COMP3141

Software System Design and Implementation

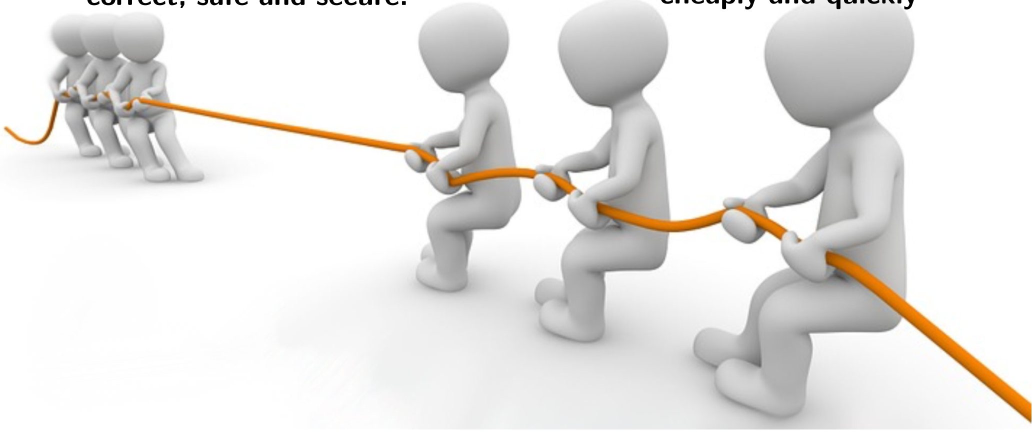
Functional Programming Practice

Curtis Millar
CSE, UNSW
Term 2 2021

Recap: What is this course?

Software must be high quality: **correct, safe and secure.**

Software must developed **cheaply and quickly**

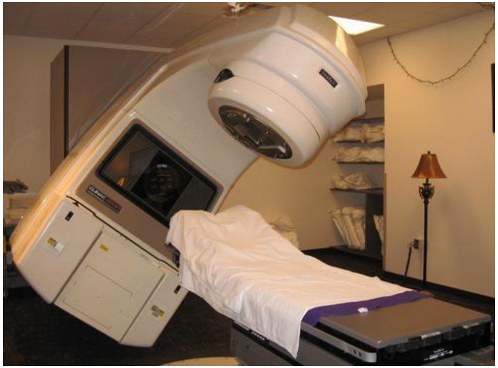


Recall: Safety-critical Applications

For safety-critical applications, failure is not an option:

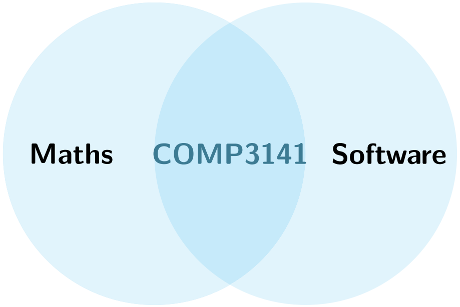
- planes, self-driving cars
- rockets, Mars probe
- drones, nuclear missiles
- banks, hedge funds, cryptocurrency exchanges
- radiation therapy machines, artificial cardiac pacemakers

Safety-critical Applications



A bug in the code controlling the Therac-25 radiation therapy machine was directly responsible for at least five patient deaths in the 1980s when it administered excessive quantities of beta radiation.

COMP3141: Functional Programming



Functional Programming: How does it Help?

1. Close to Maths: more abstract, less error-prone
2. Types: act as doc., the compiler eliminates many errors
3. Property-Based Testing: QuickCheck (in Week 3)
4. Verification: equational reasoning eases proofs (in Week 4)

COMP3141: Learning Outcomes

1. Identify basic Haskell **type errors** involving concrete types.
2. Work comfortably with **GHCi** on your working machine.
3. Use Haskell **syntax** such as guards, **let**-bindings, **where** blocks, **if** etc.
4. Understand the **precedence of function application** in Haskell, the `(.)` and `($)` operators.
5. Write Haskell programs to manipulate **lists** with recursion.
6. Makes use of **higher order functions** like *map* and *fold*.
7. Use λ-**abstraction** to define anonymous functions.
8. Write Haskell programs to compute **basic arithmetic**, **character, and string manipulation**.
9. Decompose problems using **bottom-up design.**

Functional Programming: History in Academia

1930s	Alonzo Church developed lambda calculus (equiv. to Turing Machines)
1950s	John McCarthy developed Lisp (LISt Processor, first FP language)
1960s	Peter Landin developed ISWIM (If you See What I Mean, first pure FP language)
1970s	John Backus developed FP (Functional Programming, higher-order functions, reasoning)
1970s	Robin Milner and others developed ML (Meta-Language, first modern FP language, polymorphic types, type inference)
1980s	David Turner developed Miranda (lazy, predecessor of Haskell)
1987-	An international PL committee developed Haskell (named after the logician Curry Haskell)

… received Turing Awards (similar to Nobel prize in CS).
Functional programming is now taught at most CS departments.

