A Lazy List Implementation in Squeak

Takashi Yamamiya

This material is based upon work supported in part by the National Science Foundation under Grant No. 0639876. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

VPRI Memo M-2009-006

Viewpoints Research Institute, 1209 Grand Central Avenue, Glendale, CA 91201 t: (818) 332-3001 f: (818) 244-9761

A Lazy List Implementation in Squeak

Takashi Yamamiya

takashi@vpri.org

2009-06-09

Introduction

A lazy list is a collection where each element is only evaluated when necessary. Lazy lists can represent mathematical sequence naturally even it is infinite like all natural numbers, entire prime numbers, or so.

It is also efficient, imagine that you want to replace particular words in a large text, and print out just the first 100 words in the text. If you use a normal list (sometimes it is called as "eager" list), you must either, A) replace all necessary words at first, and then take the first 100 words, or B) count a number of words carefully each time when some word is replaced. A) requires time, and B) tend to lose modularity. With lazy lists, you can write such a program as simple as A), but performance is still reasonable.

Non-strict functional language like Haskell supports this feature in everywhere, but it should be useful to implement a lazy library in "normal" languages. Even though its long history, lazy lists have not been widely used. Lazy lists are sometimes called as "Stream", Unix's pipe can be seen as a kind of lazy list. But it lacks generality. This is interesting topic to find some traps by implementing lazy lists in an object oriented language. Squeak is a powerful tool for this purpose because it has flexible runtime debugging support. I believe most part of difficulty of lazy lists comes from its unintuitive way of runtime behavior. And Squeak's dynamic introspection tools like inspector and explorer should solve the conceptual burden.

Related works

As implementation of lazy lists in imperative languages, SICP [1] is a good tutorial and there are further discussions in SRFI-40 [2] and SRFI-41 [3]. Also Ruby has good implementation [4] similar to SRFI-41. And even Squeak has two implementations (LazyList [5] and LazyCollections [6]). The `LazyList` introduced in this document is categorized as an **even stream** in the terminology of SRFI-40, which means a number of evaluated elements is same as a number of requested elements.

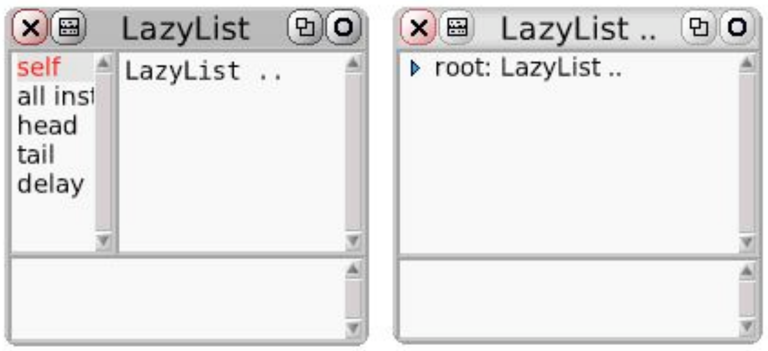
Natural numbers

As a concrete example, we take an implementation of infinite natural numbers. You can download the demo image from the web site [7] or source code from the SqueakSource repository [8].

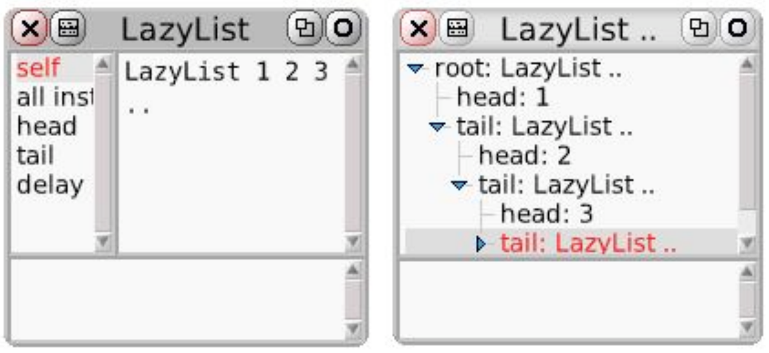
This is an expression to make natural numbers with `LazyList`.

```
list := LazyList nat.
```

We take advantage of Squeak tools, inspector and explore. If you inspect the variable `list`, you will see the internal structure of the list. And if you explore the list, you see public "interface" of it. In other words, inspector shows physical data structure used in the list, but explorer shows only logical appearance. In general, a programmer should not care about its inside, but sometimes those internal insight helps you to understand the behavior. This is the reason why there are two tools to work in different ways.



