

Principles of Programming Languages

Generators, Lazy Lists

In this section, we continue our analysis of control structures and the relation it has with delayed computation. We first return to the notion of coroutines, which we met in Section 4.1 in JavaScript, and attempt to simulate coroutines in the programming language *L5*.

We then refine our understanding of such coroutines by describing them as an abstract data type which involves closures as delayed computations. We call this type a **lazy list** (also known as *streams*). We present a programming style based on lazy lists which has become known as the functional reactive programming paradigm.

Consider the following generator example in JavaScript, using the language constructs of `function*`, `yield` and the `Iterator` interface with the `IteratorResult` data type:

Constructing Coroutines with Delayed Computation

```
function f1(n) { return n + 1; }  
function f2(n) { return n + 2; }  
function f3(n) { return n + 3; }  
  
function* gen1(n) {  
  yield f1(n);  
  yield f2(n);  
  yield f3(n);  
}
```

Constructing Coroutines with Delayed Computation

```
const g11 = gen1(0);  
g11.next(); // { value: 1, done: false }  
g11.next(); // { value: 2, done: false }  
g11.next(); // { value: 3, done: false }  
g11.next(); // { value: undefined, done: true }
```

Let us build a function which behaves in a similar manner in *L5*.

Using `(lambda () ...)` to Delay Computation

The basis of the model is to use `(lambda () ...)` to delay computation we do not want to execute immediately. This is similar to the way we proceeded when defining the CPS approach: delayed computation is wrapped in closures.

There are two ingredients to coroutines in JavaScript:

- The `Iterator` interface and the `IteratorResult` datatype
- The `function*` and `yield` mechanism

Let us model iterators in our functional language *L5*.

An `IteratorResult` in JavaScript has two fields: `value` and `done`.

```
interface IteratorResult<T> { value: T; done: boolean; }  
interface Iterator<T> { next(): IteratorResult<T>; }
```

Through the usage of the method `next()`, we understand that objects which implement the **Iterator** interface are mutable.

For example, the state of the generator `g11` in the example above is updated, so that each time we call `g11.next()` we obtain a different value. Each time `next()` is invoked, the generator advances to the next step of the computation - and it stops before completing the rest of the computation, until a new `next()` invocation.

To adapt this mechanism to a functional language, we must disentangle the mutation from the aspect of delaying computation. We thus define a data type for iterators which is immutable. We adopt the general approach of *persistent mutable datatypes* in Functional Programming which we discussed in Section 1.4 with the Stack example.

The key ingredient of this approach is to define each mutator function as a constructor. We thus obtain an iterator data type with two constructors: **yield** and **next**:

A Functional Version of Iterators

```
;; An iterator is either the "done" iterator or a pair with  
;; the current value of the iterator and a continuation which  
;; returns the next value of the iterator when evaluated.  
;; Type Iterator<T> = done | Pair(T, [Empty -> Iterator<T>])
```

A Functional Version of Iterators

```
;; Purpose: The yield Iterator constructor takes 2 arguments:  
;; - The value to return to the caller  
;; - The continuation to be executed when the Iterator  
;;   is resumed (by invoking next())  
;; Signature: yield(result, continuation)  
;; Type: [T * [Empty -> Iterator<T>] -> Iterator<T>]  
(define yield  
  (lambda (res cont)  
    (cons res cont)))
```

A Functional Version of Iterators

```
;; An iterator has 2 fields:  
;; - Next: execute the next step of the computation of the  
;;       generator  
;; - Value: access the current value of the generator  
;; And a state predicate  
;; - done?: determines whether the generator has reached  
;;       the end of the computation.
```

A Functional Version of Iterators

*;; The accessors are iter->next, iter->value and iter->done?
;; iter->next computes the next step of the iterator.*

