Functional Programming

:l foo “Haskell”

Markos Vassilis, Mediterranean College

Week 02

Haskell Fun (Part I)

Do You Have a Credit Card?

- If not so, Haskell cannot help you with that…
- However, how do you think third–party providers verify your credit (or whatever) card is valid?
  - By checking on a vast credit card database?
  - No!
  - By actually computing a checksum value:
    - Your credit card number is squeezed into a single number using a mysterious function.
    - Then, this squeezed version must adhere to some restrictions.
    - If met, then your card is valid. If not, it's not.
- So, as you might suspect, our next task will be to implement a thing like a credit card validator in Haskell!

An Example First

We will follow the algorithm below to verify credit cards:

- Double the value of every second digit beginning from the right, e.g., [4, 8, 5, 6] becomes [8, 8, 10, 6].
- Add all digits of the 'doubled" list, e.g., [8, 8, 10, 6] → 8 + 8 + 1 + 0 + 6 = **23**.
- Calculate the remainder modulo 10, e.g., → 23 mod 10 = 3.
- If the remainder is 0 then the card is valid, otherwise it is not.

Write a Haskell script that implements the above algorithm. The following should print `True` and `False`, respectively:

```
main :: IO()
main = print [(x, is_valid x) | x <- [4012888888881881, 4012888888881882]]
```

Hint #01: Ints To Digits

Define the functions:

```
toDigits :: Integer -> [Integer]
toDigitsRev :: Integer -> [Integer]
```

that convert an integer to a list of its digits in right and reverse order, respectively.

```
Example: toDigits 1234 == [1,2,3,4]
Example: toDigitsRev 1234 == [4,3,2,1]
Example: toDigits 0 == []
Example: toDigits (-17) == []
```

Hint #02: Double Every Other

Then, define the function:

```
doubleEveryOther :: [Integer] -> [Integer]
```

that doubles every other digit, **starting from right!**

```
Example: doubleEveryOther [8,7,6,5] == [16,7,12,5]
Example: doubleEveryOther [1,2,3] == [1,4,3]
```

Hint #03: Sum All Digits

Define the function:

```
sumDigits :: [Integer] -> Integer
```

that sums the contents of the provided list of digits.

```
Example: sumDigits [16,7,12,5] = 1 + 6 + 7 + 1 + 2 + 5 = 22
```

Hint #04: Validate

Define the function:

```
is_valid :: Integer -> Bool
```

that takes an integer and validates whether it is a valid credit card number.

```
Example: validate 4012888888881881 = True
```

```
Example: validate 4012888888881882 = False
```

`where?`

- As we have already said, there is no concept of mutation in functional programming, so Haskell does not support it.
- However, writing lengthy expressions is quite frustrating as a process…
- …so, we can define some shorthands for complex expressions instead of writing Haskell spaghetti code all the time.
- This is where `where` comes in handy:
  - It allows us to define “variables” for more complex expressions, as with m in the above example.
  - Note that everything defined within a `where` is valid for the scope of that specific function `where` lives in.

Primes (Again)

A number, n, is said to be prime if:

- n > 1, and;
- its only divisors are 1 and n.

Write a Haskell function that takes a single integer as an input and returns:

- True, if the number is prime;
- False, otherwise.

Primality Check

A simple Haskell program to check primality:

```
main :: IO()
main = print ([(x, is_prime x) | x <- [1..30]])

is_prime :: Int -> Bool
is_prime 1 = False
is_prime n = sum [1 | c <- [2..(n-1)], rem n c == 0] == 0
```

Anything faster?

A Bit Better

```
main :: IO()
main = print ([(x, is_prime x) | x <- [1..200]])

is_prime :: Int -> Bool
is_prime 1 = False
is_prime n = sum [1 | c <- [2..m], rem n c == 0] == 0
  where m = floor (sqrt (fromIntegral n))
```

Just be careful with where and indentation! (roughly, as with Python)

Even Better

So far, we have not handled the case where the input might be non–positive. To that end we can use guards:

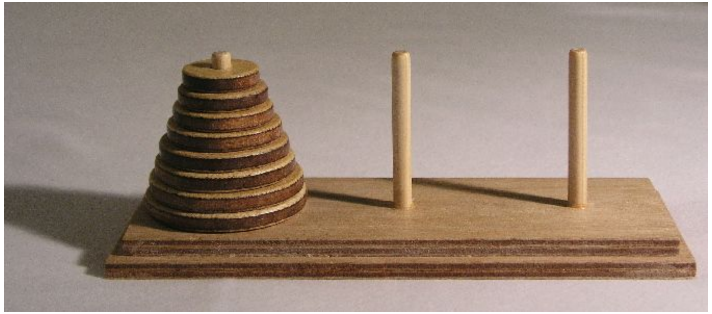
```haskell
isPrimeG :: Int -> Bool
isPrimeG n | n < 2 = False
           | otherwise = length [x | x <- [1..m], rem n x == 0] == 1
            where m = floor (sqrt (fromIntegral n))
```

- Think of guards as the Haskell equivalent for `if-elif-...-else` statements of Python.
- `otherwise` is the “else” keyword, evaluating always to True.
- As with `if-elif-...-else` statements, **order matters**, since guard expressions are evaluated **top–to–bottom**.

