EECS 776

Functional Programming and Domain Specific Languages

Professor Gill

The University of Kansas

Jan 29th 2020

Homework 1

Due Monday 3rd February

- Find a ghci interpreter
- Try out the following example
  - Write a function to compute the surface area of a sphere
  - Run the function on a small number of inputs
  - **Bring your interaction with ghci (what you typed, what ghci said back)**
  - Remember to use the 776 coversheet

Starting Haskell

```
$ ghci
GHCi, version 7.8.2: http://www.haskell.org/ghc/  :? for
help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude>
```

(I will sometimes use **`GHCi>`** to denote the GHCi prompt.)

```
Prelude> 42
42
Prelude> 1 + 1
2
Prelude> "Hello"
"Hello"
```

This Haskell prompt is **much** more powerful than it appears.

Numbers and Arithmetic

Haskell supports integers and floating point numbers (like almost every language on the planet, except JavaScript). Haskell also supports arbitrary precision numbers, and standard arithmetical operations.

```
GHCi> 1 + 2
3
GHCi> 3 * 2.5
7.5
GHCi> 2 ^ 100
1267650600228229401496703205376
```

Division is slightly quirky.

```
GHCi> 99 / 5
19.8
GHCi> 99 `div` 5
19
GHCi> 99 `mod` 5
4
```

For integers, code **`div`** is like / in C-like languages, and **`mod`** is like % in C-like languages.

GHCi & Bindings

Expressions only get you so far. You can name things, using `let`

```
GHCi> let x = 4
GHCi> x
4
GHCi> x + 2
6
```

We can now do basic math, like the area of a pizza.

```
GHCi> let r = 12.0
GHCi> r
12.0
GHCi> pi * r ^ 2
452.3893421169302
```

We can also **scope** the binding of **`r`** to a single line.

```
GHCi> let r = 12.0 in pi * r ^ 2
452.3893421169302
```

Lists and Tuples (Compounds)

There are various ways of building compound values, including strings, lists and tuples.

```
GHCi> "Hello" ++ "World"
???
GHCi> ("Hello" ++ "World", 4 * 7)
???
GHCi> [1,2,3,4]
???
GHCi> [1..10]
???
GHCi> take 5 [1..10]
???
GHCi> drop 5 [1..10]
???
GHCi> let xs = 1 : 2 : xs
???
GHCi> take 10 xs
???
```

What will these expressions evaluate to?

Functions

Functions are values, and can also be bound at the command line

```
GHCi> let f x = x * x
GHCi> f 10
100
GHCi> let area r = pi * r ^ 2
GHCi> area 12
452.3893421169302
GHCi> area 10
314.1592653589793
```

The advantage of functions is that they are reusable. **`area`** can be used later, many times.

Combining functions and lists

A function can be called many times using **map**

```
GHCi> let double n = n + n
GHCi> map double [1..10]
[2,4,6,8,10,12,14,16,18,20]
```

Basic arithmetic can also be called many times.

```
GHCi> map (*2) [1..10]
[2,4,6,8,10,12,14,16,18,20]
GHCi> map (+1) (map (*2) [1..10])
[3,5,7,9,11,13,15,17,19,21]
```

Guess what does this does:

```
GHCi> filter odd [1..10]
????
```

Computing Primes

First compute the numbers between 2 and one less than a value

```
GHCi> let inside x = [2..x-1]
GHCi> inside 10
[2,3,4,5,6,7,8,9]
```

Next, compute the factors of a number.

```
GHCi> let divExactly n m = n `mod` m == 0
GHCi> let factors n = filter (divExactly n) (inside n)
GHCi> factors 12
[2,3,4,6]
```

Now we can compute prime

```
GHCi> let isPrime n = length (factors n) == 0
GHCi> isPrime 11
True
GHCi> isPrime 12
False
```

Filtering for Primes

We can use the **`filter`** trick here as well.

```
GHCi> filter isPrime [2..100]
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,7
```

What does this do?

```
GHCi> filter isPrime [2..]
???
```

GHCi & Bindings

Remember, you can name things, using **`let`**

```
GHCi> let x = 4
GHCi> x
4
GHCi> x + 2
6
```

This is binding, not assignment. Think naming a child, not variable assignment.

GHCi & Bindings (2)

```
GHCi> let x = 4
GHCi> let y = x + 1
GHCi> let x = 9
GHCi> x
9
GHCi> y
???
```

What does **y** evaluate to?

5, because **y** is bound when **x = 4**

Comprehensions

Comprehensions in Haskell are like set comprehensions in math.

$$\{ x \mid x \leftarrow [1..10], \text{odd } x \}$$

```
GHCi> [ x | x <- [1..10], odd x ]
[1,3,5,7,9]
```

What might this do?

```
GHCi> [ x * x | x <- [1..10], odd x ]
????
```

Primes

```
GHCi> let factors n = [ x | x <- [2..n-1], n `mod` x == 0 ]
GHCi> factors 12
[2,3,4,6]
```

Now we can compute is a number is prime.

```
GHCi> let isPrime n = length (factors n) == 0
GHCi> isPrime 11
True
GHCi> isPrime 12
False
```

We can also compute with groups of primes.

Primes (2)

```
GHCi> let factors n = [ x | x <- [2..n-1], n `mod` x == 0 ]
GHCi> let isPrime n = length (factors n) == 0
```

We can also compute with groups of primes.

```
GHCi> filter isPrime [2..100]
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,8
GHCi> map isPrime [2..12]
[True,True,False,True,False,True,False,False,False,True,False]
GHCi> [ if isPrime x then 'x' else '.' | x <- [2..50]]
"xx.x.x...x.x...x.x...x.....x.x.....x...x.x...x.....x.....x.x.....x
```

Primes improved

```
GHCi> let isPrime n = length (factors n) == 0
GHCi> :set +s
GHCi> isPrime 12345678
False
(8.98 secs, 2569789760 bytes)
```

How can we optimize this? The **`null`** function checks to see if a list is empty, so we use this instead of checking the length.

```
GHCi> let isPrime' n = null (factors n)
GHCi> isPrime' 12345678
False
(0.01 secs, 2096496 bytes)
```

We turn off the timing with :unset.

```
GHCi> :unset +s
```

Testing primes improved

We want to check if **`isPrime`** and **`isPrime'`** are equal.

```
GHCi> let isPrime n = length (factors n) == 0
GHCi> let isPrime' n = null (factors n)
```

So how do we compare a function?

A function is equal if for all inputs, the result is always the same.

```
GHCi> and [ isPrime' x == isPrime x | x <- [2..100]]
True
```

This is not comprehensive, but better than no tests. We will see how to do automatic random test-case generation later.

Problem: Haskell sessions are lost on exit

We can take our declarations, and put them into a file. Then load the file.

```
module Primes where

factors n = [ x | x <- [2..n-1], n `mod` x == 0 ]
isPrime n = length (factors n) == 0
isPrime' n = null (factors n)
```

Note how the **`let`** has been dropped. This is because we only have declarations.
Back at the prompt, we can load this module.

```
Prelude> :l Primes
[1 of 1] Compiling Primes           ( Primes.hs, interpreted )
Ok, modules loaded: Primes.
*Primes> isPrime 12
False
```