COMP3141

Dr. Liam O'Conn
University of Edinburgh LFCS (an

Term 2 2020

Who are we?

I am Dr. Liam O'Connor, a lecturer in Programming Languages for Trustworthy Systems at the University of Edinburgh, currently visiting UNSW to teach this course.

I produce these lect

Who are we?

I am Dr. Liam O'Connor, a lecturer in Programming Languages for Trustworthy Systems at the University of Edinburgh, currently visiting UNSW to teach this course. I produce these lect

Curtis Millar, the te
trustworthy syst p at
data61.

Who are we?

I am Dr. Liam O'Connor, a lecturer in Programming Languages for Trustworthy Systems at the University of Edinburgh, currently visiting UNSW to teach this course. I produce these lect [illegible]

Curtis Millar, the te [illegible] trustworthy syst [illegible] p at data61.

Prof. Gabriele Keller, who now works at Utrecht University, is t [illegible] this course. Her research interests revolve around program [illegible] methods and high performance computing. Hopefully we can maintain the high standard she set.

Contacting Us

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

Forum

There is a Piazza for
should typically be made there. You can ask us private question
solutions to other students.

I highly recommend disabling the Piazza Careers rubbish.

Administrative questions should be sent to liamoc@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.

Safety-critical Applications

Remember a particularly painful uni group work assignment.

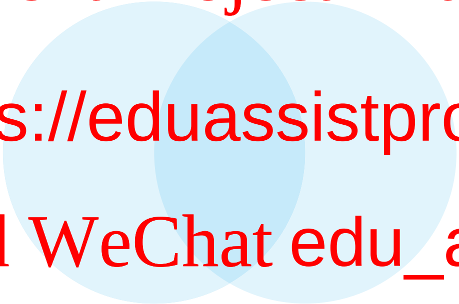
Now imagine you. . .

- Are travelli
- Are travell
- Are workin
- Have invested in a new hedge fund
- Are running a cryptocurrency exchange
- Are getting treatment from a radiation therapy machi
- Intend to launch some nuclear missiles at your enemies

. . . running on software written by other members of that group.

Safety-critical Applications

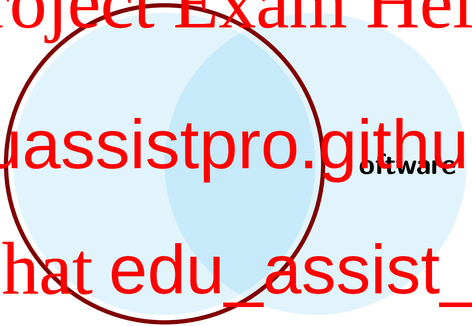
What is this course?



What is this course?

Maths?

- Logic
-
-
- Induction
- Algebra (a bit)
- No calculus ☺

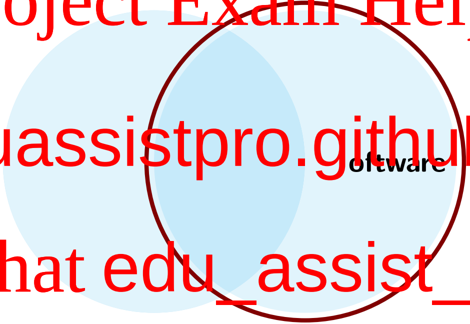


N.B: MATH1081 is neither necessary nor sufficient for COMP3141.

What is this course?

Software?

- Programming
- Testing
- Types
- Haskell



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

What isn't this course?

This course is **not**:

- a Haskell co

What isn't this course?

This course is **not**:

- **a Haskell co**
- a verificati

What isn't this course?

This course is **not**:

- **a Haskell co**
- a verificati
- an OOP soft

What isn't this course?

This course is not:

- **a Haskell co**
- a verificati
- an OOP soft
- a programming languages course (see COMP3161).

What isn't this course?