A Functional Version of Iterators

```
;; Iter->next is a constructor - it returns the value of the  
;; next iterator by invoking the continuation.  
;; Once an iterator is done, next keeps returning done.  
;; Else - next computes the next step of the iterator  
;; by invoking the continuation.  
(define iter->next  
  (lambda (iter)  
    (if (iter->done? iter)  
        iter  
        (let ((cont (cdr iter)))  
          (if (eq? cont 'done)  
              cont  
              (cont)))))))
```

A Functional Version of Iterators

```
;; The value of a done iterator is 'done.  
;; Else read it from the pair.  
(define iter->value  
  (lambda (iter)  
    (if (iter->done? iter)  
        iter  
        (car iter))))  
  
(define iter->done?  
  (lambda (iter)  
    (eq? iter 'done)))
```

Using the Iterator ADT to Construct a Generator in L5

Let us now use the iterator datatype we just defined to implement in L5 the coroutine **g2** which mimics the same behavior as **g1** in JavaScript.

The implementation is a bit unpleasant because we need to explicitly add `(lambda () ...)` to delay the next steps of the computation after a **yield**.

Using the Iterator ADT to Construct a Generator in L5

```
(define f1
  (lambda (n)
    (print (+ n 1))
    (+ n 1)))

(define f2
  (lambda (n)
    (print (+ n 2))
    (+ n 2)))

(define f3
  (lambda (n)
    (print (+ n 3))
    (+ n 3)))
```

Using the Iterator ADT to Construct a Generator in L5

;; Consider the following function g1

```
(define g1
  (lambda (n)
    (f1 n)
    (f2 n)
    (f3 n)))
```

*;; Coroutine version of g1 with interruptions after
;; each sub-function*

```
(define g2
  (lambda (n)
    (yield (f1 n)
      (lambda ()
        (yield (f2 n)
          (lambda ()
            (yield (f3 n)
              'done))))))))
```

Using the Iterator ADT to Construct a Generator in L5

```
;; Invoke g2 and resume it  
(let ((iter (g2 0)))  
  ;; Iter is of the form (res . continuation)  
  (print (iter->value iter))  
  (iter->next iter))  
;; 112'(2 . #<procedure:...c/coroutine1.rkt:101:20>)
```

The execution yields the following process:

Using the Iterator ADT to Construct a Generator in L5

- When `(g2 0)` is executed, we execute `(yield (f1 0) ...)` - which as a side-effect prints **1**. The local variable **iter** is now bound to the value returned by this **yield**.

- We then print `(iter->value iter)` which prints the current value of the iterator, which is 1.
- We then resume the coroutine by invoking `(iter->next iter)` - this executes `(yield 2 ...)` - which as a side effect, prints 2.

Using the Iterator ADT to Construct a Generator in L5

As was the case in JavaScript, we can use the Iterator Abstract Data Type to model infinite sequences.

```
(define integers
  (lambda (from)
    (letrec ((loop (lambda (n)
                     (yield n (lambda () (loop (+ n 1)))))))
      (loop from))))
```

In JavaScript, we used the generalized loop construct **for x of gen** to conveniently consume the values produced by a generator. Similarly, we define functions to ease the consumption of *L5* generators.

Consuming Generators

```
;; Purpose: add one item at the end of a list.  
;; Type: [List(T) * T -> List(T)]  
(define concat  
  (lambda (elts item)  
    (append elts (list item))))
```

Consuming Generators

```
;; Purpose: convert a finite iterator to a list.  
;; Warning: would enter in an infinite loop on an  
;;         infinite generator.  
;; Type: [Iterator(T) -> List(T)]  
(define iter->list  
  (lambda (iter)  
    (letrec ((loop (lambda (iter res)  
                      (if (iter->done? iter)  
                          res  
                          (loop (iter->next iter)  
                                (concat res  
                                         (iter->value iter)))))))  
      (loop iter (list))))))
```