In this time, neither the inspector (the left window) nor the explorer (the right window) show elements because no element is asked by other objects. Once you open a list by the explorer, elements of public accessors `head` and `tail` are shown, and also content of the inspector is updated. Now, the first element 1 is appeared in both inspector and explorer only because someone sent `head` message to the lazy list. And you can get further numbers with clicking `tail` accessor. Note that while the explorer triggers the list, but the inspector is passive, it never sends `head` or `tail` message so that you can freely observe internal behavior of the list.



Of course, you can directly call these methods in a program. Message `head` answers the first element of list, and `tail` answers following list. This is as same as CAR and CDR pair in lisp language. Various other methods are defined e.g. `take:` returns a `LazyList` including first n elements, and contents convert `LazyList` to `Array`.

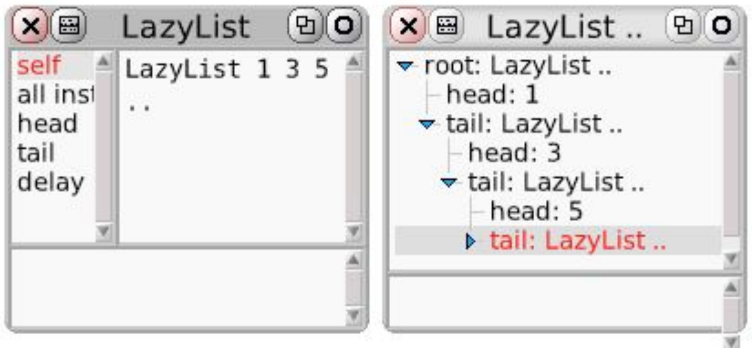
```
list head. "=> returns 1"
list tail. "=> returns LazyList .."

(list take: 10) contents. "=> #(1 2 3 4 5 6 7 8 9 10)"
```

Next expression shows more intelligent behavior. You might know that `select:` is a Smalltalk message where a selected collection filtered by a block is returned (this is called a filter function in other languages). But in a normal situation, using `select:` to an infinite collection is a bad idea because the method never returns. But in case of `LazyList`, unlike other kind of collection, it doesn't evaluate a condition block immediately. Instead, this returns a new lazy list which filters natural numbers.

```
list select: [:e | e odd].
```

This is how a `LazyList` responds to `select:` method in Squeak tools. The condition block `[:e | e odd]` is evaluated only if you click a `tail` element in the explorer.



Making a new LazyList

The easiest way to construct a new lazy list would be using `followedBy:` method. The receiver becomes the head of the new list, and the argument becomes the tail. You can send this to any object, but the argument must be a `LazyList` or block that returns a `LazyList`. Special value `LazyList nil` is used to specify end of the list. This example simply constructs a `LazyList` including 1, 2, and 3.

```
1 followedBy: (2 followedBy: (3 followedBy: LazyList nil))
"=> same as LazyList withAll: #(1 2 3)"
```

You can define natural numbers using `followedBy:` like this.

```
nat := 1 followedBy: [nat collect: [:e | e + 1]].
```

Although, it seems strange. The meaning of this expression is quite straightforward. The first element of the list is obviously 1. How about the second? `collect:` is sometimes called as map function in other languages, and `nat collect: [:e | e + 1]` answers a `LazyList` where all the elements are just larger by 1 than `nat` itself. Therefore the sequence is generated by recursive fashion as follows.

```
nat                                          = 1, ...
nat collect: [:e | e + 1]                    = 2, ...
nat = 1 followedBy: nat collect: [:e | e + 1] = 1, 2, ...
nat collect: [:e | e + 1]                    = 2, 3, ...
nat = 1 followedBy: nat collect: [:e | e + 1] = 1, 2, 3, ...
nat collect: [:e | e + 1]                    = 2, 3, 4, ...
nat = 1 followedBy: nat collect: [:e | e + 1] = 1, 2, 3, 4, ...
```

Matter of fact, This can be written more simpler. Squeak has some great heritage from APL. Many arithmetic functions are defined in collections. So does `LazyList`. There is a simpler definition of natural number.

```
nat := 1 followedBy: [nat + 1].
```

If + is sent to a collection, it adds the argument to all elements in the collection. This is same thing as `collect: [:e | e + 1]` does. This expression reminds me an ugly but familiarly assignment expression with side effect.

```
x := x + 1.
```

This is ugly because it looks like an equation but it is not. `x` is never equal to `x + 1`. Sometimes it confuses a beginner programmer. But same time, it is too useful to discard. We need some reasoning of such side effect to use it safely. On the other hand, `nat := 1 followedBy: [nat + 1]` is similar enough to `x := x + 1`, and indeed, you can see this as an equation without any contradiction. We are going to pursue this mathematical property of lazy lists.

