



Functional Programming

:| foo “Haskell”

Markos Vassilis, Mediterranean College

Week 03

Last Time's Tasks

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:

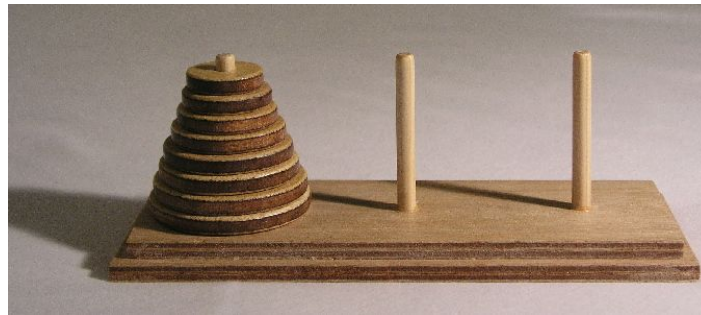
```
isPrimeG :: Int -> Bool
isPrimeG n | n < 2 = False
           | otherwise = length [x | x <- [1..m], rem n x == 0] == 1
           where m = floor (sqrt (fromIntegral n))
```

- 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?

```
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?

Algebraic Data Types

An Enumerate-Like Thing



```
-- source/data_001.hs
main :: IO()
main = do
    print cabbage
    print haskellEssentials

data HaskellStuff = Functions -- Almost all Haskell
                  | Variables -- Okay, placeholders...
                  | Cabbage -- Why not?
                  | Cabal -- of course
                  | Fun -- well...
                  deriving Show -- Just to let Haskell know this is "showable"

cabbage :: HaskellStuff -- Declare this as
cabbage = Cabbage

haskellEssentials :: [HaskellStuff]
haskellEssentials = [Functions, Cabbage, Fun, Fun, Fun]
```

What We Actually Did



In the above script, we defined;

- A new data type, named `HaskellStuff`.
 - All data types should start with an uppercase letter;
 - (Reminder) All variables and functions start with lowercase letters.
- We defined the several possible values this data type can take one by one, which actually resembles an enumerate-like object.
- We can also define functions that use our new data type!


```
isOdd :: HaskellStuff -> Bool
```

```
isOdd Cabbage = True
```

```
isOdd Fun     = True
```

```
isOdd _       = False
```

More Complex ADTs



```
main :: IO()
main = do
    print aFailure
    print aSuccess
    print (safeDiv 5 0 )
    print (safeDiv 5 9)

-- An ADT that represents a double that may be OK or "failed"
data FailableDouble = Failure
                    | OK Double
    deriving Show

aFailure = Failure
aSuccess = OK 5.42

-- Safe division, in the sense of properly handling division by zero
safeDiv :: Double -> Double -> FailableDouble
safeDiv _ 0 = Failure
safeDiv x y = OK (x / y)
```


ADTs (In General)



The general syntax for the definition of an ADT is:

```
data AlgDataType = Constr1 Type11 Type12
                  | Constr2 Type21
                  | Constr3 Type31 Type32 Type33
                  | Constr4
```

So, we can define multiple constructors for an ADT using guards and each one of them can accept any number of arguments (0 or more).

ADTs And Pattern Matching



We can pattern match ADTs with multiple constructors by matching their constructors + their arguments:

```
foo (Constr1 a b)    = ...  
foo (Constr2 a)      = ...  
foo (Constr3 a b c)  = ...  
foo Constr4          = ...
```

Recursive Data Types



We can define a data type recursively, more or less as we do with functions:

```
data IntList = Empty | Cons Int IntList
```

The above defines an Integer list roughly the way Haskell does in the background:

- The base constructor just provides an empty list (so, nothing, in practice).
- The recursive constructor constructs a list of integers using an integer (the head) and a list of integers (the tail).

Recursive Data Types



We can now work with our recursive list much like we would do with built-in ones, just using our own cumbersome syntax:

```
main :: IO()
main = do
    print iList
    print (intListProd iList)

data IntList = Empty | Cons Int IntList
    deriving Show

iList :: IntList
iList = Cons 5 (Cons 4 (Cons 3 (Cons 2 Empty)))
-- The above is equivalent to 5:4:3:2:[]
-- You can read `Cons h t` as: "Construct a list by appending h to t",
-- where h is an integer and t is a list of integers.

intListProd :: IntList -> Int
intListProd Empty      = 1
intListProd (Cons x l) = x * intListProd l
```

Fun Time!

Parsing Logs



For today's labs:

- Visit labs and open: 02-ADTs.pdf.
- Follow the instructions therein.
- All relevant materials are also available in the labs directory.
- Source: <https://www.cis.upenn.edu/~cis1940/spring13/lectures.html> → Week 2

Any Questions



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



<https://forms.gle/YU1mjxjBjBf8hyvH7>