Lazy Evaluation

We're used to **eager** evaluation of expressions:

```
f(x + y, g z)

(x+y) :: (g z) :: nil
```

Arguments to function and constructor applications are evaluated to **values** (base types, datatypes, functions, etc.) before the function is entered or the construction is made.

Evaluating arguments can be expensive (perhaps infinitely so), and sometimes this effort is wasted:

```
fun g c = ...very expensive...
fun f(a, b) =  if a < 10 then b+7 else 0
```

Under **lazy** evaluation, expressions are not evaluated until they are **needed**, i.e., used as arguments to a primitive function or subjected to a **case** pattern match. Sometimes they are not evaluated at all. On the other hand, expressions should not be evaluated more than once either:

```
fun f (b) = b + b
f (g c)
```

Most **pure** functional languages, such as Miranda and Haskell, use lazy evaluation.

Example: Comparing Tree Fringes

Consider the problem of comparing the **fringes** (leaves in left-to-right order) of two binary trees, not necessarily having the same shape.

```
datatype ''a tree = Node of ''a tree * ''a tree
                  | Leaf of ''a
val a = Node(Leaf 1,Node(Leaf 2, Leaf 3))
val b = Node(Node(Leaf 1, Leaf 2), Leaf 3)
val c = Node(Leaf 2,Node(Node(Leaf 3, Leaf 4),Leaf 5))
val d = Node(Node(Leaf 2,Leaf 3),Node(Leaf 4, Leaf 5))
fun samefringe(t1:''a tree, t2:''a tree) : bool = ???
```

Here `a` and `b` have the same fringes, as do `c` and `d`.

`samefringe` is quite tricky to write if we operate directly on the trees (try it!), but here's a simple solution (in eager ML)

```
fun flatten (Node(l,r)) = (flatten l) @ (flatten r)
  | flatten (Leaf x) = [x]
fun samefringe(t1,t2) = eqlist(flatten t1,flatten t2)
```

The only problem is that `samefringe` always flattens both trees into lists before comparing them, even if the very leftmost leaves are unequal. This could be quite expensive if the trees are large. But if we run the same program assuming **lazy** evaluation, only the heads of the lists need to be computed.

Conditionals are Lazy

Even eager languages have one form of lazy operator: the `if-then-else` or `case` construct:

```
if (a < 10) then g(z) else 0
```

It's essential that only one arm of the conditional be executed.

In a lazy language, we can actually write conditionals as functions:

```
fun if0(c:int,t:'a,f:'a) =  if c = 0 then t else f
if0(a = b, g(c), 2/0)
```

Order of Evaluation

Note that the **order of evaluation** of expressions in lazy programs is very hard to predict (and can depend on runtime values).

For this reason, side-effects don't fit in well with lazy languages, since the order of their execution is important. Hence only purely functional languages use laziness.

```
fun g(x) = (print x; x+x)
val a = g(if0(y,g(2),g(3)))
```

IO is particularly problematic.

Lazy Fringes

```
fun flatten (Node(l,r)) = (flatten l) @ (flatten r)
  | flatten (Leaf x) = [x]
fun op@(h::t,l) = h :: (t@l)
  | op@(nil,l) = l
fun eqlist (h1::t1,h2::t2) =
              h1 = h2 andalso (eqlist(t1,t2))
  | eqlist (nil,nil) = true
  | eqlist (_,_) = false
fun samefringe(t1,t2) = eqlist(flatten t1,flatten t2)

samefringe(Node(Leaf 1,a...),
           Node(Node(Leaf 2,b...),c...))
⇝ eqlist(flatten(Node(Leaf 1,a...)),
         flatten(Node(Node(Leaf 2,b...),c...)))
⇝ eqlist(flatten(Leaf 1) @ (flatten a...),
         flatten(Node(Leaf 2,b...)) @ (flatten c...))
⇝ eqlist(flatten(Leaf 1) @ (flatten a...),
         (flatten(Leaf 2) @ (flatten b...))
          @ (flatten c...))
⇝ eqlist([1] @ (flatten a...),
         ([2] @ (flatten b...)) @ (flatten c...))
⇝ eqlist(1::(nil @ (flatten a...)),
         (2::(nil @ flatten b...)) @ (flatten c...))
⇝ eqlist(1::(nil @ (flatten a...)),
         2::((nil @ flatten b...) @ (flatten c...)))
⇝ 1 = 2 andalso
          (eqlist(nil @ flatten a...,
                  (nil @ flatten b...) @ (flatten c...)))
⇝ false
```

Infinite Data Structures

Consider the following definition:

```
val rec inftree = Node(Leaf 1,inftree)
```

This doesn't represent a finite object, so may appear not to make sense. But suppose it happens (under lazy evaluation) that we **never** need to fully evaluate `inftree`, but only a finite part of it.

```
fun sumtop 0 (Node(l,r)) = 0
  | sumtop n (Node(l,r)) =
            sumtop (n-1) l + sumtop (n-1) r
  | sumtop _ (Leaf v) = v
val a = sumtop 5 inftree
```

Then the definition of `inftree` makes perfect sense.

Using **infinite data structures** with lazy evaluation is particularly useful for building modular solutions to generate-and-test problems. We write one function to generate a (potentially) infinite tree of candidate solutions, and another to test the validity of a solution; just composing the functions gives an efficient program that automatically prunes useless branches of the search space.

Encoding lazy evaluation in eager languages

We can get the effect of laziness in eager languages by making use of explicit **suspensions**. The idea is to represent a lazy expresion of type `'a` by an eager expression of type `unit -> 'a`. When a value is needed, we must obtain it by **forcing** the expression, i.e., applying it to `():unit`.

```
fun if0(c:int, t: unit -> 'a, f: unit -> 'a) : 'a =
  if c = 0 then t() else f()

if0(a = b, fn () => g(c), fn () => 2/0)
```

We can be a little neater by wrapping things into a datatype:

```
datatype 'a susp = Susp of unit -> 'a
fun force (Susp e) = e()
fun if0(c:int, t: 'a susp, f: 'a susp) : 'a =
  if c = 0 then force t else force f
if0(a = b, Susp(fn () => g(c)), Susp(fn () => 2/0))
```

Actually, to avoid repeated evaluation of suspensions we need a fancier encoding:

Sieve of Eratosthenes

A (potentially) infinite list can be viewed as a **stream**: an unbounded sequence of values provided on demand.

```
val rec ones = 1::ones
fun from n = n::(from (n+1))
```

Using streams often produces neat solutions to problems.

```
fun multipleof a b = (b mod a = 0)
fun filter p (h::t) =
   if p h then h::(filter p t) else filter p t
fun remove a l = filter(not o (multipleof a)) l
fun sieve(h::t) = h :: sieve(remove h t)

fun from n = n::(from (n+1))
val primes = sieve(from 2)

fun prefix 0 l = []
  | prefix _ [] = []
  | prefix n (h::t) = h::(prefix (n-1) t)
val first_10_primes = prefix 10 primes
```

```
datatype 'a inside = Eval of 'a | Uneval of unit -> 'a
and      'a susp = Susp of 'a inside ref
fun delay (x:unit -> 'a) = Susp(ref(Uneval x))
fun force (Susp e) =
   case !e of
     Eval x => x
   | Uneval f => let val v = f()
                 in x := Eval v;
                    v
                 end
fun f(b:int susp) = (force b) + (force b)
f(delay (fn () => g c))
```