1 Lazy List Implementation

1.1 A Simple Implementation

⟨*lazylist.R*⟩≡
```
Cons <- function(head, tail)
    structure(list(head = head, tail = function() tail),
              class = "LazyList")

Head <- function(x) x$head
Tail <- function(x) x$tail()
```

Is it worth assigning the tail to clear the promise? Use something like

⟨*tail function with assignment*⟩≡
```
tail = function() { tail <<- tail; tail }
```

⟨*lazylist.R*⟩+≡
```
print.LazyList <- function(x, ...) {
    cat(paste("<", Head(x), ", ... >\n"))
}
```

1.2 Alternate Implementations

1.2.1 Fully Lazy Version

Version with lazy head. Seems to leak too much memory with nested promises.

⟨*version with lazy head*⟩≡
```
Cons <- function(head, tail)
    structure(list(head = function() head, tail = function() tail),
              class = "LazyList")

Head <- function(x) x$head()

print.LazyList <- function(x, ...) {
    cat("<lazy list>\n")
}
```

1.2.2 Version Using Delay

Version using delay–not noticeably better for speed and less clear I think.

⟨*version using delay*⟩≡
```
Cons <- function(head, tail)
    structure(list(head = head, tail = delay(tail, environment())),
              class = "LazyList")

Tail <- function(x) { d<-x$tail; d }
```

1.2.3 Using Assignment To Remove Promise

Use assignment to eliminate the evaluated promise to avoid saving environments and nested promises. Could be done more efficiently with internal code.

⟨*version using assignment to remove promises*⟩≡

```
Cons <- function(head, tail) {
    head <- head
    structure(list(head = head,
                   tail = function() { tail <<- tail; tail }),
              class = "LazyList")
}
```

1.2.4 Implementation Without Promises

Jazz up to handle errors in force nicely?

⟨*version without promises*⟩≡

```
Cons <- function(head, tail) {
    head <- head
    expr <- substitute(tail)
    prenv <- parent.frame()
    env <- new.env(parent = NULL)
    assign("expr", expr, env = env)
    assign("prenv", prenv, env = env)
    structure(list(head = head, tail = env),
              class = "LazyList")
}

Tail <- function(x) {
    if (! exists("value", env = x$tail)) {
        expr <- get("expr", env = x$tail)
        prenv <- get("prenv", env = x$tail)
        assign("prenv", NULL, env = x$tail)
        assign("value", eval(expr, prenv), env = x$tail)
    }
    get("value", env = x$tail)
}
```

1.3 Issues

Could use internal version that uses promises and removes them when evaluating.

If an error occurs when forcing a promise then the evaluation pending marker remains set. In internal version could avoid this. (Or we could just use a try and some internal code for promise manipulation.)

2 Lazy List Utility Functions

Mostly based on Paulsen's ML book; some from Abelson and Sussman.

2.1 Constructing Lists

⟨*lazylist.R*⟩+≡

```
intList <- function(i) Cons(i, intList(i+1))

Iterates <- function(x, f) {
    iter <- function(x, f) {
        y <- f(x)
        Cons(y, iter(y, f))
    }
    Cons(x, iter(x, f))
}

#**** as.List
FromList <- function(x) {
    v <- NULL;
    for (i in rev(as.list(x)))
        v <- do.call("Cons", list(i, v))
    v
}

#**** fix Take for finite list

Append <- function(x, y) {
#**** coerce t list??
    if (is.null(x))
        y
    else
#**** force eval of y to avoid buildup of nested promises?
        Cons(Head(x), Append(Tail(x), y))
}

Repeat <- function(x)
    Append(x, Repeat(x))
```

2.2 Selecting Parts of a List

⟨*lazylist.R*⟩+≡

```
Elt <- function(x, n) {
    if (n <= 0)
        NULL
    else {
        while (n > 1 && ! is.null(x)) {
            n <- n - 1
            x <- Tail(x)
        }
        if (is.null(x))
            NULL
        else
            Head(x)
    }
}

Take <- function(n, x) {
    if (n <= 0)
        NULL
    else
        sapply(1:n, function(i) { v <- Head(x); x <<- Tail(x); v })
}

Drop <- function(n, x) {
    if (n > 0)
        for (i in 1:n)
            x <- Tail(x)
    x
}
```

