

COMP3141

Software System Design and Implementation

Introduction

Dr. Christine Rizkallah
UNSW
Term 2 2021

Who are we?

I am **Dr. Christine Rizkallah**, currently a lecturer at UNSW. I lead the Cogent team and my research specialty involves formal methods and programming languages for building reliable software.

Curtis Millar, is a UNSW academic who works on, among other things, projects involving trustworthy systems and formal methods.

Credits

We thank the former lecturers of this course for developing and continuously refining this course material. The course material offered greatly benefits from their contributions.

Dr. Liam O'Connor, who now a lecturer at the University of Edinburgh, and whose research involves programming languages for trustworthy systems.

Prof. Gabriele Keller, who now works at Utrecht University, and whose research expertise is on programming languages for formal methods and high performance computing.

Prof. Manuel Charkavarty, who now works at IOHK with a focus on research and development of programming languages and compilers for functional programming.

Contacting Us

`http://www.cse.unsw.edu.au/~cs3141`

Forum

There is a **Discourse** forum available on the website. Questions about course content should typically be made there. You can ask us private questions to avoid spoiling solutions to other students.

Administrative questions should be sent to `cs3141@cse.unsw.edu.au`.

What is this course?

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

Software must developed
cheaply and quickly



Safety-uncritical Applications



Video games: Some bugs are acceptable, to save developer effort.

Safety-critical Applications

Remember a particularly painful uni group work assignment.

Now imagine you. . .

- Are travelling on a plane
- Are travelling in a self-driving car
- Are working on a Mars probe
- Have invested in a new hedge fund
- Are running a cryptocurrency exchange
- Are getting treatment from a radiation therapy machine
- Intend to launch some nuclear missiles at your enemies

. . . running on software written by **the person next to you on the participant list.**

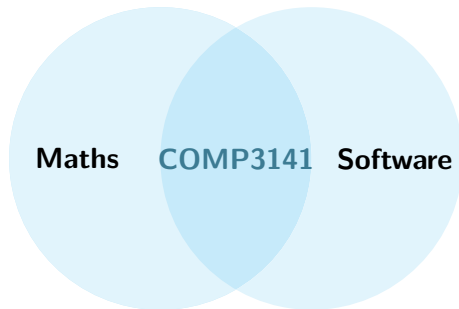
Safety-critical Applications

Airline Blames Bad Software in San Francisco Crash

The New York Times



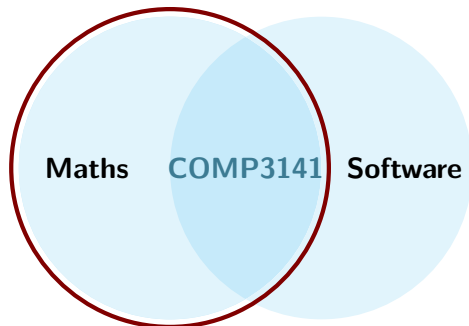
What is this course?



What is this course?

Maths?

- Logic
- Set Theory
- Proofs
- Induction
- Algebra (a bit)
- No calculus 😊

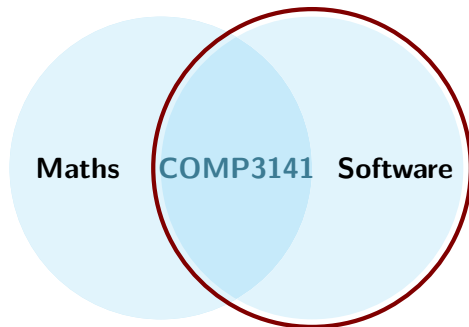


N.B: MATH1081 is neither necessary nor sufficient for COMP3141.

What is this course?

Software?

- Programming
- Reasoning
- Design
- Testing
- Types
- Haskell



N.B: Haskell knowledge is not a prerequisite for COMP3141.

What isn't this course?

This course is **not**:

- a Haskell course
- a verification course (for that, see [COMP6721](#), [COMP4161](#))
- an OOP software design course (see [COMP2511](#), [COMP1531](#))
- a programming languages course (see [COMP3161](#)).
- a WAM booster cakewalk (hopefully).
- a soul-destroying nightmare (hopefully).

Assessment

Warning

For many of you, this course will present a lot of new topics. Even if you are a seasoned programmer, you may have to learn as if from scratch.

- Class Mark (out of 100)
 - **Two** programming assignments, each worth 20 marks.
 - Weekly online quizzes, worth 20 marks.
 - Weekly programming exercises, worth 40 marks.
- Final Exam Mark (out of 100)
 - This includes 10 marks for active participation throughout the term. You will be provided with various methods and opportunities to demonstrate participation throughout the term.

$$result = \frac{class + exam}{2}$$

Lectures

- The lectures will be held on zoom. As there is a limit of 300 attendees on zoom, we will live stream the lectures to YouTube, in case we hit the zoom limit.
- I, Christine, or someone qualified whom I nominate, will run lectures introducing new material on Wednesdays.
- Curtis will reinforce this new material with questions and examples on Fridays.
- Curtis has kindly agreed to offer three help sessions throughout the term. The dates and times for these sessions will be announced at least one week in advance on the course website.
- You **must** attend lectures or watch recordings as they come out.
- Recordings will be available from the course website.
- Online quizzes are due one week after the lectures they examine, but **do them early!**

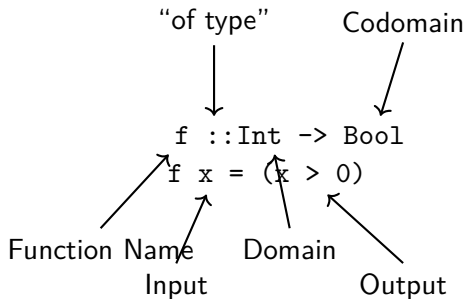
Books

There are no set textbooks for this course, however, there are various books that are useful for learning Haskell listed on the course website.

