

## Comp 205: Comparative Programming Languages

### Lazy Evaluation and Infinite Lists

Lecture notes, exercises, etc., can be found at:  
[www.csc.liv.ac.uk/~grant/Teaching/COMP205/](http://www.csc.liv.ac.uk/~grant/Teaching/COMP205/)

## Never-Ending Recursion

The expression `[n..]` can be implemented generally by a function:

```
natsfrom :: num -> [num]
natsfrom n = n : natsfrom (n+1)
```

This function can be invoked in the usual way:

```
Main> natsfrom 0
[0,1,2,3,....
```

## Example: Indexing Lists

Suppose we want a function that will index the elements in a list, i.e., number the elements:

```
indexList :: [a] -> [(Int,a)]
```

such that, for example,

```
indexList ["Those", "are", "pearls"]
=
[(1, "Those"), (2, "are"), (3, "pearls")]
```

## One Way...

One way of implementing this is with a recursive definition:

```
indexList xs
= ind 1 xs
  where
    ind n [] = []
    ind n (x:xs) = (n,x) : ind (n+1) xs
```

## ... Or ...

Another way of implementing this is to use the built-in function `zip`:

```
zip :: [a] -> [b] -> [(a,b)]

zip xs [] = []
zip [] ys = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

```
Main> zip [1,2,3] ["Those", "are"]
[(1,"Those"),(2,"are")]
```

## ... Another

Using an infinite list ensures there are enough indices:

```
indexList xs = zip [1..] xs
```

Here, `zip` is a selector.

## Recursive Definitions

The Fibonacci sequence can be recursively defined as follows:

```
fibs m n = m : fibs n (m+n)
```

(cf. previous definitions using pairs).

```
Main> fibs 1 1  
[1,1,2,3,5,8,13,21,34,55,89,144,...]
```

## Defining Constants

Constants are functions of no arguments (they vary over nothing...)

```
days = ["Mon", "Tue", "Wed", "Thu", "Fri"]  
  
-- NB: days is a constant,  
-- not a "variable":  
-- we can't now write  
--  
-- days = ["M", "T", "W", "Th", "F"]
```

## Using Constants

Constants can be used in expressions, but they cannot be redefined:

```
Main> days ++ ["Sat"]  
["Mon", "Tue", "Wed", "Thu", "Fri", "Sat"]
```

We cannot write (in the script file):

```
days = days ++ ["Sat", "Sun"]
```

## Recursive Constants

Constants can be recursively defined:

```
allOnes = 1 : allOnes
```

```
Main> allOnes  
[1,1,1,1,1,1,1,1,1,1,1,1,...]  
Main> zip allOnes [1,2]  
[(1,1),(1,2)]
```

## Recursive Constants

Constants can be recursively defined with local definitions:

```
fibs = 1 : 1 : fplus fibs (tl fibs)  
  where  
    fplus (l:ls) (m:ms)  
      = 1 + m : fplus ls ms
```

## Eratosthenes' Sieve

A number is prime iff

- it is divisible only by 1 and itself
- it is at least 2

The sieve:

- start with all the numbers from 2 on
  - delete all multiples of the first number from the remainder of the list
- repeat

## Eratosthenes' Sieve I

To generate a list of all the prime numbers:

```
primes = eratosthenate [2..]
```

## Eratosthenes' Sieve II

To remove multiples of a number  $n$  from a list  $l$ :

```
sieve n = filter (/=0).(`mod` n)
```

## Eratosthenes' Sieve III

To generate one prime:

```
next n ps = n : sieve n ps
```

To generate all primes:

```
eratosthenate = foldr next []
```

## Eratosthenes' Sieve IV

To generate all primes:

```
primes = sieveAll [2..]  
  where  
    sieveAll (p:ns)  
      = p : sieveAll (sieve p ns)
```

## Summary

- Infinite lists and lazy evaluation give a powerful combination for recursive definitions
- (sometimes hard to follow)

*Next: Introducing OBJ*