COMP105 Lecture 13

Higher Order Functions

Class Test 1

There will be an in-person class test **next week**

- Not in our usual room
- Check your timetable

Covers Lectures 1 – 10

- What is a pure function?
- Haskell basics
- Recursion

Class Test: Procedure

Format

- Multiple choice
- 20 questions
- 35 minutes
- Answers filled in on a computer-readable sheet
- You will need to bring an HB pencil

The test will start **promptly** and last for 35 minutes

- Late comers will get less time!
- Leaving early is not permitted

Class Test

A **practice class test** is available

- On the assessments page on Canvas
- Same format as the class test

We will go through the solutions in a **revision lecture**

- The lecture before the class test

Outline

Today

- A few more type classes
- Higher order functions
- Function composition
- Anonymous functions

Relevant book chapters

- Programming In Haskell Chapters 4 and 7
- Learn You a Haskell Chapter 6

Converting to strings

The **show** function converts other types to strings

```
ghci> show 123
"123"

ghci> show [1,2,3]
"[1,2,3]"

ghci> show (True, 2.5)
"(True,2.5)"
```

Converting to strings

The **Show** type class contains types that can be shown

```
ghci> :t show
show :: Show a => a -> String
```

Show contains

- all basic types
- all tuples containing showable types
- all lists that contain showable types

Converting from strings

Read converts strings to other types

```
ghci> read "123" :: Int
123

ghci> read "False" :: Bool
False

ghci> read "[1,2,3,4]" :: [Int]
[1,2,3,4]
```

The use of `::` is necessary to tell Haskell what type it is parsing

Converting from strings

It is not necessary to use `::` when Haskell can deduce the type from the context

```
ghci> not (read "False")
True

ghci> :t not
not :: Bool -> Bool
```

```
ghci> read "4" * read "6"
24
```

Converting from strings

The **Read** type class contains all types that can be read

```
ghci> :t read
read :: Read a => String -> a
```

As with show, it contains

- all basic types
- all tuples containing readable types
- all lists that contain readable types

Ordered types

The type class **Ord** contains all types that can be compared

```
ghci> :t (>)
(>) :: Ord a => a -> a -> Bool

ghci> :t (<=)
(<=) :: Ord a => a -> a -> Bool

ghci> :t max
max :: Ord a => a -> a -> a
```

Ordered types

It contains numbers, but also **all** basic types, tuples, and lists

```
ghci> 'a' < 'b'
True

ghci> True > False
True

ghci> (1, 10) <= (1, 11)
True

ghci> [1..10] < [2..11]
True
```

Tuples and lists are compared **lexicographically** (element by element)

Higher order functions

A **higher order function** is a function that

- Takes another function as an argument, or
- Returns a function

```
apply_twice :: (a -> a) -> a -> a
apply_twice f input = f (f input)
```

```
ghci> apply_twice tail [1,2,3,4]
[3,4]
```

Apply_twice examples

```haskell
apply_twice :: (a -> a) -> a -> a
apply_twice f input = f (f input)
```

```
ghci> apply_twice ((+) 2) 2
6

ghci> apply_twice (drop 2) [1,2,3,4,5]
[5]

ghci> apply_twice reverse [1,2,3,4]
[1,2,3,4]
```

The apply_twice type

```haskell
apply_twice :: (a -> a) -> a -> a
apply_twice f input = f (f input)
```

The type specifies that

- `f :: (a -> a)`
- `input :: a`
- The function returns type `a`

So the following will give a type **error**

```
ghci> apply_twice head [[1,2], [3,4]]
```

Function composition

Function **composition** applies one function to the output of another

- Composing `f` with `g input` gives `f (g input)`

```haskell
compose :: (b -> c) -> (a -> b) -> a -> c
compose f g input = f (g input)
```

```
ghci> compose (+1) (*2) 4
9

ghci> compose head head [[1,2], [3,4]]
1
```

The . operator

In Haskell `compose` is implemented by the `.` operator

```
ghci> compose head head [[1,2], [3,4]]
1

ghci> (head . head) [[1,2], [3,4]]
1
```

```
ghci> :t (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
```

The . operator

The . operator is particularly useful when composing a **long list** of functions

```
f list = length (double (drop_evens (tail list)))
```

```
f' list = (length . double . drop_evens . tail) list
```

The use of . removes the need for nested brackets

- but it is stylistic
- you never need to use .

