COMP3141

Software System Design and Implementation

Introduction

Paul Hunter
University of New South Wales
Term 2 2024

Acknowledgement of Country

I would like to acknowledge and pay my respect to the Bedegal people who are the Traditional Custodians of the land on which UNSW is built, and of Elders past and present.

Meet the staff

I am Paul Hunter. I'm a lecturer at UNSW. My areas of interest are formal verification, graph theory and algorithms, lecturing, listing my interests. This is the first time I have taught this course.

Meet the staff

I am Paul Hunter. I'm a lecturer at UNSW. My areas of interest are formal verification, graph theory and algorithms, lecturing, listing my interests. This is the first time I have taught this course.

Raphael Douglas Giles will deliver most of the practical lectures (Fridays). This is not the first time he has taught this course.

Meet the staff

I am Paul Hunter. I'm a lecturer at UNSW. My areas of interest are formal verification, graph theory and algorithms, lecturing, listing my interests. This is the first time I have taught this course.

Raphael Douglas Giles will deliver most of the practical lectures (Fridays). This is not the first time he has taught this course.

I am currently setting up Help sessions. These will be announced once I know the details. The staff will have taken the course before.

Contacting Us

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

Forum

There is an Ed forum linked on the course website (click here to join). Ask questions there. To avoid spoiling solutions, you can and should ask private questions.

Administrative questions should be sent to the course email

`cs3141@cse.unsw.edu.au`

Student Support

For help with anything else, there is always

Student Support - I Need Help With...

Screenshot This Slide

Uni and Life in Australia Stress, Financial, Visas, Accommodation & More	**Student Support**	student.unsw.edu.au/**advisors**	
Reporting Sexual Assault/Harassment	**Equity Diversity and Inclusion (EDI)**	edi.unsw.edu.au/**sexual-misconduct**	
Educational Adjustments To Manage my Studies and Disability / Health Condition	**Equitable Learning Services (ELS)**	student.unsw.edu.au/**els**	
Academic and Study Skills	**Academic Skills**	student.unsw.edu.au/**skills**	
Special Consideration Because Life Impacts our Studies and Exams	**Special Consideration**	student.unsw.edu.au/**special-consideration**	

My Feelings and Mental Health Managing Low Mood, Unusual Feelings & Depression	**Mental Health Connect**	student.unsw.edu.au/**counselling** Telehealth	**In Australia Call Afterhours** **UNSW Mental Health Support Line**	1300 787 026 5pm-9am
	Mind HUB	student.unsw.edu.au/**mind-hub** Online Self-Help Resources	**Outside Australia Afterhours** **24-hour Medibank Hotline**	+61 (2) 8905 0307

What is this course?

Our software should be
correct, safe and secure.

Our software should be
developed cheaply and quickly.

Safety-uncritical Applications


;

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

Safety-critical Applications

Think of the worst group assignment you ever had!

Safety-critical Applications

Think of the worst group assignment you ever had! Imagine you...

- are logging into your online banking...
- are investing in a new hedge fund...
- are travelling in a self-driving car...
- are travelling on a plane...
- are getting treatment from a radiation therapy machine...
- are about to launch nuclear missiles...

Safety-critical Applications

Think of the worst group assignment you ever had! Imagine you. . .

- are logging into your online banking. . .
- are investing in a new hedge fund. . .
- are travelling in a self-driving car. . .
- are travelling on a plane. . .
- are getting treatment from a radiation therapy machine. . .
- are about to launch nuclear missiles. . .

. . . using software written by your groupmates from that group.

