list comprehensions

```
map::  (a -> b) -> [a] -> [b]
map f xs = [f x | x <- xs]

filter::  (a -> Bool) -> [a] -> [a]
filter p xs = [x | x <- xs, p x]

cross::  [a] -> [b] -> [(a,b)]
cross xs ys = [(x,y) | x <- xs, y <- ys]

quicksort ::  Ord a => [a] -> [a]

quicksort  []    =  []
quicksort (x:xs) =  quicksort [y | y <- xs, y < x ]
                    ++ [x]
                    ++ quicksort [y | y <- xs, y >= x]
```

list comprehensions

“Python’s list comprehension syntax is taken (with trivial keyword/symbol modifications) directly from Haskell. The idea was just too good to pass up.” `wiki.python.org`

```
>>> [x + 42 for x in range(5)]
[42, 43, 44, 45, 46]

>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

playing with Haskell

Ranges:

```
prompt> [1..10]
[1,2,3,4,5,6,7,8,9,10]
prompt> [1,3..10]
[1,3,5,7,9]
prompt> [10,9..1]
[10,9,8,7,6,5,4,3,2,1]
prompt> [10,8..1]
[10,8,6,4,2]
```

lazy evaluation

Don’t evaluate before you have to:

```
and' :: Bool -> Bool -> Bool
and' False _ = False
and' _ x = x
```

```
prompt> head []

*** Exception: prompt.head: empty list
prompt> and' False (head [] == 2)

False
```

How did this work?

lazy evaluation

- Expressions are not evaluated when they are bound to variables, but their evaluation is **deferred** until their results are needed by other computations.
- Arguments are not evaluated before they are passed to a function, but only when their values are actually used.
- A **thunk** is an unevaluated value with a recipe that explains how to evaluate it.
- It is possible to **partially evaluate** an expression, for example `( thunk, thunk )`.
- Aside: lookup Racket's `thunk` function.

lazy evaluation

```
length [] = 0
length (_:xs) = 1 + length xs
```

What do we need to evaluate in the expression `length [42^1234, 42^2345, 42^3456]`?

```
length thunk{[42^1234, 42^2345, 42^3456]}  pattern match
length (_ : thunk{[42^2345, 42^3456]})     function body
1 + length thunk{[42^2345, 42^3456]}       pattern match
1 + length (_ : thunk{[42^3456]})          function body
1 + (1 + length thunk{[42^3456]})          pattern match
1 + (1 + length (_ : thunk{[]}))           function body
1 + (1 + (1 + length thunk{[]}))           pattern match
1 + (1 + (1 + length []))                  function body
1 + (1 + (1 + 0))                          display
3
```

lazy evaluation

```
ones = 1 : ones
```

What kind of a thing is `ones`?
What happens if we evaluate `ones` in the REPL?
But we can use it in other ways. For example:

```
prompt> tenOnes = take 10 ones
[1,1,1,1,1,1,1,1,1,1]
```

How does this work? We need to know how `take` is defined:

```
take _ [] = []
take 0 _ = []
take n (x:xs) = x : take (n-1) xs
```

lazy evaluation

```
take _ [] = []
take 0 _ = []
take n (x:xs) = x : take (n-1) xs
```

Then:

```
take 3 thunk{ones}                      pattern match
take 3 (1 : thunk{ones})                function body
1: take thunk{3-1} thunk{ones}          pattern match
1: take 2 (1 : thunk{ones})             function body
1:1 : take thunk{2-1} thunk{ones}       pattern match
1:1 : take 1 (1 : thunk{ones})          function body
1:1:1 : take thunk{1-1} thunk{ones}     pattern match
1:1:1 : take 0 _                        function body
1:1:1: []
```

lazy evaluation

More fun examples:

```
numsFrom n = n : numsFrom (n + 1)

nats = numsFrom 0
nats = 0 : map (+1) nats

squares = map (^2) nats

odds = filter odd nats
evens = filter even nats

fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
fibs = 0:1:[x+y | (x, y) <- zip fibs (tail fibs)]

prime x = null [y | y <- [2..(x-1)], x `mod` y == 0]
primes = [x | x <- numsFrom 2, prime x]
primes = filter prime $ numsFrom 2
```