The $ operator

```haskell
evaluate :: (a -> b) -> a -> b
evaluate f input = f input
```

This function just **evaluates** its input

```
ghci> evaluate length [1,2,3]
3
```

The $ operator

The $ operator is exactly the same as **evaluate**

```
ghci> ($) length [1,2,3]
3

ghci> length $ [1,2,3]
3

ghci> :t ($)
($) :: (a -> b) -> a -> b
```

The $ operator

The $ operator has the lowest **precedence** of all operators

- It is mainly used to avoid brackets

```
ghci> length ([1,2,3] ++ [4,5,6])
6

ghci> length $ [1,2,3] ++ [4,5,6]
6


ghci> (length . tail) [1,2,3,4]
3

ghci> length . tail $ [1,2,3,4]
3
```

Exercise

What do these functions do?

```haskell
mystery :: (a -> a) -> a -> a
mystery f input = (f . f . f) input
```

```haskell
mystery2 :: (a -> Int) -> (a -> Int) -> a -> Int
mystery2 f g input = f input + g input
```

```haskell
mystery3 :: (a -> b -> c) -> a -> b -> c
mystery3 f in1 in2 = in1 `f` in2
```

Anonymous functions

Sometimes it is convenient to define a function **inline**

```
ghci> (\x -> x + 1) 2
3
```

```
ghci> :t (\x -> x+1)
(\x -> x+1) :: Num a => a -> a
```

```
ghci> apply_twice (\x -> 2 * x) 2
8
```

These are called **anonymous** functions: they have no name

Anonymous functions syntax

The **syntax** for an anonymous function is:

```
\ [arg1] [arg2] ...  -> [expression]
```

The \ is supposed to resemble a lambda (λ)

- Anonymous functions are sometimes called λ-functions

Examples:

```haskell
\ x y -> x + y + 1
```

```haskell
\ list -> head list + last list
```

Functions that return functions

Higher order functions can also **return** other functions

```haskell
f_that_adds_n :: Int -> (Int -> Int)
f_that_adds_n n = (\ x -> x + n)
```

```
ghci> let f = (f_that_adds_n 10) in (f 1)
11

ghci> (f_that_adds_n 20) 1
21

ghci> (f_that_adds_n 2 . f_that_adds_n 3) 0
5
```

Functions that take and return functions

Higher order functions can take **and** return functions

```haskell
swap :: (a -> b -> c) -> (b -> a -> c)
swap f = \ x y -> f y x
```

```
ghci> take 4 [1..10]
[1,2,3,4]
```

```
ghci> (swap take) [1..10] 4
[1,2,3,4]
```

Currying revisited

Previously we've seen that it is possible to **partially** apply a function

```
add_two = (+2)
```

```
ghci> add_two 2
4
```

```
drop_six = drop 6
```

```
ghci> drop_six [1..10]
[7,8,9,10]
```

Currying revisited

This is just **nicer syntax** for a function that returns a function

```haskell
add_two = (+2)

add_two' = (\ x -> x + 2)
```

```haskell
drop_six = drop 6

drop_six' = (\ x -> drop 6 x)
```

Exercise

What do these queries return?

```
ghci> (\ x -> take 4 x) [1..10]

ghci> (\ f -> f [1,2,3,4]) length

ghci> drop 2 . drop 2 . drop 2 $ [1..10]
```

Exercises

1. Rewrite the following functions using the `.` operator

   1.1 `sum_tail list = sum (tail list)`

   1.2 `third_head list = head (tail (tail list))`

   1.3 `length_middle list = length (tail (init list))`

2. Write anonymous functions that implement the following functions. You can use

   ```
   let f = (your function) in (f arguments)
   ```

   in ghci to test your functions.

   2.1 A function with one argument `x` that returns `2*x + 1`

   2.2 A function with two arguments `x` and `y` that returns x^y

   2.3 A function with two arguments `list` and `n` that returns the `(2*n)`th element of `list`.

Exercises

3. Use `apply_twice` to create a function `second_tail` that returns all but the first two elements of a list.

4. Write a function `without_last_4` that takes a list, and returns that list without the last four elements (Hint: use `reverse` and `drop`)

Exercises

5. Write a function `rotate_args` that takes a function `f` with three arguments `x`, `y`, and `z`, and returns a new function that behaves like `f z x y`

6. (*) Write a function `apply_n :: (a -> a) -> Int -> (a -> a)` that takes a function `f` and an integer `n`, and returns `f` composed with itself `n` times

Summary

- A few more type classes
- Higher order functions
- Function composition
- Anonymous functions

Next time: Map