Iteration, Data flow, Lazy list, and Polymorphic function

One of the problems in `x := x + 1` is that value of x depends on the position in the source code. So you have to be careful about order of each expression. Are there any better ways? We assume a mathematical programming language to allow you to define a function as a equation.

```
$x_1 = 1$ --- some initial number
$x_n = x_{n - 1} + 1$
```

In this equations, it is clear that right hand side of $x_{n - 1}$ is older than left hand side of `x`. This is pure and easy to reason. But another problem occurs. In practice, the program requires whole history of `xs` because now a programmer can specify any old value like $x_{n - 100}$.

So we could add a restriction to avoid the problem. What if only **addition** is allowed in a subscript. In other words, what if only **future** value can be referred but past so that we can forget unnecessary past conveniently (this is possible by nice GC).

```
$x_1 = 1$ --- some initial number
$x_{n + 1} = x_n + 1$
```

It makes the relationship between an iteration `x := x + 1` and a lazy list more clear. This iteration can be converted a lazy list definition directly.

```
nat := 1 followedBy: [nat + 1].
       ~~~            ~~~~~~~~~
    $x_1 = 1$;        $x_{n + 1} = x + 1$
```

You can find further discussions from Lucid language. The point of this implementation is that data flow structures (mathematical iteration) can be naturally expressed by combination of lazy lists, polymorphic methods and block expression without non-strict language.

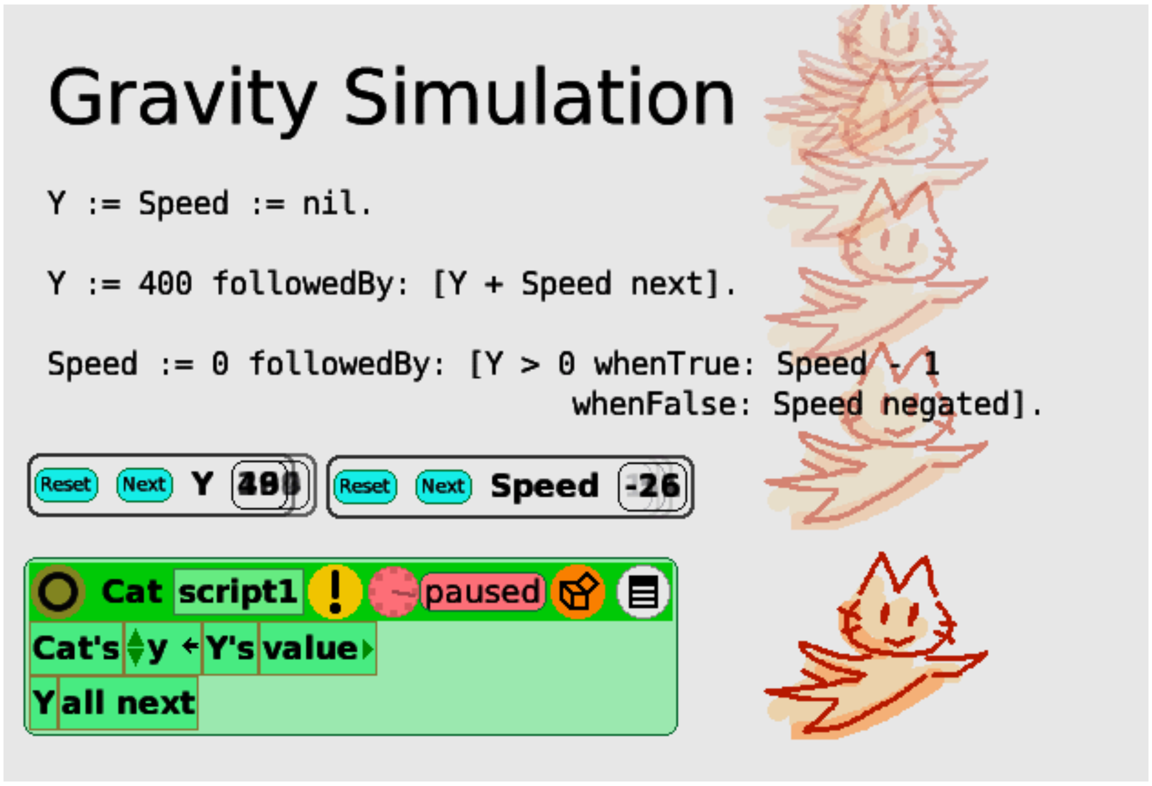
This is another example of definition of a lazy list in iterative way to compute Fibonnaci numbers. `next` is a synonym of `tail`.

```
fib := 1 followedBy: [1 followedBy: [fib + fib next]].
```

