COMP3141

Software System Design and Implementation

Functional Programming Practice

Christine Rizkallah
CSE, UNSW (and Data61)
Term 2 2019

Recap: What is this course?

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

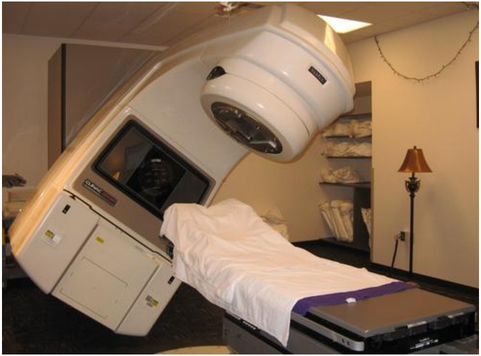
Software must developed
cheaply and quickly

Recall: Safety-critical Applications

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

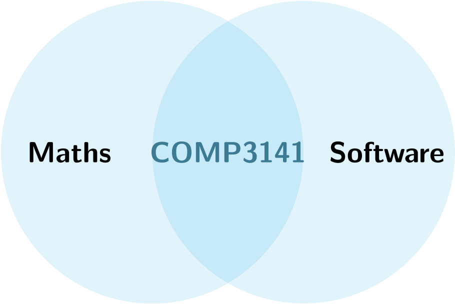
- planes, self-driving cars
- rockets, Mars probe
- drones, nuclear missiles
- banks, hedge funds, cryptocurrency exchanges
- radiation therapy machines, artificial cardiac pacemakers

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.

COMP3141: Learning Outcomes

1. Identify basic Haskell **type errors** involving concrete types.
2. Work comfortably with **GHCi** on your working machine.

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.

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.

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.

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*.

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.

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**.

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)

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: 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.

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?

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?

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
```

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?

Practice Types

In the previous lecture, you learned about the importance of types in functional programming. Let's practice figuring out the types of terms.

1. `True`

Practice Types

In the previous lecture, you learned about the importance of types in functional programming. Let's practice figuring out the types of terms.

1. `True :: Bool`

Practice Types

In the previous lecture, you learned about the importance of types in functional programming. Let's practice figuring out the types of terms.

1. `True :: Bool`
2. `'a'`

Practice Types

In the previous lecture, you learned about the importance of types in functional programming. Let's practice figuring out the types of terms.

1. `True :: Bool`
2. `'a' :: Char`

Practice Types

In the previous lecture, you learned about the importance of types in functional programming. Let's practice figuring out the types of terms.

1. `True :: Bool`
2. `'a' :: Char`
3. `['a', 'b', 'c']`

Practice Types

In the previous lecture, you learned about the importance of types in functional programming. Let's practice figuring out the types of terms.

1. `True :: Bool`
2. `'a' :: Char`
3. `['a', 'b', 'c'] :: [Char]`

Practice Types

In the previous lecture, you learned about the importance of types in functional programming. Let's practice figuring out the types of terms.

1. `True :: Bool`
2. `'a' :: Char`
3. `['a', 'b', 'c'] :: [Char]`
4. `"abc"`

Practice Types

In the previous lecture, you learned about the importance of types in functional programming. Let's practice figuring out the types of terms.

1. `True :: Bool`
2. `'a' :: Char`
3. `['a', 'b', 'c'] :: [Char]`
4. `"abc" :: [Char]`

Practice Types

In the previous lecture, you learned about the importance of types in functional programming. Let's practice figuring out the types of terms.

1. `True :: Bool`
2. `'a' :: Char`
3. `['a', 'b', 'c'] :: [Char]`
4. `"abc" :: [Char]`
5. `["abc"]`

Practice Types

In the previous lecture, you learned about the importance of types in functional programming. Let's practice figuring out the types of terms.

1. `True :: Bool`
2. `'a' :: Char`
3. `['a', 'b', 'c'] :: [Char]`
4. `"abc" :: [Char]`
5. `["abc"] :: [[Char]]`

Practice Types

In the previous lecture, you learned about the importance of types in functional programming. Let's practice figuring out the types of terms.

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

Practice Types

In the previous lecture, you learned about the importance of types in functional programming. Let's practice figuring out the types of terms.

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

Practice Types

In the previous lecture, you learned about the importance of types in functional programming. Let's practice figuring out the types of terms.

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

Practice Types

In the previous lecture, you learned about the importance of types in functional programming. Let's practice figuring out the types of terms.

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

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

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.**

Recall: Higher Order List Functions

The rest of last lecture was spent introducing various list functions that are built into Haskell's standard library by way of live coding.

Functions covered:

1. `map`
2. `filter`
3. `concat`
4. `sum`
5. `foldr`
6. `foldl`

In the process, you saw **guards** and **if**, and the `.` operator.

Higher Order List Functions

The rest of last lecture was spent introducing various list functions that are built into Haskell's standard library by way of live coding.

Functions covered:

1. `map`
2. `filter`
3. `concat`
4. `sum`
5. `foldr`
6. `foldl`

In the process, you saw **guards** and **if**, and the `.` operator.

Higher Order List Functions

The rest of last lecture was spent introducing various list functions that are built into Haskell's standard library by way of live coding.

Functions covered:

1. `map`
2. `filter`
3. `concat`
4. `sum`
5. `foldr`
6. `foldl`

In the process, you saw **guards** and **if**, and the `.` operator.

Let's do that again in Haskell.

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.**

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.

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

$$0 \leq n < 10$$

```
units :: [String]
units =
    ["zero", "one", "two", "three", "four", "five",
     "six", "seven", "eight", "nine", "ten"]
```

Single Digit Numbers into Words

$$0 \leq n < 10$$

```
units :: [String]
units =
    ["zero", "one", "two", "three", "four", "five",
     "six", "seven", "eight", "nine", "ten"]

convert1 :: Int -> String
```

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

$$0 \leq n < 100$$

```haskell
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)
```

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
```

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
```

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

```
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
```

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

```
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)
```

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
```

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
```

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

```
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

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

for **infix** notation, write:

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

Infix Notation

Instead of

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

for **infix** notation, write:

```
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:

```
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
```

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

```
convert3 :: Int -> String

convert3 n
    |  h == 0      = convert2 n
```

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

```
convert3 :: Int -> String

convert3 n
    |  h == 0       = convert2 n

    |  t == 0       = convert1 h ++ "hundred"
```

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

```haskell
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
```

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

```haskell
convert6 :: Int -> String

convert6 n
    | m == 0       = convert3 n
```

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

```
convert6 :: Int -> String

convert6 n
    | m == 0      = convert3 n
    | h == 0      = convert3 m ++ "thousand"
```

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
```

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.**

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!).