Functional Programming: Influence In Industry

- Facebook's motto was:
  - "Move fast and break things."
  - as they expanded, they understood the importance of bug-free software
  - now Facebook uses functional programming!
- JaneStreet, Facebook, Google, Microsoft, Intel, Apple
  (... and the list goes on)
- Facebook building React and Reason, Apple pivoting to Swift, Google developing MapReduce.

Closer to Maths: Quicksort Example

Let's solve a problem to get some practice:

Example (Quicksort, recall from Algorithms)

Quicksort is a divide and conquer algorithm.

1. Picks a pivot from the array or list
2. Divides the array or list into two smaller sub-components: the smaller elements and the larger elements.
3. Recursively sorts the sub-components.

- What is the average complexity of Quicksort?
- What is the worst case complexity of Quicksort?
- Imperative programs describe **how** the program works.
- Functional programs describe **what** the program does.

Quicksort Example (Imperative)

```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p - 1)
        quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) is
    pivot := A[hi]
    i := lo
    for j := lo to hi - 1 do
        if A[j] < pivot then
            swap A[i] with A[j]
            i := i + 1
    swap A[i] with A[hi]
    return i
```

Quick Sort Example (Functional)

```haskell
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort smaller ++ [x] ++ qsort larger
               where
                smaller = filter (\ a-> a <= x) xs
                larger  = filter (\ b-> b > x) xs
```

Is that it? Does this work?

Expressions and Types

Haskell constructs programs out of *expressions* but does not have any notion of a *statements* (unlike, e.g., C, Java, Python). Every expression has a type.
Let's practice figuring out the types of some expressions.

1. `True :: Bool`
2. `'a' :: Char`
3. `['a', 'b', 'c'] :: [Char]`
4. `"abc" :: [Char]`
5. `["abc"] :: [[Char]]`
6. `[('f',True), ('e', False)] :: [(Char, Bool)]`

1. In Haskell and GHCi using `:t`.
2. Using Haskell documentation and GHCi, answer the questions in this week's quiz (assessed!).

Functions

A function takes a single value as input and produces a single value as output. Functions themselves are values, so a function can return a function that accepts further arguments.
We can create *anonymous* functions, i.e. functions without a name, using lambdas.
What is they type of the following lambda function?

```
\name -> "Hello, " ++ name ++ "!"
```

```
\name -> "Hello, " ++ name ++ "!" :: [Char] -> [Char]
```

Polymorphism

A function can have a *polymorphic* type, one that can take many forms. Such functions have *type variables* in the type that can be instantited with any type.
The most simply polymorphic function is the identity function that takes in input value and returns that same value with the same type.

```
id = \x -> x :: forall a. a -> a
-- Alternatively
id :: a -> a
id x = x
```

For any type a, the function takes a value of that type and returns a value of the same type.

Polymorphism

Another surprisingly useful polymorphic function is the *constant* function that takes an input value and returns another function that, regardless of its input, always returns the value of the first argument. What is its type?

```
const = \x -> \y -> x
```

```
const :: a -> b -> a
const x y = x
```

Case and pattern matching

Some types have multiple *constructors*, such as `Char` (`'a'`, `'b'`, ...), `Int` (..., `-1`, `0`, `1`, ...), and `[a]` (`[]` and `x : xs`).
To access the values within a given datatype, we can *pattern match* a value of that type and *bind* parts of the constructor to names within a given expression.

```
howLong :: [a] -> String
howLong = \xs -> case xs of
        [] -> "empty"
        _ : [] -> "a single element"
        _ : xs -> "longer than a list that is " ++ (howLong xs)
```

Functions and pattern matching

```haskell
howLong :: [a] -> String
howLong = \xs -> case xs of
        []      -> "empty"
        _ : []  -> "a single element"
        _ : xs  -> "longer than a list that is " ++ (howLong xs)
```

Can also be written as:

```haskell
howLong :: [a] -> String
howLong []         = "empty"
howLong (_ : [])   = "a single element"
howLong (_ : xs)   = "longer than a list that is " ++ (howLong xs)
```

Higher Order Functions

As functions are also values, we can also accept functions as input values. How could we implement a function with the following type?

```haskell
flip :: (a -> b -> c) -> (b -> a -> c)
```

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

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

Right Fold

Let's try and devise a more complex function for some practice.

Example (Demo Task)

We want a function that can consume an entire list and use each value in the list to construct a new value.

