))))))

• Can similarly translate other lazy list examples from Haskell to Scheme

• See Section 3.5 of SICP for lots of examples.
Implementing Lazy Data in a Strict Language

- **Idea** -- use memoizing promises to implement lazy lists in Scheme:

  ```scheme
  (cons-stream E1 E2) is syntactic sugar for (cons E1 (delay E2))
  (define (head s) (car s))
  (define (tail s) (force (cdr s)))
  (define (null-stream? s) (null? s))
  (define the-empty-stream '())
  ```

- Can generalize this idea to handle infinite trees.
- Can similarly implement lazy lists in ML.
- Lazy data is very helpful, but sometimes need even more laziness (e.g. translating addMax example to Scheme or ML).

Java Enumerations

Like streams, Java’s enumerations can be conceptually infinite. For example:

```java
public class FibEnumeration implements Enumeration {
    private int a, b;
    public FibEnumeration () { a = 0; b = 1; }
    public boolean hasMoreElements () { return true; }
    public Object nextElement () {
        int old_a = a;
        a = b;
        b = old_a + b;
        return new Integer(old_a);
        // Convert int to Integer to satisfy type of nextElement
    }
}
```

- Unlike streams, enumerations are not persistent; can’t hold on to a snapshot of the enumeration at given point in time without copying it.
- While lazy data is easy to adapt to trees, enumerations are inherently linear.