Project Euler

Dr. Mattox Beckman

University of Illinois at Urbana-Champaign
Department of Computer Science

Our Environment

- Create a new directory for this project.
- Using an editor, create a file `euler.hs`. Add the following two lines to it:

```haskell
inc :: Integer -> Integer
inc x = x + 1
```

- Now type `stack repl euler.hs`.

Expected Output

```
% stack repl euler.hs

[1 of 1] Compiling Main             ( /home/mattox/euler/euler.hs, interpre
Ok, one module loaded.
Loaded GHCi configuration from /run/user/1000/ghci4772/ghci-script
*Main> :t inc
inc :: Integer -> Integer
*Main> inc 10
11
*Main> :r
Ok, one module loaded.
*Main>
Leaving GHCi.
```

Problem 1 – Multiples of 3 or 5

If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6, and 9. The sum of these multiples is 23. Find the sum of all the multiples of 3 or 5 below 1000.

Some Arithmetic

```
*Main> mod 10 3
1
*Main> 10 `mod` 3
1
*Main> 10 `mod` 3 == 0
False
*Main> 10 `mod` 3 == 0 || 10 `mod` 5 == 0
True
*Main> mod3or5 x = x `mod` 3 == 0 || x `mod` 5 == 0
*Main>
```

Add the following line to your `euler.hs`.

```
mod3or5 x = x `mod` 3 == 0 || x `mod` 5 == 0
```

A Type Constraint

- HASKELL will *infer* the types of things if you don't specify them!
- Let's see what it thinks of our new function ...

```
*Main> :t mod3or5
mod3or5 :: Integral a => a -> Bool
```

A Type Constraint

- Haskell will *infer* the types of things if you don't specify them!
- Let's see what it thinks of our new function …

```
*Main> :t mod3or5
mod3or5 :: Integral a => a -> Bool
```

- "The input can be any type that is an Integral, and the output is a Bool."

List Operations

```
*Main> [3,5,7,9]
[3,5,7,9]
*Main> map inc [3,5,7,9]
[4,6,8,10]
*Main> map mod3or5 [3,5,7,9]
[True,True,False,True]
*Main> filter mod3or5 [3,5,7,9]
[3,5,9]
*Main> sum (filter mod3or5 [3,5,7,9])
17
```

Put the following definition into your `euler.hs`:

```
sumMods xx = sum (filter mod3or5 xx)
```

So how do we get a list from 1 to 999?

Big Lists

```
*Main> [1..20]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
*Main> [1,3..20]
[1,3,5,7,9,11,13,15,17,19]
*Main> [1,5..20]
[1,5,9,13,17]
*Main> [1..999]   -- Go ahead and try it!
```

So now add this line to your `euler.hs`:

```
euler1 = sumMods [1..999]
```

Sample Run

```
*Main> :r
[1 of 1] Compiling Main             ( /home/mattox/euler/euler.hs, interp
Ok, one module loaded.
*Main> euler1
233168
```

Final Result

```haskell
inc :: Integer -> Integer
inc x = x + 1

  -- Euler Problem 1

mod3or5 x = x `mod` 3 == 0 || x `mod` 5 == 0
sumMods xx = sum (filter mod3or5 xx)
euler1 = sumMods [1..999]
```

We can clean this up a little …

Cleaner Version

```haskell
inc :: Integer -> Integer
inc x = x + 1

  -- Euler Problem 1

euler1 = sumMods [1..999]
  where mod3or5 x = x `mod` 3 == 0 || x `mod` 5 == 0
        sumMods xx = sum (filter mod3or5 xx)
```

- The `where` keyword introduces local definitions.
- Indentation determines the scope of definitions.
- Be sure your editor never uses tabs!!

Euler Problem 3 – Prime Factors

The prime factors of 13195 are 5, 7, 13, and 29. What is the largest prime factor of the number 600851475143?

Sectioning

```
*Main> plus a b = a + b
*Main> :t plus
plus :: Num a => a -> a -> a
*Main> plus 10 20
30
*Main> :t (plus 1)
(plus 1) :: Num a => a -> a
*Main> addTwo = plus 2
*Main> addTwo 10
12
```

- You can also say things like `(+1)` to get a partially applied operator.

The Sieve

- We will make something like the Sieve of Eratosthenes.

```
*Main> notDivides a n = n `mod` a /= 0
*Main> notDivides 2 10
False
*Main> notDivides 3 10
True
*Main> filter (notDivides 3) [1..10]
[1,2,4,5,7,8,10]
```

Go ahead and add the definition of `notDivides` to your file.

Building Up Lists

- The operator `:` creates a list from an element and another list.
- HASKELL "`a:b`" is like JAVA/C++ "`new Node(a,b)`."
- The built-in function `head` will get you the first element of a list.

```
*Main> 2 : filter (notDivides 2) [2..20]
[2,3,5,7,9,11,13,15,17,19]
*Main> 2 : filter (notDivides 2)
              (3 : filter (notDivides 3) [2..20])
[2,3,5,7,11,13,17,19]
*Main> 2 : filter (notDivides 2)
             (3 : filter (notDivides 3)
                (5 : filter (notDivides 5) [2..20]))
[2,3,5,7,11,13,17,19]
```

- We need a recursive solution for this!

Making the Sieve

```
sieve (x:xs) = x : (sieve (filter (notDivides xs) xs))

primes = sieve [2..]
```

Sample Run

```
*Main> sieve [2..20]
[2,3,5,7,11,13,17,19,*** Exception: Prelude.head: empty list
*Main> take 20 (primes)
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71]
*Main> take 20 $ primes
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71]
```

Factors

- Now to get the factors ...

```
factors n = aux n primes
  where aux 1 _        = []
        aux n (p:ps) = case divMod n p of
                           (n', 0) -> p : aux n' ps
                           (_ , _) -> aux n ps

maxFactor n = foldr1 max $ factors n

euler3 = maxFactor 600851475143
```

```
*Main> foldr1 (+) [2,3,4,5]
14
*Main> euler3
6857
```

Problem 20 – Factorial Digit Sum

$n!$ means $n \times (n-1) \times \cdots \times 3 \times 2 \times 1$.

For example, $10! = 10 \times 9 \times \cdots \times 3 \times 2 \times 1 = 3628800$, and the sum of the digits in the number $10!$ is $3 + 6 + 2 + 8 + 8 + 0 + 0 = 27$.

Find the sum of the digits in the number $100!$

BigInts

- Most functional languages have "Big Integers," constrained only by your computer's memory.
- To get started, here's the definition for factorial:

```
fact 0 = 1
fanc n = n * fact (n-1)
```

- If we run this on 100 it actually works!

```
*Main> fact 100
933262154439441526816992388562667004907159682643816214685
929638952175999932299156089414639761565182862536979208272
2375825118521091686400000000000000000000000000
```

Divide and Conquer!

- To get the least significant digit, just take a modulus!
- To divide by 10 without remainder, just use `div`.

```
sumDigits 0 = 0
sumDigits n = n `mod` 10 + sumDigits (n `div` 10)

euler20 = sumDigits $ fact 100
```

Now try …

```
*Main> euler20
648
```