Consuming Generators

```
;; Purpose: return the first n elements generated  
;;           by an iterator as a list.  
;; Type: [Iterator(T) * number -> List(T)]  
;; Returns a list of up to n elements - can be less if  
;; the generator is done before.  
;; On a done iterator, returns an empty list.  
(define iter->take  
  (lambda (iter n)  
    (letrec ((loop (lambda (iter n res)  
                      (if (<= n 0)  
                          res  
                          (if (iter->done? iter)  
                              res  
                              (loop (iter->next iter)  
                                    (- n 1)  
                                    (concat res  
                                              (iter->value iter))))))))  
      (loop iter n (list)))))
```

Consuming Generators

```
;; Purpose: return an iterator that yields at most n elements  
;;           from another (possibly longer or infinite)  
;;           iterator.  
;; Type: [Iterator(T) -> Iterator(T)]  
(define iter->take*  
  (lambda (iter n)  
    (if (= n 0)  
        'done  
        (yield (iter->value iter)  
                (lambda ()  
                  (iter->take* (iter->next iter)  
                               (- n 1)))))))
```

With these functions, we can easily consume elements from a generator:

```
(iter->list (g2 0))  
;; '(1 2 3)
```

```
(iter->take (integers 0) 10)  
;; '(0 1 2 3 4 5 6 7 8 9)
```

One can refer to generators as *lists in comprehension*, and accordingly define higher-order functions similar to the **map/filter/reduce** family we explored in Chapter 1 for lists.

Higher-Order Generator Functions

```
;; Purpose: apply the procedure proc on each of the elements  
;;           of a generator. This consumes all the elements  
;;           of the generator.  
;; Type: [(T1 -> T2) * Iterator(T1) -> Void]  
(define iter->for-each  
  (lambda (proc iter)  
    (cond ((iter->done? iter) iter)  
          (else (proc (iter->value iter))  
                  (iter->for-each proc (iter->next iter))))))
```

Higher-Order Generator Functions

```
;; Purpose: create a new generator returning (proc x)  
;;           for each item x generated by iter.  
;; Type: [(T1 -> T2) * Iterator(T1) -> Iterator(T2)]  
(define iter->map  
  (lambda (proc iter)  
    (if (iter->done? iter)  
        iter  
        (yield (proc (iter->value iter))  
                (lambda ()  
                  (iter->map proc (iter->next iter)))))))
```

`iter->map` constructs a new generator on the basis of an existing one. The resulting generator produces the same number of items as the original one.

```
(iter->take (iter->map (lambda (x) (* x x)) (integers 0)) 10)  
;; '(0 1 4 9 16 25 36 49 64 81)
```

```
(iter->for-each (lambda (i) (print i))  
               (iter->take* (integers 0) 10))  
;; 0123456789'done
```

We could design a slight variant of the generators abstract data type using mutation. This gives a result which behaves in a manner more similar to JavaScript generators.

The main idea of this model is that the `next()` operation must now actually mutate the generator. Hence, the generator must hold a state - which remembers at any given stage the current value and the continuation needed to produce the next state.

The following function illustrates how we obtain this behavior.

A Mutable Alternative Implementation of Generators

In this function, the `make-generator` constructor creates a closure which encapsulates the state of the generator - which is a generator as described above. The closure we construct exposes an interface for a `'next!` operation and a value accessor operation. The client interfaces with the generator by sending a message with the name of the operation it wants to run.

A Mutable Alternative Implementation of Generators

```
(define make-generator
  (lambda (thunk)
    (let ((gen (yield 'init thunk)))
      (lambda (op)
        (cond ((eq? op 'next!)
               (set! gen (iter->next gen))
               (iter->value gen))
              ((eq? op 'value)
               (iter->value gen))
              (else (error "Unknown operation"))))))))
```

A Mutable Alternative Implementation of Generators

*;; Same as g1 with interruptions and resume with mutable generator
;; g3 behaves like the value of function* in JS*

```
(define g3
  (make-generator
    (lambda ()
      (yield (f1 n)
        (lambda ()
          (yield (f2 n)
            (lambda ()
              (yield (f3 n)
                'done))))))))))
```

A Mutable Alternative Implementation of Generators

```
(define gen (g3 0))  
(gen 'next!) ;; 11  
(gen 'next!) ;; 22  
(gen 'value) ;; 2  
(gen 'next!) ;; 33  
(gen 'next!) ;; 'done
```

We derived above the definition of Generators in *L5* by mimicking the behavior of coroutines in JavaScript and attempting to derive a functional implementation of coroutines using delayed computation with (`lambda () ...`) as the delaying operator and functional mutable data structures.