(Reminder) Towers of Hanoi

This is a classic puzzle involving three towers (pegs) and a set of disks of different sizes. The objective is to move all the disks from one peg to another, following these rules:

- Only one disk can be moved at a time.
- A larger disk cannot be placed on top of a smaller one.



(Reminder) A Line of Attack

Consider the following (name the pegs A, B, C, right to left):

- For n = 1 disk you just need to move that one disk from A to C.
- For n = 2 disks you first move disk 1 from A to B, then disk 2 from A to C and then disk 1 from B to C.
    - This is essentially moving disk 1 to B and then solving the problem for n-1 (=1) twice.
- For n = 3 disks you first solve the n = 2 case by moving disks 1 and 2 to B, then move disk 3 to C and solve again the n = 2 case by moving disks 1 and 2 to C, on top of disk 3.

(Reminder) A Line of Attack

- For n = 4 disks you:
  - First move disks 1, 2, 3 to B, solving the n = 3 case.
  - Then move disk 4 to C.
  - Then move disks 1, 2, 3 to C, solving again the n = 3 case.
- For n = 5 disks you:
  - First move disks 1, 2, 3, 4 to B, solving the n = 4 case.
  - Then move disk 5 to C.
  - Then move disks 1, 2, 3, 4 to C, solving again the n = 4 case.
- Can you see a pattern here?

(Reminder) A Line of Attack

In general, in order to solve the problem for any n:

- First move the first n - 1 disks from A to B (the auxiliary peg).
- Then move disk n to the target peg, C.
- Then move the first n - 1 disks from B to C.

The above has a clearly recursive structure, as while solving the case for n we have to solve the case for n - 1.

(Reminder) Recursion of Hanoi?

Can you think of a recursive function that solve the Towers of Hanoi problem?

```python
def tower_of_hanoi(n, source, destination, auxiliary):
    if n == 1:
        print(f"Move disk 1 from {source} to {destination}")
        return
    tower_of_hanoi(n-1, source, auxiliary, destination)
    print(f"Move disk {n} from {source} to {destination}")
    tower_of_hanoi(n-1, auxiliary, destination, source)

if __name__ == "__main__":
    tower_of_hanoi(3, "A", "C", "B")
```

Now In Haskell

Implement the above solution in Haskell:

- You might choose recursion or any other way you wish.
- After completing the above, compare the Python and Haskell solutions.
- Then, consider crafting a solution for a 4-peg variant of the puzzle: instead of three pegs, you have four, so the problem should now be solvable in far less moves.
- Can you find the optimal solution in the above?

Haskell Resources

Some interesting and really cool Haskell resources (not as cool as Haskell, though):

- https://learnyouahaskell.com/ – large parts of our course will be based on this book, so, give it a read yourselves!
- A list of Haskell resources I could not copy–paste here due to spacing: https://wiki.haskell.org/Learning_Haskell
- Some notes: https://www.seas.upenn.edu/~cis1940/spring13/lectures.html
- For visual learners: https://www.youtube.com/playlist?list=PLF1Z-APd9zK7usPMx3LGMZEHrECUGodd3
- The "official" Haskell community: https://www.haskell.org/community/
- There's a subreddit, too: https://www.reddit.com/r/haskellquestions/

Haskell Fun (Part II)

Labwork!

In the following file:

`labs/5CM524 Lab 2.docx`

you can find this course's second lab.

Complete any tasks found therein and share your thoughts in class!

Remember, this lab is graded!

Homework

Read the following Wikipedia Lemma about the Sieve of Eratosthenes:

https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

Then, using your soon to be all time favourite programming language (Haskell, if that was unclear), implement the sieve algorithm. That is, you should implement a function:

```
sieve :: Integer -> [Integer]
```

which accepts a single integer, n, and prints all primes up to n.

Any Questions

Don’t forget to fill in the questionnaire shown right!

https://forms.gle/YU1mjxjBjBf8hyvH7