Lazy Evaluation and Infinite Lists

The expression \([\ldots]\) can be implemented generally by a function:

\[
\text{natsfrom} :: \text{num} \rightarrow \text{[num]}
\]

\[
\text{natsfrom} \ n = \text{natsfrom} \ (n+1)
\]

This function can be invoked in the usual way:

\[
\text{Main>} \ \text{natsfrom} \ 0
\]

\[
[0,1,2,3,\ldots]
\]

Example: Indexing Lists

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

\[
\text{indexList} :: \text{[a]} \rightarrow \text{[(Int,a)]}
\]

such that, for example,

\[
\text{indexList} \ ["Those", "are", "pearls"]
\]

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

One Way...

One way of implementing this is with a recursive definition:

\[
\text{indexList} \ \text{x}s
\]

\[
\text{where}
\]

\[
\text{ind} \ [\_] = []
\]

\[
\text{ind} \ (n:xs) = (n, x) : \text{ind} \ (n+1) \ x s
\]

\[
\text{Main>} \ \text{indexList} \ [1,2,3,\ldots] \ ["Those", "are"]
\]

\[
\text{[(1,"Those"),(2,"are")]}\]

... Or ...

Another way of implementing this is to use the built-in function \text{zip}:

\[
\text{zip} :: \text{[a]} \rightarrow \text{[b]} \rightarrow \text{[(a,b)]}
\]

\[
\text{zip} \ \text{xs} \ [] = []
\]

\[
\text{zip} \ [] \ \text{ys} = []
\]

\[
\text{zip} \ (\text{x}:\text{xs}) \ (\text{y}:\text{ys}) = \text{((x,y)} : \text{zip} \ \text{xs} \ \text{ys}
\]

\[
\text{Main>} \ \text{zip} \ [1,2,3] \ ["Those", "are", "pearls"]
\]

\[
\text{[(1,"Those"),(2,"are"),(3,"pearls")]}\]

... Another

Using an infinite list ensures there are enough indices:

\[
\text{indexList} \ \text{x}s = \text{zip} \ [\ldots] \ \text{x}s
\]

Here, \text{zip} is a selector.
Recursive Definitions

The Fibonacci sequence can be recursively defined as follows:

```haskell
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...)

```haskell
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):

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

Recursive Constants

Constants can be recursively defined:

```haskell
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)]
```

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:

\[
\text{primes = eratosthenes } [2..]
\]

Eratosthenes' Sieve II

To remove multiples of a number \( n \) from a list \( l \):

\[
\text{sieve } n = \text{filter } (\neq 0). (\mod \ n)
\]

Eratosthenes' Sieve III

To generate one prime:

\[
\text{next } n \text{ ps } = n : \text{sieve } n \text{ ps}
\]

To generate all primes:

\[
\text{eratosthenes } = \text{foldr } \text{next } \[]
\]

Eratosthenes' Sieve IV

To generate all primes:

\[
\text{primes } = \text{sieveAll } [2..]
\]

where

\[
\text{sieveAll } (p:\text{ns})
\]

\[
= p : \text{sieveAll } (\text{sieve } p \text{ ns})
\]

Summary

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

Next: Introducing OBJ