Safety-critical Applications

Airline Blames Bad Software in San Francisco Crash

The New York Times

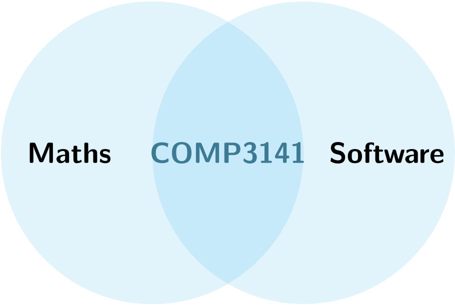
Safety-critical Example

What is wrong with this code:

Example

```
transfer(account to, account from, uint amount){
 require (balances[from] > amount);
 balancesFrom := balances[from] - amount;
 balancesTo := balances[to] + amount;
 balances[from] := balancesFrom;
 balances[to] := balancesTo;
}
```

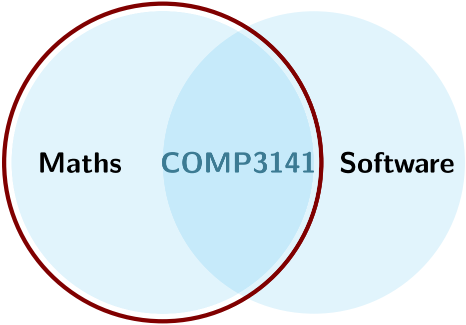
What is this course?



What is this course?

Maths?

- Logic
- Sets
- Proofs
- Induction
- Algebra (a bit)
- but no Calculus

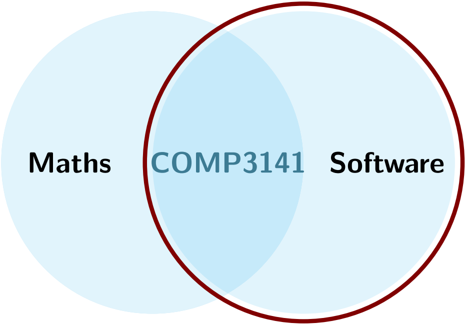


MATH1081 is neither necessary nor sufficient for COMP3141.

What is this course?

Software?

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



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

What this course is not?

What this course is not?

- not a Haskell course

What this course is not?

What this course is not?

- **not a Haskell course**
- **not** a formal verification course (see COMP3153/COMP4161),

What this course is not?

What this course is not?

- **not a Haskell course**
- **not** a formal verification course (see COMP3153/COMP4161),
- **not** an OOP software design course (see COMP2511),

What this course is not?

What this course is not?

- **not** a **Haskell course**
- **not** a formal verification course (see COMP3153/COMP4161),
- **not** an OOP software design course (see COMP2511),
- **not** a programming languages course (see COMP3161).

What this course is not?

What this course is not?

- **not a Haskell course**
- **not** a formal verification course (see COMP3153/COMP4161),
- **not** an OOP software design course (see COMP2511),
- **not** a programming languages course (see COMP3161).
- Certainly **not** a cakewalk; but hopefully **not** a soul-crushing nightmare either.

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 you were starting from scratch.

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 you were starting from scratch.

- Class Marks (out of 100)
  - **Two** programming assignments, each worth 20 marks.
  - Weekly online quizzes, worth 20 marks.
  - Weekly programming exercises, worth 40 marks.
- Final Exam Marks (out of 100, hurdle: 40)

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

Lectures

- **Lecture (Wed 11am-1pm):** I introduce new material.
- **Practical (Fri 11am-1pm):** Scott (usually) reinforces Wednesday's material with questions and examples.
- **Quiz**: due on Fri (one week after the lectures they examine), but start early!

Books

We won't set a textbook (a long COMP3141 tradition).

Resources: see the course outline for various books and online resources that are useful for learning Haskell.

Why Haskell?

- This course uses Haskell, because it is the most widely used language with good support for *mathematically structured programming*.

Why Haskell?

- This course uses Haskell, because it is the most widely used language with good support for *mathematically structured programming*.
- You will learn a substantial amount of Haskell (we will provide some guidance). But the course is about learning techniques for mathematically structured programming.

Why Haskell?

