Introduction to Functional Programming in Haskell

MAC x MAPS

x

Lauren Yim

16 Oct 2023

Setup

Installing locally

- Install GHCup: `haskell.org/ghcup`
- `ghcup tui`, install (`i`) `stack`, set (`s`) it

```
✔✔  Stack 2.11.1        latest
✗   Stack 2.9.3         recommended
✗   Stack 2.9.1
```

- Clone `github.com/cherryblossom000/mac-x-maps-haskell-workshop`
- Run `stack test` (there will be test failures)

Using Replit

- Fork `replit.com/@cherryblossom00/MAC-x-MAPS-Haskell-Workshop`
- Run `stack test` in a ‘Shell’ tab (there will be test failures)

What is Haskell?

- **General-purpose** language
- Enforces **pure functional programming**
- **Statically typed**: no more `TypeErrors`
- **Lazy**: only computes values when needed

What is functional programming?

- Declarative programming paradigm based on applying and composing functions
- Usually synonymous with ‘pure functional programming’, which also tries to minimise side effects
- Code can be a lot shorter and easier to read/understand

What is functional programming?

Imperative

```
vector<int> numbers{2, 4, 3, 1, 6, 10, 5};
vector<int> result;
for (int i = 0; i < numbers.size(); i++) {
  int x = numbers[i] * 3;
  if (x % 2 == 0) {
    result.push_back(x);
  }
}

int sumOfResult = 0;
for (int i = 0; i < result.size(); i++) {
  sumOfResult += result[i];
}
```

Functional

```
numbers :: [Int]
numbers = [2, 4, 3, 1, 6, 10, 5]

result :: [Int]
result = filter even (map (*3) numbers)

sumOfResult :: Int
sumOfResult = sum result
```

What is functional programming?

Imperative

```
int result = 0;
int count = 0;
int a = 1;
int b = 1;
while (count < 10) {
  if (a % 2 != 0) {
    result += a;
    count++;
  }
  int tmp = a + b;
  a = b;
  b = tmp;
}
```

Functional

```haskell
fibs :: [Int]
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)

result :: Int
result = sum (take 10 (filter odd fibs))
```

What is functional programming?

Imperative

```python
def partition(array, low, high):
  pivot = low
  for i in range(low + 1, high + 1):
    if array[i] <= array[low]:
      pivot += 1
      swap(array, i, pivot)
    swap(array, low, pivot)
  return pivot
def quicksort_aux(array, low, high):
  if low >= high: return
  pivot = partition(array, low, high)
  quicksort_aux(array, low, pivot - 1)
  quicksort_aux(array, pivot + 1, high)
def quicksort(array):
  quicksort_aux(array, 0, len(array) - 1)
```

Functional

```haskell
import Data.List (partition)

quicksort :: Ord a => [a] -> [a]
quicksort [] = []
quicksort (x:xs) = lt ++ [x] ++ gt
  where (lt, gt) = partition (<x) xs
```

Hello, World!

```haskell
-- Every Haskell program needs a 'main' defined
main :: IO ()
--   ^^^^^^^^ type annotation (IO indicates a side effect)
main = putStrLn "Hello, world!"
--     ^^^^^^^^ prints a string
-- no need for () to call a function
```

Run the program with `stack run`

Hello, World!

```haskell
-- Every Haskell program needs a 'main' defined
main :: IO ()
--   ^^^^^^^^ type annotation (IO indicates a side effect)
main = putStrLn "Hello, world!"
--     ^^^^^^^^ prints a string
-- no need for () to call a function
```

Run the program with `stack run`

~~a monad is just a monoid in the category of endofunctors~~

Haskell Syntax

- Parentheses are used for grouping expressions together, not for calling functions
  - Python: `function1(arg1, function2(arg2), arg3)`
  - Haskell: `function1 arg1 (function2 arg2) arg3`
- Everything is an expression
- Run `stack ghci` to play around in a REPL

