Lecture 1

Informal Introduction to Haskell

Chapters 1 and 2 of The Haskell Road.

Lazy lists

```
sentence = "Sentences can go " ++ onAndOn
onAndOn  = "on and " ++ onAndOn
```

Try this out with

```
take 65 sentence
```

Lazy lists

```
sentence = "Sentences can go " ++ onAndOn
onAndOn  = "on and " ++ onAndOn
```

Try this out with

```
take 65 sentence
```

"Sentences can go on and on and on and on and on and on and on and"

Lazy lists

Next consider:

```
sentences = "Sentences can go on":
            map (++ " and on") sentences
```

Try this out with

```
take 10 sentences
```

Lazy lists

Next consider:

```
sentences = "(Large) Sentences can go on":
          map (++ " and on") sentences
```

Try this out with

take 10 sentences

["(Large) Sentences can go on","(Large) Sentences can go on and on","(Large) Sentences can go on and on and on","(Large) Sentences can go on and on and on and on","(Large) Sentences can go on and on and on and on and on","(Large) Sentences can go on and on and on and on and on and on","(Large) Sentences can go on and on and on and on and on and on and on","(Large) Sentences can go on and on and on and on and on and on and on and on","(Large) Sentences can go on and on and on and on and on and on and on and on and on","(Large) Sentences can go on and on and on and on and on and on and on and on and on and on"]

```
*Lecture1> :t sentence
sentence :: [Char]
*Lecture1> :t sentences
sentences :: [[Char]]
```

Review question

Can you give your own definitions of map and take?

Review question

Can you give your own definitions of map and take?

```
*Lecture1> :t map
map :: (a -> b) -> [a] -> [b]

*Lecture1> :t take
take :: Int -> [a] -> [a]
```

First Order logic formulas and functional programs

Example domain: the natural numbers
Example properties: being even, odd, prime, 3-fold, etc

```
threefold :: Integer -> Bool
threefold n = rem n 3 == 0
```

Why does threefold express a property of integer numbers?
What is the most general specification of a property?

The lazy list of natural numbers that are threefolds:

```
threefolds = filter threefold [0..]
```

Review Question

Can you give your own definition of filter?

Review Question

Can you give your own definition of filter?

```
*Lecture1> :t filter
filter :: (a -> Bool) -> [a] -> [a]
```

How to express things with predicate logic

there is no largest natural number

there is a smallest natural number

Translations use any and all but they will run forever

```
nats = [0..]

query1 = all (\ n -> any (\ m -> n < m) nats) nats

query2 = any (\ n -> all (\ m -> n <= m) nats) nats
```

Let’s make it look more “natural”

```
forall = flip all
exist  = flip any

query1' = forall nats (\ n -> exist nats (\ m -> n < m))

query2' = exist nats (\ n -> forall nats (\ m -> n <= m))
```

Let’s make it look more “natural”

```
forall = flip all
exist  = flip any

query1' = forall nats (\ n -> exist nats (\ m -> n < m))

query2' = exist nats (\ n -> forall nats (\ m -> n <= m))
```

where

```
flip :: (a -> b -> c) -> b -> a -> c
```

Review question

Give your own definition of all.

Review question

Give your own definition of all.

```haskell
myall :: (a -> Bool) -> [a] -> Bool
myall p [] = True
myall p (x:xs) = p x && myall p xs
```

One way to see that this is correct is by giving an inductive proof. (See Workshop 1 on Induction and Recursion.)

Another way to check the implementation is by means of a test.

Using Quickcheck

Let's try and test this with QuickCheck.
The method is to write a test property. The obvious property is:

```
\ p xs -> all p xs == myall p xs
```

Will this work?