- This course uses Haskell, because it is the most widely used language with good support for *mathematically structured programming*.
- You will learn a substantial amount of Haskell (we will provide some guidance). But the course is about learning techniques for mathematically structured programming.
- Based on feedback from previous iterations, I will endeavour to include examples from other languages.

About Haskell

- Haskell is old!

About Haskell

- Haskell is old! It's turning 34 this year.

About Haskell

- Haskell is old! It's turning 34 this year.
- Throughout the years: **Haskell 98**, **Haskell 2010**, **GHC2021**.

About Haskell

- Haskell is old! It's turning 34 this year.
- Throughout the years: **Haskell 98**, **Haskell 2010**, **GHC2021**.

Warning

This means that some (possibly even most) tutorials, resources, answers you find on the Internet will be outdated!

Demo 1: Haskell Workflow

- Now we'll give you a Haskell Crash Course.
- This is to get you coding (solving problems) quickly.

Demo 1: Haskell Workflow

- Now we'll give you a Haskell Crash Course.
- This is to get you coding (solving problems) quickly.
- If you prefer "deep" understanding, don't worry: next week.

Demo 1: Haskell Workflow

- Now we'll give you a Haskell Crash Course.
- This is to get you coding (solving problems) quickly.
- If you prefer "deep" understanding, don't worry: next week.

Demo: GHCi, Modules

Demo 2: Declaring Functions

```
isEven :: Int -> Bool
```

Name of the function

Demo 2: Declaring Functions

“has type”

```
isEven :: Int -> Bool
```

Name of the function

Demo 2: Declaring Functions

"has type" Domain

```haskell
isEven :: Int -> Bool
```

Name of the function

Demo 2: Declaring Functions

“has type” Domain Codomain

```
isEven :: Int -> Bool
```

Name of the function

Demo 2: Declaring Functions

```
isEven :: Int -> Bool
isEven x = x `mod` 2 == 0
```

Argument (`Int`) Result (`Bool`)

Demo 2: Declaring Functions

```
isEven :: Int -> Bool
isEven x = x `mod` 2 == 0
```

Argument (`Int`) Result (`Bool`)

In mathematics, we would apply a function f to an argument x by writing $f(x)$. In Haskell we write `f x`, omitting the parentheses.

Demo: basic functions

Demo 3: Currying

- Haskell functions have one input domain and one output codomain. But some functions take multiple inputs.

Demo 3: Currying

- Haskell functions have one input domain and one output codomain. But some functions take multiple inputs.
- 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 $\log_n(x)$.

```haskell
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`?

Demo 3: Currying

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

Demo 3: Currying

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

(parentheses are optional above, we could write:)

```haskell
logBase :: Double -> Double -> Double
```

Demo 3: Currying

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

(parentheses are optional above, we could write:)

```
logBase :: Double -> Double -> Double
```

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

`logBase 2 64` $\equiv$ `(logBase 2) 64`

Demo: currying, multiple arguments

Demo 4: Tuples

We now know how to handle multiple inputs to a function? But what if we want to have multiple outputs?

Demo 4: Tuples

We now know how to handle multiple inputs to a function? But what if we want to have multiple outputs?
Haskell provides data types called tuples to handle multiple outputs:

```haskell
neighbors :: Int -> (Int, Int)
neighbors x = (x - 1, x + 1)
```

Now, `(neighbors 1)` evaluates to `(0,2)`.

Demo: tuples

Demo 5: Higher Order Functions

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

```haskell
applyTwice :: (t -> t) -> t -> t
applyTwice f x = f (f x)

square :: Int -> Int
square x = x * x

fourthPower :: Int -> Int
fourthPower = applyTwice square
```

Demo: higher-order functions, equational reasoning

Demo 6: 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)]
```

Demo 6: Lists

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 square [3, -2, 4]       = [9, 4, 16]
map (\x -> x + 1) [1, 5]    = [2, 6]
```

Demo 6: Lists

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 square [3, -2, 4]       = [9, 4, 16]
map (\x -> x + 1) [1, 5]    = [2, 6]
```

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?