2.3 Mapping and Filtering

⟨*lazylist.R*⟩+≡

```
TakeWhile <- function(test, x) {
    y <- x
    n <- 0
    while (! is.null(x) && test(Head(x))) {
        n <- n + 1
        x <- Tail(x)
    }
    Take(n, y)
}

DropWhile <- function(test, x) {
    while (! is.null(x) && test(Head(x)))
        x <- Tail(x)
    x
}
Filter <- function(x, test) {
    if (test(Head(x)))
        Cons(Head(x), Filter(Tail(x), test))
    else
        Filter(Tail(x), test)
}

Filter <- function(x, test) {
    while (! test(Head(x)))
        x <- Tail(x)
    Cons(Head(x), Filter(Tail(x), test))
}
Filter <- function(x, test) {
    while (! is.null(x) && ! test(Head(x)))
        x <- Tail(x)
    if (is.null(x))
        NULL
    else
        Cons(Head(x), Filter(Tail(x), test))
}

Map <- function(x, fun) {
    if (is.null(x))
        NULL
    else
        Cons(fun(Head(x)), Map(Tail(x), fun))
}
```

2.4 Interleaving Lists

⟨*lazylist.R*⟩+≡
```
Interleave <- function(x, y) {
    if (is.null(x))
        y
    else if (is.null(y))
        x
    else
        Cons(Head(x), Interleave(y, Tail(x)))
}

Zip <- function(x, y) {
    if (is.null(x) || is.null(y))
        NULL
    else
        Cons(Head(x), Cons(Head(y), Zip(Tail(x), Tail(y))))
}
```

3 Examples

3.1 Simple Examples

⟨*R session*⟩≡
```
> intList(3)
<lazy list>
> i3<-intList(3)
> Head(i3)
[1] 3
> Head(Tail(i3))
[1] 4
> Head(Tail(Tail(i3)))
[1] 5
```

3.2 Classical Examples

3.2.1 Fibonacci Sequence

⟨*fibonacci sequence*⟩≡
```
fib <- function(x,y)
   Cons(x, Cons(y, Tail(fib(y, x+y))))

fib1 <- function(x,y) Cons(x, fib1(y, x+y))
```

Abelson and Sussman version:

⟨*fibonacci sequence*⟩+≡
```
Add <- function(x,y)
    Cons(Head(x) + Head(y), Add(Tail(x), Tail(y)))
fibs <- Cons(0, Cons( 1, Add(Tail(fibs), fibs)))
```

Defining fib "naturally" would need `+.LaxyList` method

⟨*fibonacci sequence*⟩+≡
```
fibs <- Cons(0, Cons( 1, Tail(fibs) + fibs))
```

3.2.2 Finding Primes

Abelson and Sussman version:

⟨*primes*⟩≡
```
Sieve <- function(x) {
    p <- Head(x)
    Cons(p, Sieve(Filter(Tail(x), function(y) y %% p != 0)))
}
p<-Sieve(intList(2))

Take(20, p)
```

Blows up with too deep recursion much a

⟨*R session*⟩+≡
```
> options(expressions=2000)
> p<-Sieve(intList(2))
> Take(97,p)  # 98 fails
```

Paulson version:

⟨*primes*⟩+≡
```
Sift <- function(x, p)
    Filter(x, function(y) y %% p != 0)

Sieve <- function(x) {
    p <- Head(x)
    Cons(p, Sieve(Sift(Tail(x), p)))
}

Sieve <- function(x)
    Cons(Head(x), Sieve(Sift(Tail(x), Head(x))))
```

Alternate version from Abelson and Sussman:

⟨*primes*⟩+≡
```
#***Higher order function to simplify this?
isPrime <- function(x) {
    for (p in TakeWhile(function(p) p^2 <= x, primes))
        if (x %% p == 0)
            return(FALSE)
    TRUE
}

primes <- Cons(2, Filter(intList(3), isPrime))
isPrime <- function(x) {
    p <- primes
    while (Head(p)^2 <= x) {
        if (x %% Head(p) == 0)
            return(FALSE)
        else
            p <- Tail(p)
    }
    TRUE
}
```

3.3 More Statistical Examples

Richardson extrapolation; Aitken extrapolation; Newton's method for Gamma, say, with different convergence rules; Monte Carlo sequence (drop first n, keep every m-th);

3.4 Miscellaneous Stuff

This seems OK

⟨*tests*⟩≡
```
i <- intList(1)
gc() # clear out .Last.value
while (TRUE) i <- Tail(i)
```

def

⟨*tests*⟩+≡
```
f <- function() {
  i <- intList(1)
  while (TRUE) i <- Tail(i)
}
```

Look at GC triggering again some time

`is_missing_arg` in methods probably shouldn't exist – make `isMissing` in `envir.c` public?

Think about separating missing/substitute stuff so it doesn't need to look at promises. Missing/substitute stuff can be determined from the matched call. Matched calls can be computed at compile time in some settings; in others they can be computed lazily if they are not needed anyway.

Look at CL Series and Gatherers package. Waters' papers

Look ad Gatherers in Python (with and without Stackless). http://www.amk.ca/python/2.2/index.html http://python.sourceforge.net/peps/pep-0255.html

Think about Iterators (eg CLU).

Does Python list comprehension stuff have anything to do with this?