This course is not:

- **a Haskell co**
- a verificati
- an OOP soft
- 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 progra

Assessment

Warning

For many of you, this course will present a lot of new topics. Even if you are a seasoned progra

- Class Mark (
  - **Two**
  - Weekly online quizzes, worth 20 marks.
  - Weekly programming exercises, worth 40 marks.
- Final Exam Mark (out of 100)

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

Lectures

- Lecture videos like this one are released once per week. These generally introduce new material.
- Curtis will ru
  new materi
  This lectur i

- You must watch recordings as they come out.
- Recordings are available from the course website.

- All board-work will be done digitally and made available t
- Online quizzes are due one week after the lectures they examine, but do them early!

Books

There are no set text.

useful for learning

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.


```
f :: Int -> Bool
```


Function Name

Haskell

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


```
f :: Int -> Bool
```


Function Name

Haskell

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


```
f :: Int -> Bool
```

Function Name   Domain

Haskell

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


```
f :: Int -> Bool
```

Function Name   Domain

Haskell

In this course we use Haskell, because it is the most widespread language with good support for mathe

Input

Haskell

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

Input

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 fo l

```
log1
log1
```



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



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

What's the type of `logBase`?

Currying and Partial Application


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

Currying and Partial Application


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


Function applic

```
logBase 2 64   ≡   (
```

Currying and Partial Application


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

Function applic


```
logBase 2 64  ≡  (
```


Functions of more than one argument are usually written this w
possible to use tuples instead...

Tuples

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

```
toCartesian :
toCartesian (
    where x = r * cos th
          y = r * sin theta
```


N.B: The order of bindings doesn't matter. Haskell functions h
just return a result.

Higher Order Functions

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

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

double :: Int -> In
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 t


```
[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 negat                   = [
map (\x -> x + 1) [
```

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 negat                   = [
map (\x -> x + 1) [
```


The last example h n without giving it a name.

What's the type of map?

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 negat                   = [
map (\x -> x + 1) [
```


The last example h n without giving it a name.

What's the type of map?

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

Strings

The type Stri

```
type String = [Ch
```

This is a *type sy*

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 wo

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 wo

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

            (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
(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

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 xs is called the tail. Thus:


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

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` … `xs` is called the tail. Thus:


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

When we define recursive functions on lists, we use the last form f…


```haskell
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 toU
```

Equational Evaluation

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

We can evaluate programs *equationally*:

```
map toU
```

Equational Evaluation

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

We can evaluate programs *equationally*:

```
map toU
```

Equational Evaluation

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

We can evaluate programs *equationally*:

```
map toU
```

Equational Evaluation

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

We can evaluate programs *equationally*:

```
map toU
```

Equational Evaluation

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


We can evaluate programs *equationally*:

```
map toU
```



```
≡ 'H' : toUpper 'i
```

Equational Evaluation

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

We can evaluate programs *equationally*:


```
map toU
```



```
≡  'H' : toUpper 'i
≡  'H' : 'I' : map toUp
```

Equational Evaluation

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


We can evaluate programs *equationally*:

```
map toU
```



```
≡ 'H' : toUpper 'i
≡ 'H' : 'I' : map toUp
≡ 'H' : 'I' : map toUp
```

Equational Evaluation

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

We can evaluate programs *equationally*:

```
map toU
```



```
≡ 'H' : toUpper 'i
≡ 'H' : 'I' : map toUp
≡ 'H' : 'I' : map toUp
≡ 'H' : 'I' : '!' : map toUpper ""
```

Equational Evaluation

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

We can evaluate programs *equationally*:

```
      map toU
```



```
≡ 'H' : toUpper 'i
≡ 'H' : 'I' : map toUp
≡ 'H' : 'I' : map toUp
≡ '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 toU

≡ 'H' : toUpper 'i
≡ 'H' : 'I' : map toUp
≡ 'H' : 'I' : map toUp
≡ '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 toU
```



```
≡ 'H' : toUpper 'i
≡ 'H' : 'I' : map toUp
≡ 'H' : 'I' : map toUp
≡ '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

1. Get Haskell
   course web
2. Using Hask
   (assessed!).
3. Attend Curtis' online lecture on Wednesday!