Demo 6: Lists

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 square [3, -2, 4]       = [9, 4, 16]
map (\x -> x + 1) [1, 5]    = [2, 6]
```

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]
```

Demo 6: Lists

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

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

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

Thus:

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

Demo: lists

Word Frequencies

Let's solve a problem to get some practice implementing stuff:

Example (Task 1)

Given a number n and a string s containing English words, generate a report that lists the n most common words in the given string s.

I'll even give you an algorithm:

Word Frequencies

Let's solve a problem to get some practice implementing stuff:

Example (Task 1)

Given a number *n* and a string *s* containing English words, generate a report that lists the *n* most common words in the given string *s*.

I'll even give you an algorithm:

1. Break the input string into words.
2. Convert the words to lowercase.
3. Sort the words.
4. Group adjacent occurrences (runs) of the same word.
5. Sort runs words by length.
6. Take the longest *n* runs of the sorted list.
7. Generate a report.

Demo: word frequencies

The Dollar Pattern

We used *the dollar operator* `$` to reduce the use of parentheses.

- The dollar operator does normal function application, like `f x` (evaluation of a function at a value).
- However, while application has high operator precedence ("is done as early as possible"), the dollar operator has extremely low precedence ("is done as late as possible").

The Dollar Pattern

We used *the dollar operator* `$` to reduce the use of parentheses.

- The dollar operator does normal function application, like `f x` (evaluation of a function at a value).
- However, while application has high operator precedence ("is done as early as possible"), the dollar operator has extremely low precedence ("is done as late as possible").
- `reverse [1,2,3] ++ [4]` results in `[3,2,1,4]`. The application of the `reverse` function binds very tightly, so we do it first, then concatenate.

The Dollar Pattern

We used *the dollar operator* `$` to reduce the use of parentheses.

- The dollar operator does normal function application, like `f x` (evaluation of a function at a value).
- However, while application has high operator precedence ("is done as early as possible"), the dollar operator has extremely low precedence ("is done as late as possible").
- `reverse [1,2,3] ++ [4]` results in `[3,2,1,4]`. The application of the `reverse` function binds very tightly, so we do it first, then concatenate.
- `reverse $ [1,2,3] ++ [4]` results in `[4,3,2,1]`. We concatenate first, then apply the reversing function. Same as `reverse ([1,2,3] ++ [4])`.

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!

```haskell
-- Vector addition
(.+) :: (Int, Int) -> (Int, Int) -> (Int, Int)
(x1, y1) .+ (x2, y2) = (x1 + x2, y1 + y2)
```

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

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)
```

Conditionals

Demo: polarity using guards, if statements.

Demo: (if we have time), loops via recursion.

Lists

We used a bunch of list functions. How could we implement them ourselves??

Lists

We used a bunch of list functions. How could we implement them ourselves??

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' : '!' : []
```

Lists

We used a bunch of list functions. How could we implement them ourselves??

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

```haskell
map f []     = []
map f (x:xs) = f x : map f xs
```

We can evaluate programs *equationally*:

```haskell
map toUpper "hi!"
```

Equational Evaluation

```haskell
map f []     = []
map f (x:xs) = f x : map f xs
```

We can evaluate programs *equationally*:

```haskell
map toUpper "hi!"  ≡  map toUpper ('h':"i!")
```

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!"
```

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!"
```

Equational Evaluation

```haskell
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':"!")
```

Equational Evaluation

```haskell
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 "!"
```

Equational Evaluation

```haskell
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 "!"
```

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 ('!':"")
```

Equational Evaluation

```haskell
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 ""
```

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 []
```

Equational Evaluation

```haskell
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' : '!' : []
```

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!"
```

FIN

The quiz will be up on the course website sometime on Friday.

Warning

The quiz is assessed. The deadline is the end of next Friday.