Let us revisit this derivation of functional generators with one more model, based on type analysis. We obtain an almost identical result - the focus here is on describing generators as sequences by describing their data type.

Reminder: The List Data Type in Scheme

We define the regular list data type through an inductive type definition:

$$\text{List}(T) = \text{Empty} \mid \text{Pair}(T, \text{List}(T))$$

Reminder: The List Data Type in Scheme

This means that the set of all `List(T)` values contains:

- The Empty list (which we write: `' ()`)
- Non-empty lists - which are made up of a pair with a first element of type `T` and a second element of type `List(T)`.

Reminder: The List Data Type in Scheme

We define a functional interface on this data type consisting a value constructor (**cons**), two accessors (**first** and **rest** also known as **car** and **cdr**) and two type predicates **empty?** and **list?**.

We derive a similar type definition for **lazy lists**
– that is, lists whose first element is known and
the rest of the list is a delayed computation
which describes how to compute the rest of the
list.

Adapting the List Data Type to Lazy-Lists

The list is not built **in extension** by filling it in memory with its items, but **in comprehension** by describing which elements belong to the list as a computation instead of listing the elements that belong to the list.

Adapting the List Data Type to Lazy-Lists

To describe this idea as a data type definition, we use the following type equation defining the new **Lzl** data type (which stands for *Lazy List*):

$$\text{Lzl}(T) = \text{Empty-Lzl} \mid \text{Pair}(T, [\text{Empty} \rightarrow \text{Lzl}(T)])$$

In words, this means a value belongs to the type $\text{Lzl}(T)$ either if it is the empty lazy list (Empty-Lzl) or if it is a pair with a first element of type T and the second element is a continuation, which when computed produces a $\text{Lzl}(T)$ value.

We derive from this type definition the functional interface to manipulate values that belong to the type:

Adapting the List Data Type to Lazy-Lists

```
;; The empty lazy list value (a singleton datatype)  
(define empty-lzl '())
```

```
;; Purpose: Value constructor for non-empty lazy-list values  
;; Type: [T * [Empty -> LZL(T)] -> LZT(T)]  
(define cons-lzl cons)
```

```
;; Accessors  
;; Type: [LZL(T) -> T]  
;; Precondition: Input is non-empty  
(define head car)
```

Adapting the List Data Type to Lazy-Lists

```
;; Type: [LZL(T) -> LZL(T)]  
;; Precondition: Input is non-empty  
;; Note that this *executes* the continuation  
(define tail  
  (lambda (lzl)  
    ((cdr lzl))))  
  
;; Type predicate  
(define empty-lzl? empty?)
```

Adapting the List Data Type to Lazy-Lists

This definition parallels that of regular lists - as recursive data types defined inductively. The only difference is that the tail of the list is *delayed* by using a `(lambda () ...)` wrapper.

Adapting the List Data Type to Lazy-Lists

Explore the behavior of these definitions on simple values:

```
(define lzl1 empty-lzl)
(define lzl2 (cons-lzl 1 (lambda () lzl1)))
(define lzl3 (cons-lzl 2 (lambda () lzl2)))

(head lzl3) ;; 2
(tail lzl3) ;; '(1 . #<procedure>)
(head (tail lzl3)) ;; 1
(empty-lzl? (tail (tail lzl3))) ;; #t
```

Because the tail of lazy lists is computed, we can build inductively infinite sequences. In contrast to regular recursive functions, lazy lists can be defined inductively **without** a base case to terminate the recursion.