Quickcheck

Unfortunately, **not**. QuickCheck is designed to display counterexamples, but there is no general way to display functions. So?

```
*Lecture1> quickCheckResult (\ p xs -> all p xs == myall p xs)

<interactive>:4:1:
    No instance for (Show (a0 -> Bool))
      arising from a use of ‘quickCheckResult’
    In the expression:
      quickCheckResult (\ p xs -> all p xs == myall p xs)
    In an equation for ‘it’:
        it = quickCheckResult (\ p xs -> all p xs == myall p xs)
```

Quickcheck

Here is a general conversion from lists to properties:

```
list2p :: Eq a => [a] -> a -> Bool
list2p = flip elem
```

Now we can define:

```
myallTest :: [Int] -> [Int] -> Bool
myallTest = \ ys xs -> let p = list2p ys in
  all p xs == myall p xs
```

Specialising to [Int] is necessary, as QuickCheck can only test for specific types.
Test it like this:

```
*Lecture1> quickCheck myallTest
 +++ OK, passed 100 tests.
```

general recursion pattern

Definition of myall:

- you have to specify a value for the base case,
- you have to give a definition for the recursive case (x:xs) using
  - the first element of the list x and
  - a recursive call for the tail of the list xs.

Generalization with foldr makes this recursion recipe explicit.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Quickcheck

Using this we can give a home-made definition of all as follows:

```
myall' p = foldr (\ x b -> p x && b) True
```

Again, test it with QuickCheck:

```
myallTest' :: [Int] -> [Int] -> Bool
myallTest' = \ ys xs -> let p = list2p ys in
  all p xs == myall' p xs

*Lecture1> quickCheck myallTest'
 +++ OK, passed 100 tests.
```

Playing with Primes

Definition of being a prime number with a formula of predicate logic:

$$P(n) :\equiv n \in N \land n > 1 \land \forall d \in N (1 < d < n \rightarrow \neg D(d,n)).$$

In words:
n is prime :≡ n is a natural number and n>1 and for all natural numbers d with 1<d<n it holds that d does not divide n.

What do we need first to implement this?

Playing with Primes

Definition of being a prime number with a formula of predicate logic:

$$P(n) :\equiv n \in N \wedge n > 1 \wedge \forall d \in N (1 < d < n \rightarrow \neg D(d,n)).$$

In words:
n is prime :≡ n is a natural number and n>1 and for all natural numbers d with 1<d<n it holds that d does not divide n.

What do we need first to implement this?
We need is the divide relation:

```
divide :: Integer -> Integer -> Bool
divide n m = rem m n == 0
```

Playing with Primes

Next, we write the definition, using the Haskell implementations of &&, all and not.

```
isPrime :: Integer -> Bool
isPrime n = n > 1 && all (\ d -> not (divide d n)) [2..n-1]
```

Of course, this is not efficient.

The point is that Haskell has all, not, any, &&, ||, etc, to express all the constructs of first order logic.

Take home (not graded) exercise:

Are you able to write the types of the predicate logic operators?

Improving the definition

Here is a slightly more efficient version of isPrime:

```
isPrime' :: Integer -> Bool
isPrime' n = all (\ x -> rem n x /= 0) xs
  where xs = takeWhile (\ y -> y^2 <= n) [2..]
```

Why is this better?

Improving further

```haskell
prime :: Integer -> Bool
prime n = n > 1 && all (\ x -> rem n x /= 0) xs
  where xs = takeWhile (\ y -> y^2 <= n) primes

primes :: [Integer]
primes = 2 : filter prime [3..]
```

Why is this even better?

Why do we need to give the starting value 2 explicitly?

Eratosthenes’ Sieve

```
sieve :: [Integer] -> [Integer]
sieve (n:ns) = n : sieve (filter (\m -> rem m n /= 0) ns)

eprimes = sieve [2..]
```

Least natural number having a given property

Start the search with 0.

Note: if no number satisfies the property, the query will run forever.

```
least :: (Integer -> Bool) -> Integer

least p = head (filter p nats)
```

Least natural number having a given property

Start the search with 0.

Note: if no number satisfies the property, the query will run forever.

```
least :: (Integer -> Bool) -> Integer

least p = head (filter p nats)
```

Alternative

```
least1 p = lst p 0
    where lst p n = if p n then n else lst p (n+1)
```

Prime Pairs

A prime pair is a pair (p,p+2) with the property that both p and p+2 are primes. The first prime pair is (3,5).

Implement a function for generating prime pairs, and use this to find the first 100 prime pairs.

Prime Pairs

A prime pair is a pair (p,p+2) with the property that both p and p+2 are primes. The first prime pair is (3,5).