- This function should be able to take any operation and starting value.
- When this function is passed the operation `:` and the starting value `[]`, it should return the input list

```haskell
-- What is the type?
listFold = unimplemented

testListFold xs = listFold (:) [] xs == xs
```

Right Fold

Next we'll implement each of the following directly then using our fold function:

- `map`
- `reverse`
- `leftFold`
- `filter`
- `append`
- `concat`

Word Frequencies

Let's solve a problem to get some practice:

Example (First Demo Task)

Given a number n and a string s, generate a report (in `String` form) that lists the n most common words in the string s.

We must:

1. Break the input string into words.
2. Convert the words to lowercase.
3. Sort the words.
4. Count adjacent runs of the same word.
5. Sort by size of the run.
6. Take the first n runs in the sorted list.
7. Generate a report.

Numbers into Words

Let's solve a problem to get some practice:

Example (Demo Task)

Given a number n, such that $0 \leq n < 1000000$, generate words (in `String` form) that describes the number n.

We must:

1. Convert single-digit numbers into words ($0 \leq n < 10$).
2. Convert double-digit numbers into words ($0 \leq n < 100$).
3. Convert triple-digit numbers into words ($0 \leq n < 1000$).
4. Convert hexa-digit numbers into words ($0 \leq n < 1000000$).

Single Digit Numbers into Words

$$0 \leq n < 10$$

```haskell
units :: [String]
units =
    ["zero", "one", "two", "three", "four", "five",
     "six", "seven", "eight", "nine", "ten"]
convert1 :: Int -> String
convert1 n = units !! n
```

Double Digit Numbers into Words

$$0 \leq n < 100$$

```
teens :: [String]
teens =
    ["ten", "eleven", "twelve", "thirteen", "fourteen",
     "fifteen", "sixteen", "seventeen", "eighteen",
     "nineteen"]

tens :: [String]
tens =
    ["twenty", "thirty", "fourty", "fifty", "sixty",
     "seventy", "eighty", "ninety"]
```

Double Digit Numbers into Words Continued ($0 \leq n < 100$)

```haskell
digits2 :: Int -> (Int, Int)
digits2 n = (div n 10, mod n 10)
combine2 :: (Int, Int) -> String
combine2 (t, u)
    | t == 0             = convert1 u
    | t == 1             = teens !! u
    | t > 1 && u == 0    = tens !! (t-2)
    | t > 1 && u /= 0    = tens !! (t-2)
                           ++ "-" ++ convert1 u
convert2 :: Int -> String
convert2 = combine2 . digits2
```

Infix Notation

Instead of

```haskell
digits2 n = (div n 10, mod n 10)
```

for **infix** notation, write:

```haskell
digits2 n = (n `div` 10, n `mod` 10)
```

Note: this is not the same as single quote used for `Char` (`'a'`).

Simpler Guards but Order Matters

You could also simplify the guards as follows:

```
combine2 :: (Int, Int) -> String
combine2 (t,u)
   | t == 0      = convert1 u
   | t == 1      = teens !! u
   | u == 0      = tens !! (t-2)
   | otherwise   = tens !! (t-2) ++ "-" ++ convert1 u
```

but now the order in which we write the equations is crucial. `otherwise` is a synonym for `True`.

Where instead of Function Composition

Instead of implementing `convert2` as `digit2.combine2`, we can implement it directly using the `where` keyword:

```haskell
convert2 :: Int -> String
convert2 n
  | t == 0       = convert1 u
  | t == 1       = teens !! u
  | u == 0       = tens !! (t-2)
  | otherwise    = tens !! (t-2) ++ "-" ++ convert1 u
  where (t, u) = (n `div` 10, n `mod` 10)
```

Triple Digit Numbers into Words
$(0 \leq n < 1000)$

```
convert3 :: Int -> String
convert3 n
    |  h == 0       = convert2 n
    |  t == 0       = convert1 h ++ "hundred"
    |  otherwise    = convert1 h ++ " hundred and "
                      ++ convert2 t
    where (h, t) = (n `div` 100, n `mod` 100)
```

Hexa Digit Numbers into Words $(0 \leq n < 1000000)$

```
convert6 :: Int -> String
convert6 n
    |  m == 0       = convert3 n
    |  h == 0       = convert3 m ++ "thousand"
    |  otherwise    = convert3 m ++ link h ++ convert3 h
    where (m, h) = (n `div` 1000, n `mod` 1000)
link :: Int -> String
link h = if (h<100) then " and " else " "
convert :: Int -> String
convert = convert6
```

Homework

1. Get Haskell working on your development environment. Instructions are on the course website.
2. Using Haskell documentation and GHCi, answer the questions in this week's quiz (assessed!).