Adapting the List Data Type to Lazy-Lists

```
;; Signature: integers-from(n)  
;; Type: [number -> LZL(number)]  
(define integers-from  
  (lambda (n)  
    (cons-lzl n (lambda () (integers-from (+ n 1))))))  
  
(define ints (integers-from 0))  
ints ;; '(0 . #<procedure>)
```

To manipulate easily `Lz1` values, we define an extended functional interface - which is equivalent to the loops that we used in JavaScript.

Manipulation of LzL Values

```
;; Signature: take(lz-lst,n)  
;; Type: [LzL * Number -> List]  
;; If n > length(lz-lst) then the result  
;; is lz-lst as a List  
(define take  
  (lambda (lz-lst n)  
    (if (or (= n 0) (empty-lzl? lz-lst))  
        empty  
        (cons (head lz-lst)  
              (take (tail lz-lst) (- n 1)))))))
```

Manipulation of LzL Values

```
; Signature: nth(lz-lst,n)  
;; Type: [LzL*Number -> T]  
;; Pre-condition: n < length(lz-lst)  
(define nth  
  (lambda (lz-lst n)  
    (if (= n 0)  
        (head lz-lst)  
        (nth (tail lz-lst) (- n 1)))))
```

Observe that when we evaluate the following calls, the successive steps of the expansion of the integers-from lazy-list are repeated:

```
(take ints 10) ;; '(0 1 2 3 4 5 6 7 8 9)
(take ints 5)  ;; '(0 1 2 3 4)
```

Besides representing sequences in comprehension, the model of lazy lists is a useful abstraction to describe recursive processes.

Let us start with a simple idea: repeating a value an unbounded number of times.

```
(define ones (cons-lzl 1 (lambda () ones)))
```

```
(take ones 7) ;; '(1 1 1 1 1 1 1)
```

This computation is interesting because it is a form of infinite loop which is controlled by the caller.

Let us now describe a list of values which are built incrementally on top of each other. Let us build the list of all factorial values.

We start with a simple definition:

```
(define fact
  (lambda (n)
    (if (= n 1)
        1
        (* n (fact (- n 1))))))
```

```
;; Type: [Number -> LzL(Number)]
```

```
(define facts-from
  (lambda (n)
    (cons-lzl (fact n)
              (lambda () (facts-from (+ n 1))))))
```

When we observe the computation, we realize though, that if we know the prefix of this lazy-list, we can compute the next element in a faster way than by invoking `(fact (+ n 1))` - as long as we have access to the first element when we compute the tail. We derive the following pattern of lazy-list construction:

Computing with Lazy-Lists

```
(define facts-gen
  (lambda ()
    (letrec ((loop
              (lambda (n fact-n)
                (cons-lzl fact-n
                          (lambda () (loop (+ n 1)
                                           (* (+ n 1) fact-n)))))))
      (loop 1 1))))

(take (facts-gen) 6) ;; '(1 2 6 24 120 720)
```

A good way to think about **facts-gen** is as the *unrolling of the **fact** computation*. It returns the list of the values of a recursive computation. The structure of the function is typical: the local function loop *remembers* the last computed value **fact-n** and passes it to the delayed computation in the tail.

Lazy list composition functions operate over `Lz1` values and return new `Lz1` values.

Let us consider the `lzl-add` operator: given two `Lzl(number)`, it returns the `Lzl` of the sum of their respective values.

Composition of Lazy Lists

```
;; Signature: lzl-add(lz1,lz2)  
;; Type: [LzL(Number) * LzL(Number) -> LzL(number)]  
(define lzl-add  
  (lambda (lz1 lz2)  
    (cond ((empty-lzl? lz1) lz2)  
          ((empty-lzl? lz2) lz1)  
          (else (cons-lzl (+ (head lz1) (head lz2))  
                           (lambda ()  
                             (lzl-add (tail lz1) (tail lz2)))))))
```

This operator allows us to re-define the sequence of integers using lazy-list addition:

```
(define integers  
  (cons-lzl 0  
            (lambda () (lzl-add ones integers))))
```

Similarly, let us define the sequence of Fibonacci numbers using lazy-list addition:

```
(define fib-numbers
  (cons-lzl 0
            (lambda () (cons-lzl 1
                                  (lambda ()
                                    (lzl-add (tail fib-numbers)
                                             fib-numbers)))))))
```

What is remarkable about these definitions is that they replace loops and recursive functions (the inner functions **loop** in the examples above) with recursive data flow: the **fib-numbers** list is built incrementally from the prefix of the **fib-numbers** list.