Haskell Syntax

```haskell
-- Comments start with two hyphens
{- Multiline comments are
like this -}

integer :: Int
integer = 40 + 2

float :: Double
float = 3.141592
```

```haskell
string :: String
string =
  let
    noun = "Haskell"
    adjective = "awesome"
  in noun ++ " is " ++ adjective

list :: [Int]
list = [x, y, 3]
  where
    x = 1
    y = 2
```

Haskell Syntax

```haskell
double :: Int -> Int
double x = x * 2

add :: Int -> Int -> Int
add x y = x + y

listDescription :: [a] -> String
listDescription [] = "list is empty"
listDescription _  = "list has stuff"
```

```haskell
if1 :: Int -> Int
if1 n = if n == 10 then 0 else n + 1

if2 :: Int -> Int
if2 n
  | n == 10   = 1
  | otherwise = n + 1

if3 :: Int -> Int
if3 10 = 0
if3 n  = n + 1
```

Side Effects and Purity

- Side effects are any changes to state outside of the function. For example:
    - Modifying global variables
    - Modifying an array that was passed into the function
    - Printing to console
    - Making network requests
- Pure functions have no side effects and always return the same output for the same input

Side Effects and Purity

- Pure functions are a lot easier to reason about due to **referential transparency**
  - This is the property that you can replace a function call with its output without changing the behaviour of the program
  - More generally, you can replace any expression with another expression that evaluates to the same value

Side Effects and Purity

```
def add_one(x: int) -> int:
  print(x)
  return x + 1

import time
def get_time() -> float:
  return time.time()
```

```
def double(xs: list[int]) ->
list[int]:
  for i, x in enumerate(xs):
    xs[i] = x * 2
  return xs

counter = 0
def inc_counter() -> None:
  counter += 1
```

Tasks

- `src/Part1.hs`
- Replace all `error "blah blah"`s with your own code
- Run `stack test --test-arguments 1` to test
- Run `stack ghci` for a REPL to help debug if needed
- Tip: Hoogle (`hoogle.haskell.org`) can be used to search for functions by their name or even type (e.g. try searching for `Int -> a -> [a]`)

Higher-Order Functions

- In many languages such as Haskell, Python, and JavaScript, functions are first-class, meaning that they are can be treated like any other value
  - You can assign a function to a variable, use a function as an argument to another function, return a function from a function...
- Higher-order functions are functions that either accept a function as a parameter or return a function

Higher-Order Functions

- Functions can be composed together with `.` (like $\circ$ in maths)
  - $(f \circ g)(x) = f(g(x))$
  - `(f . g) x = f (g x)`
  - e.g. `addOneThenDouble = (* 2) . (+ 1)`
- All functions in Haskell are **curried**
  - A function that takes e.g. 2 parameters actually takes in a single parameter and returns a new function that takes the second parameter
  - Helpful for reusing functions

Higher-Order Functions

```haskell
add :: Int -> Int -> Int
add x y = x + y

five :: Int
five = add 2 3

addOne :: Int -> Int
addOne = add 1

three :: Int
three = addOne 2
```

```python
def add(x: int) -> Callable[[int], int]:
  return lambda y: x + y

five = add(2)(3)

add_one = add(1)

three = add_one(2)
```

Higher-Order Functions

```haskell
-- Applies a function to all the elements of a list and
-- returns the results in a new list
map :: (a -> b) -> [a] -> [b]
-- Only keeps elements in the list satisfying a predicate
filter :: (a -> Bool) -> [a] -> [a]
-- Applies a function to each element of the list (from right to left)
-- and an accumulator value and returns the final accumulator value
foldr :: (a -> b -> b) -> b -> [a] -> b
-- Same as foldr but left to right
foldl :: (b -> a -> b) -> b -> [a] -> b
-- Anonymous functions in Haskell: \arg1 arg2 -> arg1 + arg2
map    (* 2)                   [1, 2, 3, 4, 5, 6] -- [2, 4, 6, 8, 10, 12]
filter (\x -> x `mod` 3 == 0) [1, 2, 3, 4, 5, 6] -- [3, 6]
```