This corresponds next mathematical equations.

$$fib_1 = 1$$
$$fib_2 = 2$$
$$fib_{n+2} = fib_n + fib_{n+1}$$

Simulation



Gravity Simulation

```
Y := Speed := nil.

Y := 400 followedBy: [Y + Speed next].

Speed := 0 followedBy: [Y > 0 whenTrue: Speed - 1
                              whenFalse: Speed negated].
```

As a little bit more realistic example, I will show you a gravity simulation using lazy lists. In this demo, the global variable `Y` shows the height of the object, the first position is 400, and the `Y` is increased by `Speed` (it is decreased in case of `Speed` is negative). The initial value of `Speed` is 0 and the value is decreased by 1 while the hight `Y` is positive, it makes gravity acceleration. When `Y` becomes negative, `Speed` is suddenly negated, which means the object is bounced by the ground.

```
Y := 400 followedBy: [Y + Speed next].

Speed := 0 followedBy: [Y > 0 whenTrue: Speed - 1
                              whenFalse: Speed negated].
```

Although, the program is simple enough. It makes a quite fun animation combined with Morphic graphics system. Morphic part of this demo uses a special morph named `ListPointer` (in the screen shot, rounded rectangles with Reset and Next buttons) that worked as an interface between `LazyList` and Morphs. When Reset button is pressed, a `ListPointer` points the first element of a globally defined `LazyList` specified by its name. And when Next button is pressed, it points the next value with `tail` message. There is a special tile named `all next` which works as same as all next buttons in the world is pressed. Current value what a `ListPointer` points can be accessed by `value` tile in Etoys.

In this case, the `[ Cat's y <- Y's value ]` tile assigns current value of `Y` to the cat's y coordinate, and the `[Y all next]` tile steps forward all `LazyList` values captured by `ListPointer`.

Implementation

This is the class definition of `LazyList`. The definition of super class `List` is omitted, but `List` is a subclass of `Collection` and provides common behavior to list classes. `List` has no members. Other subclasses are `EagerList` and `Iterator`. `EagerList` is normal LISP style cons cell, and `Iterator` is more efficient implementation of homogeneous lazy list. The class variable `Nil` is used as a sentinel object. Note that this `Nil` is **not** a singleton object. See the definition of `isEmpty` below.

```
List subclass: #LazyList
        instanceVariableNames: 'head tail delay'
        classVariableNames: 'Nil'
        poolDictionaries: ''
        category: 'Collections-Lazy'
```

The most basic constructor of `LazyList` is `LazyList class >> delay:`. When new `LazyList` is constructed, either `head` or `tail` are nil. But a block value is stored to `delay`. The block is used as a "thunk" in the list and it is evaluated once `head` or `tail` is called. Sometimes delay is also called promise.

```
LazyList class >> delay: aBlock
        "aBlock is a block which answers {anObject. aLazyList}"
        ^ self new privateDelay: aBlock

LazyList >> privateDelay: aBlock
        delay := aBlock
```

In most of cases, a helper method `Object >> followedBy:` or `LazyList class >> head:tail:` is much easier than `delay:`. You can define a `LazyList` by normal cons style notation with these methods.

```
Object >> followedBy: aLazyListOrBlock
        "Lucid's followed by (fby) operator"
        ^ LazyList head: self tail: aLazyListOrBlock

LazyList class >> head: anObject tail: listOrBlock
        ^ self
                delay: [{anObject. listOrBlock value}]
```

For example, `List class >> nat` introduced to make a list of infinite natural numbers above is defined in terms of `head:tail:`

```
List class >> nat
        | nat |
        nat := self head: 1 tail: [nat + 1].
        ^ nat
```