Append and Interleave of Lazy Lists

Recall the definition of the **append** function for regular lists to return a list which contains all the elements in **l1** followed by those in **l2**:

```
;; Type: [List(T) * List(T) -> List(T)]  
(define append  
  (lambda (l1 l2)  
    (if (empty? l1)  
        l2  
        (cons (car l1)  
              (append (cdr l1) l2))))))
```

Append and Interleave of Lazy Lists

Let us define a similar function for lazy lists:

```
;; Signature: lzl-append(lz1, lz2)  
;; Type: [Lzl(T) * Lzl(T) -> Lzl(T)]  
(define lzl-append  
  (lambda (lz1 lz2)  
    (if (empty-lzl? lz1)  
        lz2  
        (cons-lzl (head lz1)  
                   (lambda ()  
                     (lzl-append (tail lz1) lz2)))))))
```

Observe the elements of the appended list: we see that all elements of the first lazy-list come before the second lazy-list. What if the first list is infinite? There is no way to reach the second list.

This version does not satisfy a fundamental property of lazy-list functions: **Every finite part of a lazy-list “depends” on at most a finite part of the lazy-list.**

Therefore, when dealing with possibly infinite lists, **append** is replaced by a different function we call **interleave()**. **interleave()** returns the elements of the lazy-lists in a way that guarantees that every element of the lazy-lists is reached within finite time:

Append and Interleave of Lazy Lists

```
;; Signature: interleave(lz1, lz2)  
;; Type: [Lzl(T) *Lzl(T) -> Lzl(T)]  
(define interleave  
  (lambda (lz1 lz2)  
    (if (empty-lzl? lz1)  
        lz2  
        (cons-lzl (head lz1)  
                   (lambda () (interleave lz2 (tail lz1)))))))  
  
(take (lzl-append (integers-from 100) fibs) 7)  
;; '(100 101 102 103 104 105 106)  
  
(take (interleave (integers-from 100) fibs) 7)  
;; '(100 0 101 1 102 1 103)
```

Looking at `Lzl` values as sequences, it is natural to adapt the sequence interface of `map/filter/reduce` to this data type as well. This perspective allows us to define interesting data dependencies to capture recursive relations in a compact way.

Higher Order Lazy List Functions

```
;; Signature: lzl-map(f, lz)
;; Type: [[T1 -> T2] * Lzl(T1) -> Lzl(T2)]
(define lzl-map
  (lambda (f lzl)
    (if (empty-lzl? lzl)
        lzl
        (cons-lzl (f (head lzl))
                   (lambda () (lzl-map f (tail lzl)))))))

(take (lzl-map (lambda (x) (* x x)) ints) 5)
;; '(0 1 4 9 16)
```

Higher Order Lazy List Functions

```
;; Signature: lzl-filter(p,lz)  
;; Type: [[T1 -> Boolean] * Lzl(T1) -> LzL(T1)]  
(define lzl-filter  
  (lambda (p lzl)  
    (cond ((empty-lzl? lzl) lzl)  
          ((p (head lzl))  
           (cons-lzl (head lzl)  
                     (lambda ()  
                       (lzl-filter p (tail lzl))))))  
          (else (lzl-filter p (tail lzl)))))
```

Let us explore ways to define complex recursive definitions using these higher-order functions.

