Lecture 4

The Sieve of Eratosthenes.

In this lecture, we look at an algorithm to find primes which you might have learned in school. It is called the Sieve of Eratosthenes. The basic idea is the following.

1. Enumerate all positive integers starting from 2.
2. Forever do the following
3. Take the smallest of the remaining uncrossed integer say p and circle it.
4. Cross out all numbers that are the factors of the circled integer p.

All the circled integers are the list of primes. For an animated description see the wiki link `http://en.wikipedia.org/Sieve_of_Eratosthenes`.

We want to convert this algorithm to haskell code. Notices that the Sieve seperates the primes from the composites. The primes are those that are circled and composites are those that are crossed. So let us start by defining

```
> primes = circledIntegers
```

i.e. the primes are precisely the list of circled integers.

An integer gets circled if and only if it is not crossed at any stage of the sieving process. Furthermore, a integer gets crossed in the stage when its least prime factor is circled. So to check whether an integer is crossed all we need to do is to check whether there is a prime which divides it.

```
> divides x n = n `mod` x == 0
>
> check (x:xs) n | x * x > n      = True       -- the rest are bigger.
>                | x `divides` n  = False      -- We hit a factor
>                | otherwise      = check xs n -- need to do more work
>
> isCircled = check primes
>
```

The function `isCircle x` checks whether `x` will eventually be circled. One need not go over all primes as the smallest non-trivial prime p that divides a composite number n should always be less than or equal to $\sqrt{n}$. This explains the first guard statement of the check function.

4.1 Guards detour.

We now explain another feature of Haskell namely guards. Look at the definition of the function `check`. Recall that a function is defined by giving a set of equations. For each such equation, we can have a set of guards. The syntax of these guarded equation looks like

```
f p_1 ... p_n | g_1  = e_1
              | g_2  = e_2
              | g_3  = e_3
              | ...
              | g_m  = e_m
```

Each of the guards `g_i` is a boolean expression. You should read this as "if `f`'s arguments match the patterns `p1 ... pn` then its value is `e_1` if `g_1` is true, `e_2` if `g_2` is true, etc `e_m` if `g_m` is true". If multiple guards are true then the guard listed first has priority. For example consider the following function

```
>
> f x | x >= 0 = "non-negative"
>     | x <= 0 = "negative"
```

Then `f 0` is the string `"non-negative"`. If you want to add a default guard, then use the keyword `otherwise`. The keyword `otherwise` is nothing but the boolean value `True`. However, in guards it is a convention to write `otherwise` instead of `True`.

Finally, we want to define the list of `circledInteger`. Clearly the first integer to be circled is 2. The rest are those integers on which the function `isCircled` returns true. And here is the Haskell code.

```
>
> circledIntegers = 2 : filter isCircled [3..]
>
```

Here filter is a function that does what you expect it to do. Its first argument is a predicate, i.e it take an element and returns a boolean, and its second argument is a list. It returns all those elements in the list that satisfies the predicate. The type of filter is given by `filter :: (a -> Bool) -> [a] -> [a]`. This completes the program for sieving primes. Now load the program into the interperter

```
$ ghci Sieve-of-Eratosthense.lhs
GHCi, version 7.0.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
[1 of 1] Compiling Main             ( src/lectures/Sieve-of-Eratosthenes.lhs, interpreted )
Ok, modules loaded: Main.
*Main> :type take
take :: Int -> [a] -> [a]
*Main> take 10 primes
[2,3,5,7,11,13,17,19,23,29]
```

The `take n xs` returns the first `n` elements of the list `xs`.

4.2 How is the circularity of primes handled?

One thing you might have noticed is that the list `primes` has a circular definition. The compiler is able to grok this circularity due to the fact that Haskell is a lazy language. No expression is evaluated unless it is required. For each integer in the list `primes` to decide that it is circled, we need to consider only the primes that are less than its square root. As a result the definition of primes does not blow up into an infinitary computation.

4.3 Infix application of function and prefix application of operators.

We now explain another feature of a Haskell. Notice the use of `mod` in the expression n ‘mod‘ x == 0 in the definition of the function `divides` and in the guard x ‘divides‘ n in the definition of the function `check`. We have used a two argument function (i.e. its type is `a -> b -> c`) like an operator. For any such function `foo` we can convert it into a binary operator by back quoting it, i.e. ‘foo‘.

The converse is also possible, i.e. we can convert an operator into the corresponding function by enclosing it in a bracket. For example the expression `(+)` (there is no space in side the bracket otherwise it is an error) is a function `Int -> Int -> Int`.

```
> incr1 = map ((+) 1)
> incr2 = map (+1)
```

The two function `incr1` and `incr2` both does the same thing; increments all the elements in a list of integers by 1.