I can also provide more specialised text recommendations for specific topics.

Haskell

In this course we use **Haskell**, because it is the most widespread language with good support for mathematically structured programming.



In mathematics, we would apply a function by writing $f(x)$. In Haskell we write `f x`.

Demo: GHCi, basic functions

Currying

- In mathematics, we treat $\log_{10}(x)$ and $\log_2(x)$ and $\ln(x)$ as separate functions.
- In Haskell, we have a single function `logBase` that, given a number n , produces a function for $\log_n(x)$.

```
log10 :: Double -> Double  
log10 = logBase 10
```

```
log2 :: Double -> Double  
log2 = logBase 2
```

```
ln :: Double -> Double  
ln = logBase 2.71828
```

What's the **type** of `logBase`?

Currying and Partial Application

```
logBase :: Double -> (Double -> Double)
```

(parentheses optional above)

Function application associates to the **left** in Haskell, so:

$$\text{logBase } 2 \ 64 \quad \equiv \quad (\text{logBase } 2) \ 64$$

Functions of more than one argument are usually written this way in Haskell, but it is possible to use **tuples** instead...

Tuples

Tuples are another way to take multiple inputs or produce multiple outputs:

```
toCartesian :: (Double, Double) -> (Double, Double)
toCartesian (r, theta) = (x, y)
    where x = r * cos theta
           y = r * sin theta
```

N.B: The order of bindings doesn't matter. Haskell functions have no side effects, they just return a result.

Higher Order Functions

In addition to returning functions, functions can take other functions as arguments:

```
twice :: (a -> a) -> (a -> a)
twice f a = f (f a)
```

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

```
quadruple :: Int -> Int
quadruple = twice double
```

Lists

Haskell makes extensive use of lists, constructed using square brackets. Each list element must be of the same type.

```
[True, False, True]    ::    [Bool]
[3, 2, 5+1]             ::    [Int]
[sin, cos]             ::    [Double -> Double]
[ (3,'a'),(4,'b') ]    ::    [(Int, Char)]
```

Map

A useful function is `map`, which, given a function, applies it to each element of a list:

```
map not [True, False, True] = [False, True, False]
map negate [3, -2, 4]       = [-3, 2, -4]
map (\x -> x + 1) [1, 2, 3] = [2, 3, 4]
```

The last example here uses a *lambda expression* to define a one-use function without giving it a name.

What's the type of `map`?

```
map :: (a -> b) -> [a] -> [b]
```

Strings

The type `String` in Haskell is just a list of characters:

```
type String = [Char]
```

This is a *type synonym*, like a `typedef` in C.

Thus:

```
"hi!" == ['h', 'i', '!']
```

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.

Function Composition

We used *function composition* to combine our functions together. The mathematical $(f \circ g)(x)$ is written $(f \ . \ g) \ x$ in Haskell.

In Haskell, operators like function composition are themselves functions. You can define your own!

```
-- Vector addition
```

```
(.+)  
:: (Int, Int) -> (Int, Int) -> (Int, Int)
```

```
(x1, y1) .+ (x2, y2) = (x1 + x2, y1 + y2)
```

```
(2,3) .+ (1,1) == (3,4)
```

You could even have defined function composition yourself if it didn't already exist:

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

```
(f . g) x = f (g x)
```

Lists

How were all of those list functions we just used implemented?

Lists are **singly-linked** lists in Haskell. The empty list is written as `[]` and a list node is written as `x : xs`. The value `x` is called the **head** and the rest of the list `xs` is called the **tail**. Thus:

```
"hi!" == ['h', 'i', '!'] == 'h':('i':('!':[]))  
      == 'h' : 'i' : '!' : []
```

When we define recursive functions on lists, we use the last form for pattern matching:

```
map :: (a -> b) -> [a] -> [b]  
map f []      = []  
map f (x:xs) = f x : map f xs
```

Equational Evaluation

`map f [] = []`

`map f (x:xs) = f x : map f xs`

We can evaluate programs *equationally*:

```
map toUpper "hi!"  ≡ map toUpper ('h':"i!")
                   ≡ toUpper 'h' : map toUpper "i!"
                   ≡ 'H' : map toUpper "i!"
                   ≡ 'H' : map toUpper ('i':"!")
                   ≡ 'H' : toUpper 'i' : map toUpper "!"
                   ≡ 'H' : 'I' : map toUpper "!"
                   ≡ 'H' : 'I' : map toUpper ('!':"")
                   ≡ 'H' : 'I' : '!' : map toUpper ""
                   ≡ 'H' : 'I' : '!' : map toUpper []
                   ≡ 'H' : 'I' : '!' : []
                   ≡ "HI!"
```

Higher Order Functions

The rest of this lecture will be spent introducing various list functions that are built into Haskell's standard library by way of **live coding**.

Functions to cover:

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

In the process, we will introduce **let** and **case** syntax, **guards** and **if**, and the **\$** operator.

Homework

- ➊ Get Haskell working on your development environment. Instructions are on the course website.
- ➋ Using Haskell documentation and GHCi, answer the questions in this week's quiz (**assessed!**).
- ➌ Attend Curtis' online lecture on Friday!