Implement a function for generating prime pairs, and use this to find the first 100 prime pairs.

```
dif2 :: [Integer] -> [(Integer,Integer)]
dif2 (p:q:rs) = if p + 2 == q then (p,q) : dif2 (q:rs)
                else dif2 (q:rs)

primePairs = dif2 primes
```

```
Lecture1> take 100 primePairs
 [(3,5),(5,7),(11,13),(17,19),(29,31),(41,43),(59,61),(71,73),
 (101,103),(107,109),(137,139),(149,151),(179,181),(191,193),
 (197,199),(227,229),(239,241),(269,271),(281,283),(311,313),
 (347,349),(419,421),(431,433),(461,463),(521,523),(569,571),
 (599,601),(617,619),(641,643),(659,661),(809,811),(821,823),
 (827,829),(857,859),(881,883),(1019,1021),(1031,1033),
 (1049,1051),(1061,1063),(1091,1093),(1151,1153),(1229,1231),
 (1277,1279),(1289,1291),(1301,1303),(1319,1321),(1427,1429),
 (1451,1453),(1481,1483),(1487,1489),(1607,1609),(1619,1621),
 (1667,1669),(1697,1699),(1721,1723),(1787,1789),(1871,1873),
 (1877,1879),(1931,1933),(1949,1951),(1997,1999),(2027,2029),
 (2081,2083),(2087,2089),(2111,2113),(2129,2131),(2141,2143),
 (2237,2239),(2267,2269),(2309,2311),(2339,2341),(2381,2383),
 (2549,2551),(2591,2593),(2657,2659),(2687,2689),(2711,2713),
 (2729,2731),(2789,2791),(2801,2803),(2969,2971),(2999,3001),
 (3119,3121),(3167,3169),(3251,3253),(3257,3259),(3299,3301),
 (3329,3331),(3359,3361),(3371,3373),(3389,3391),(3461,3463),
 (3467,3469),(3527,3529),(3539,3541),(3557,3559),(3581,3583),
 (3671,3673),(3767,3769),(3821,3823)]
```

Review Question

Observe that it holds for all of these pairs (x,x+2) but for the first pair that x+1 (the number in between) is divisible by 3.

Why is this so?

Prime Triples

Assume a prime triple is a triple (n,m,k) such that n<m<k and n,m,k are all prime, and n and k differ by six. The first prime triple is (5,7,11).

Implement a function for generating prime triples.
Use this to find the first 100 prime triples.
How about (p, p+2, p+4)?

```
dif6 :: [Integer] -> [(Integer,Integer,Integer)]
dif6 (p:q:r:ss) = if p + 6 == r then (p,q,r) : dif6 (q:r:ss)
                  else dif6 (q:r:ss)

primeTriples = dif6 primes

sol = take 100 primeTriples
```

Next Prime

Implement a function nextPrime with the property that nextPrime n returns n when n is prime, and the next prime after n otherwise.

```
nextPrime :: Integer -> Integer
nextPrime n = if prime n then n else nextPrime (n+1)
```

More curio - Mersenne numbers

A *Mersenne number* is a natural number of the form 2^p-1 where p is a prime number. A Mersenne prime is a Mersenne number that is prime.

Write a function for generating Mersenne primes. How far do you get? Note: only 48 Mersenne primes are known. See here for details.

```
mersenne :: [Integer]
mersenne = [ p | p <- primes, prime (2^p - 1) ]
```

More curio - Pythagorean triples

A Pythagorean triple is a triple of natural numbers (x,y,z) with the property that $x^2+y^2=z^2$. The smallest example is (3,4,5).
Implement a Haskell function that generates Pythagorean triples.
Are there Pythagorean triples (x,y,z) with x=y?
If your answer is "Yes", give the smallest one. If your answer is "No", explain why this is impossible.

```
pythTriples :: [(Integer,Integer,Integer)]
pythTriples = filter (\ (x,y,z) -> x^2 + y^2 == z^2)
  [ (x,y,z) | z <- [1..], x <- [1..z], y <- [1..z], x < y  ]
```

Lab Info and Assignment Deadline

Sunday, 10.09, 6pm

Create a separate directory in your repo containing your submission ready version

- A melange of various contributions from the team members
- Argue why a specific implementation has been chosen
- Each team member has a personal directory in the repository
  - Does not matter if you cannot solve all across all assignments
- Please write your time next to each solution