When an accessor `head` or `tail` is called, both pair values are evaluated from the delay block. Once values are determined, the list is marked as **forced**, values are stored to each instance variables, and the block never executed again (it is ensured by assigning `nil` to the `delay` slot).

```
LazyList >> head
        self force.
        ^ head

LazyList >> tail
        self force.
        ^ tail

LazyList >> force
        | pair |
        delay
                ifNil: [^ nil].
        pair := delay value.
        pair
                ifNil: [head := tail := nil]
                ifNotNil: [head := pair first.
                        tail := pair second].
        delay := nil
```

Although an empty list is used to mark the end of list, a pitfall is that an empty list is not a singleton object. Because it is not possible to tell whether a `LazyList` is empty or not before it is forced. For example, any lazy list created by `select:` can become a nil list whenever the source list is a nil or rest of all elements don't satisfy the condition block. As a convention, an nil list is defined as a `LazyList` where its tail value is `nil`.

```
LazyList >> isEmpty
        ^ self tail = nil
```

This is a kind of control structure made from a `LazyList` used in the gravity simulation. This returns a mixed list from `list2` and `list3` which depends on boolean values of the receiver.

```
whenTrue: list2 whenFalse: list3
        "Listwize condition. Note: list2 and list3 are lists, not blocks."
        ^ self class
                delay: [(self isEmpty
                                        or: [list2 isEmpty
                                                        or: [list3 isEmpty]])
                                ifFalse: [
                                        {self head
                                                ifTrue: [list2 head]
                                                ifFalse: [list3 head].
                                        self tail
                                                whenTrue: list2 tail
                                                whenFalse: list3 tail}]]
```

Iterator

`Iterator` is another implementation of lazy lists. One of useful applications of lazy lists is a stream. For example, an output text stream can be represented as a lazy list where each element is a character. A stream is a data structure commonly used when you need to concatenate many strings. Typical implementation of streams are designed to delay memory allocation regardless how often concatenation happens.

`Iterator` is more suitable for the purpose than `LazyList`, because instead of each element, a chunk data is evaluated when it is forced. Common use case of `Iterator` is shown below. You can use comma operator to concatenate as normal `String` object. But actual concatenation happens only when `contents` is sent.

```
stream := (Iterator withAll: 'Hello'), (Iterator withAll: 'World'). "=> an Iterator ..."
stream contents. "=> HelloWorld"
```

Also, an argument of `,` operator can be a block. With this feature, you can express infinite length of string. The `printOn:` method in the library is designed with this function.

```
helloForever := (Iterator withAll: 'Hello '), [helloForever].
(helloForever take: 20) contents "=> Hello Hello Hello He"
```

This is the definition of `Iterator`. `collection` is a chunk of this list, position is the first position of the list, `successor` is next list, and `delay` is a delay block for the thunk.

```
List subclass: #Iterator
        instanceVariableNames: 'collection position successor delay'
        classVariableNames: 'Nil'
        poolDictionaries: ''
        category: 'Collections-Lazy'
```

`position` is used to keep a part of chunk. For example, when an iterator with "Hello World" is created and forced, the position is 1. If you access the third element of the stream with `tail tail`, a new stream beginning with the third element is returned. The position of return value is 3. While standard `Stream` classes in Smalltalk updates its state when `next` is called. An iterator's `tail` method returns a newly created object.

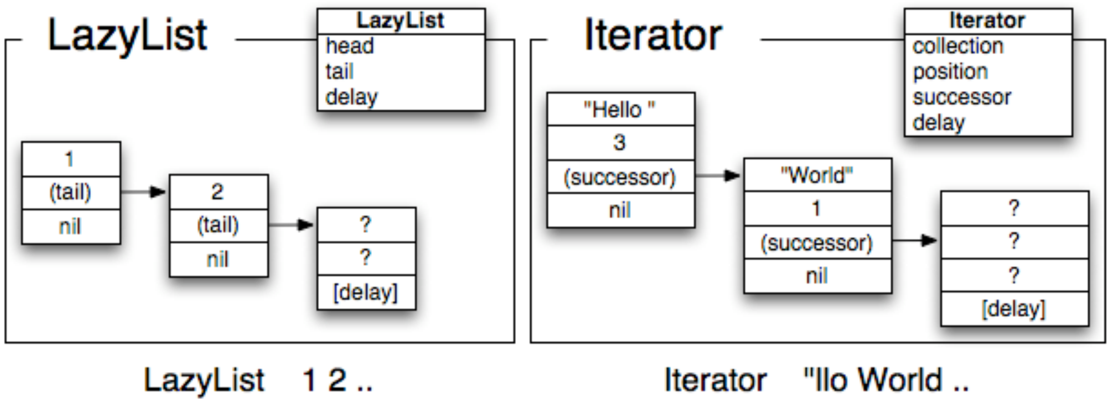
Although `tail` return a new object, the chunk data in `collection` is shared with original list. This a good property in terms of performance of space and time. This is safe because LazyList and `Iterator` are functional and they never change the states except force operation.

```
stream := Iterator withAll: 'Hello World'.
stream position. "=> 1"
stream contents. "=> Hello World"

stream tail tail position. "=> 3"
stream tail tail contents. "=> llo World"
```