We develop here a generator of the sequence of prime numbers based on the sieve method:

Higher Order Lazy List Functions

```
(define divisible?  
  (lambda (x y)  
    (= (remainder x y) 0)))
```

```
(define no-sevens (lzl-filter (lambda (x)  
                                (not (divisible? x 7))) ints))
```

```
(nth no-sevens 100) ;The 100th integer not divisible by 7:  
;; 117
```

```
;; lazy-list scaling: return (c*x for x in lzl)
```

```
;; Signature: lzl-scale(c, lzl)
```

```
;; Type: [Number * Lzl(Number) -> Lzl(Number)]
```

```
(define lzl-scale  
  (lambda (c lzl)  
    (lzl-map (lambda (x) (* x c)) lzl)))
```

Higher Order Lazy List Functions

In a way similar to which we defined Fibonacci numbers as a recursive equation involving the **fib-numbers** sequence, we define the generator of the powers of 2 as follows:

```
;; The lazy-list of powers of 2:  
(define double  
  (cons-lzl 1 (lambda () (lzl-scale 2 double))))  
  
(take double 7) ;; '(1 2 4 8 16 32 64)
```

Recall the integers lazy-list creation function:

```
(define integers-from  
  (lambda (n)  
    (cons-lzl n (lambda () (integers-from (+ n 1))))))
```

It can be re-written as follows, where we explicitly abstract the step operation which makes us proceed from one element in the list to the next as the operation

$$(\text{lambda } (k) (+ k 1)):$$

```
(define integers-from
  (lambda (n)
    (cons-lzl n (lambda ()
                  (integers-from ((lambda (k) (+ k 1)) n))))))
```

A further generalization replaces the concrete step function `(lambda (k) (+ k 1))` by a function parameter:

```
;; Signature: integers-iterate(f,n)  
;; Type: [[Number -> Number] * Number -> Lzl(Number)]  
(define integers-iterate  
  (lambda (f n)  
    (cons-lzl n (lambda () (integers-iterate f (f n))))))
```

Lazy List Iteration

```
(take (integers-iterate (lambda (k) (+ k 1)) 3) 7)  
;; (3 4 5 6 7 8 9)
```

```
(take (integers-iterate (lambda (k) (* k 2)) 3) 7)  
;; '(3 6 12 24 48 96 192)
```

```
(take (integers-iterate (lambda (k) k) 3) 7)  
;; '(3 3 3 3 3 3 3)
```

Observe that this simple generalization of **integers** covers the examples we defined above of the repetition (**ones**), the simple integers sequence, and the powers of two.

Lazy List Iteration

;; Primes – First definition

```
(define primes
  (cons-lzl 2
    (lambda ()
      (lzl-filter prime? (integers-from 3)))))

(define prime?
  (lambda (n)
    (letrec ((iter (lambda (lz)
                      (cond ((> (sqr (head lz)) n) #t)
                            ((divisible? n (head lz)) #f)
                            (else (iter (tail lz)))))))
      (iter primes))))

(take primes 6)
;; '(2 3 5 7 11 13)
```

The second definition we present avoids the redundancy of the computation above. It implements the **sieve** algorithm. The lazy-list of primes can be created as follows:

Lazy List Iteration

- Start with the integers lazy-list:
'(2 3 4 5 ...).
- Select the first prime: 2.
- Filter the current lazy-list from all multiples of 2: '(3 5 7 9 ...)
- Select the next element on the list: 3

Lazy List Iteration

- Filter the current lazy-list from all multiples of 3: '(5 7 11 13 ...).
- i -th step: Select the next element on the list: k . Surely it is a prime, since it is not a multiplication of any smaller integer.
- Filter the current lazy-list from all multiples of k .
- All elements of the resulting lazy-list are primes, and all primes are in the resulting lazy-list.

Lazy List Iteration

```
;; Signature: sieve(lzl)  
;; Type: [Lzl(Number) -> Lzl(Number)]  
(define sieve  
  (lambda (lzl)  
    (cons-lzl (head lzl)  
              (lambda ()  
                (sieve (lzl-filter (lambda (x)  
                                     (not (divisible? x (head lzl))))  
                                   (tail lzl)))))))  
  
(define primes1 (sieve (integers-from 2)))  
(take primes1 7) ;; '(2 3 5 7 11 13 17)
```