Higher-Order Functions

x	acc
	0
5	0 + 5 = 5
4	5 + 4 = 9
3	9 + 3 = 12
2	12 + 2 = 14
1	14 + 1 = 15

```haskell
foldr (\x acc -> x + acc) 0 [1..5] -- 15
-- could also be written as
foldr (+) 0 [1..5]
-- or
sum [1..5]
```

Lists

- Lists in Haskell are linked lists

```
data [a] = [] | a : [a]
-- the : is an operator that can be used to
-- create a list given the head and tail

-- you can think of it like this:
data List a = Empty | Cons a (List a)
```

- `[1, 2, 3]` is just syntax sugar for `1 : 2 : 3 : []`
- `[1..4] == [1, 2, 3, 4]`
- `[1, 3..9] == [1, 3, 5, 7, 9]`
- `[1..]` is an infinite list `[1, 2, 3, ...]`

Tasks

- `src/Part2.hs`
- Replace all `error "blah blah"`s with your own code
- Run `stack test --test-arguments 2` to test
- Run `stack ghci` for a REPL to help debug if needed
- Tip: Hoogle (`hoogle.haskell.org`) can be used to search for functions by their name or even type (e.g. try searching for `Int -> a -> [a]`)

Data Types

```haskell
data PairOfInts = PairOfInts Int Int
--   ^^^^^^^^^^   ^^^^^^^^^^
--   type         constructor
x :: PairOfInts
x = PairOfInts 1 2

data Pair a = Pair a a
-- 'a' is a type variable
y :: Pair String
y = Pair "abc" "def"
```

```haskell
getFirst :: Pair a -> a
getFirst (Pair x _) = _

getSecond :: Pair a -> a
getSecond pair = case pair of
  (Pair _ y) -> y
```

Data Types

```haskell
data Person = Person
  { name :: String
  , age :: Int
  }

person1 :: Person
person1 = Person "Lauren" 19

person2 :: Person
person2 = Person
  { age = 19
  , name = "Lauren"
  }
```

```haskell
getName1, getName2, getName3 :: Person -> String
getName1 (Person n _) = n
getName2 Person{name = n} = n
getName3 person = name person
```

Data Types

```haskell
data Suit = Hearts
          | Diamonds
          | Clubs
          | Spades

suit :: Suit
suit = Hearts

message :: String
message = case suit of
  Hearts -> "it's hearts!"
  _      -> "something else"
```

```haskell
displaySuit :: Suit -> String
displaySuit Hearts   = "♥"
displaySuit Diamonds = "♦"
displaySuit Clubs    = "♣"
displaySuit Spades   = "♠"
```

`Maybe`

```haskell
data Maybe a = Nothing | Just a
```

- Used instead of `null`/`nil`/`None`
- Example: getting a value for a key in a dictionary/hash map

  ```haskell
  lookup :: k -> Map k v -> Maybe v
  ```

- Nice way to handle functions that may fail rather than having to catch exceptions

Non-Empty Lists

```
data NonEmpty a = a :| [a]
```

- List that has at least one element

```
nonEmpty :: [a] -> Maybe (NonEmpty a)
head :: NonEmpty a -> a   -- first
last :: NonEmpty a -> a   -- last
tail :: NonEmpty a -> [a] -- all except first
init :: NonEmpty a -> [a] -- all except last
```

Recursive Data Types

```haskell
data Expression
  = Integer Int
  | Add Expression Expression
  | Subtract Expression Expression
  | Multiply Expression Expression
  | Power Expression Expression
```

Tasks

- `src/Part3.hs`
- Run `stack run` to run the calculator
- Replace all `error "blah blah"`s with your own code
- Run `stack test --test-arguments 3` to test
- Run `stack ghci` for a REPL to help debug if needed
- Tip: Hoogle (`hoogle.haskell.org`) can be used to search for functions by their name or even type (e.g. try searching for `Int -> a -> [a]`)

Learning More + Resources

- Haskell Wiki: `wiki.haskell.org`
- Hoogle (Search engine for Haskell functions): `hoogle.haskell.org`
- Learn You a Haskell for Great Good (tutorial): `learnyouahaskell.github.io`

If you liked this, consider taking FIT2102 Programming paradigms!