The largest advantage of lazy lists compare to standard `Stream` is that lazy lists has same interface as `Collection` classes. And `Collection` and `Stream` can be treated same in abstract way.

```
((Iterator withAll: 'hello') collect: [:e | e asUppercase ]) contents "=> HELLO"
```



Two lazy list implementations

Method summary

Constructors

```
LazyList class >> delay: aBlock
```

The primary constructor of `LazyList`. aBlock must return an array with two elements when `value` is sent. The first element will be the `head` and second will be the `tail` if it is forced. `tail` must be a new `LazyList`

```
Iterator class >> delay: aBlock
```

The primary constructor of `Iterator`. aBlock must return an array with three elements when `value` is sent. The first element will be the `collection`, second will be the `position`, and third will be the `successor`. `successor` must be a new `Iterator`.

```
LazyList class >> head: anObject tail: listOrBlock
Iterator class >> head: anObject tail: listOrBlock
Object >> followedBy: aLazyListOrBlock
```

Utility constructors. `head:tail:` constructs a new lazy list where the head is `anObject` and the tail is `listOrBlock`. Message `value` is always sent to the `listOrBlock` when list is is forced. `Object >> followedBy:` is same as `LazyList class >> head:tail:` but the receiver becomes the head and the argument becomes the tail.

```
LazyList class >> constant: anObject
```

Return a infinite lazy list with each elements is all `anObject`.

Accessors

Each accessor forces the receiver when necessary. If a lazy list is forced, the delay block is executed and elements are visible to outside.

```
List >> head
List >> tail
```

These are the primary accessors. `head` returns the first element of the receiver. `tail` returns rest of the receiver.

`List >> isEmpty`

Return `true` if the list is empty, otherwise `false`.

`List >> length`

Return the number of elements in the receiver if it is finite length.

`List >> contents`

`LazyList >> contents` returns an `Array` with all elements in the receiver. `Iterator >> contents` returns same `Collection` as receiver's collection with all elements in the receiver.

List functions

`List >> , aList`

Return a new lazy list which the receiver and `aList` will be concatenated.

`List >> collect: aBlock`

Return a new lazy list where each element will be valued by `aBlock`.

`List >> select: aBlock`

`aBlock` must return a boolean value when `value: anObject` is sent. `select:` returns a new lazy list including only elements where `aBlock` returns true from it.

`List >> take: aNumber`

Returns new a lazy list with first `aNumber` elements in the receiver.

Conclusion

Lazy lists are widely used in functional programming languages because it can express infinite list naturally and avoid unnecessary evaluation. Although these advantages come from the fact that separation between execution order and the data structure, it is hard to follow the runtime behavior to people who are not familiar with lazy evaluation. This implementation combines the mathematical cleanness of lazy lists with easy debugging facility by Squeak tools, concise data flow oriented expression by polymorphic method and block syntax. It also shows that conventional `Stream` object can be represented as a lazy list.

References

- [1] Structure and Interpretation of Computer Programs: http://mitpress.mit.edu/sicp/full-text/sicp/book/node69.html
- [2] SRFI-40: http://srfi.schemers.org/srfi-40/srfi-40.html
- [3] SRFI-41: http://srfi.schemers.org/srfi-41/srfi-41.html
- [4] Lazylist implementation for Ruby: http://lazylist.rubyforge.org/
- [5] LazyList (SqueakSource): http://www.squeaksource.com/LazyList.html
- [6] LazyCollections (SqueakSource): http://www.squeaksource.com/LazyCollections.html
- [7] LazyList demo image: http://languagegame.org/pub/LazyList.zip
- [8] LazyList Leisure (SqueakSource): http://www.squeaksource.com/Leisure.html