# Introduction



## ECS713 : Functional Programming
## Week 01

Prof. Edmund Robinson
Dr Paulo Oliva

# Week 1: Lecture Plan

1. Student Online Survey



2. ECS713 Module Structure





3. History of Functional Programming





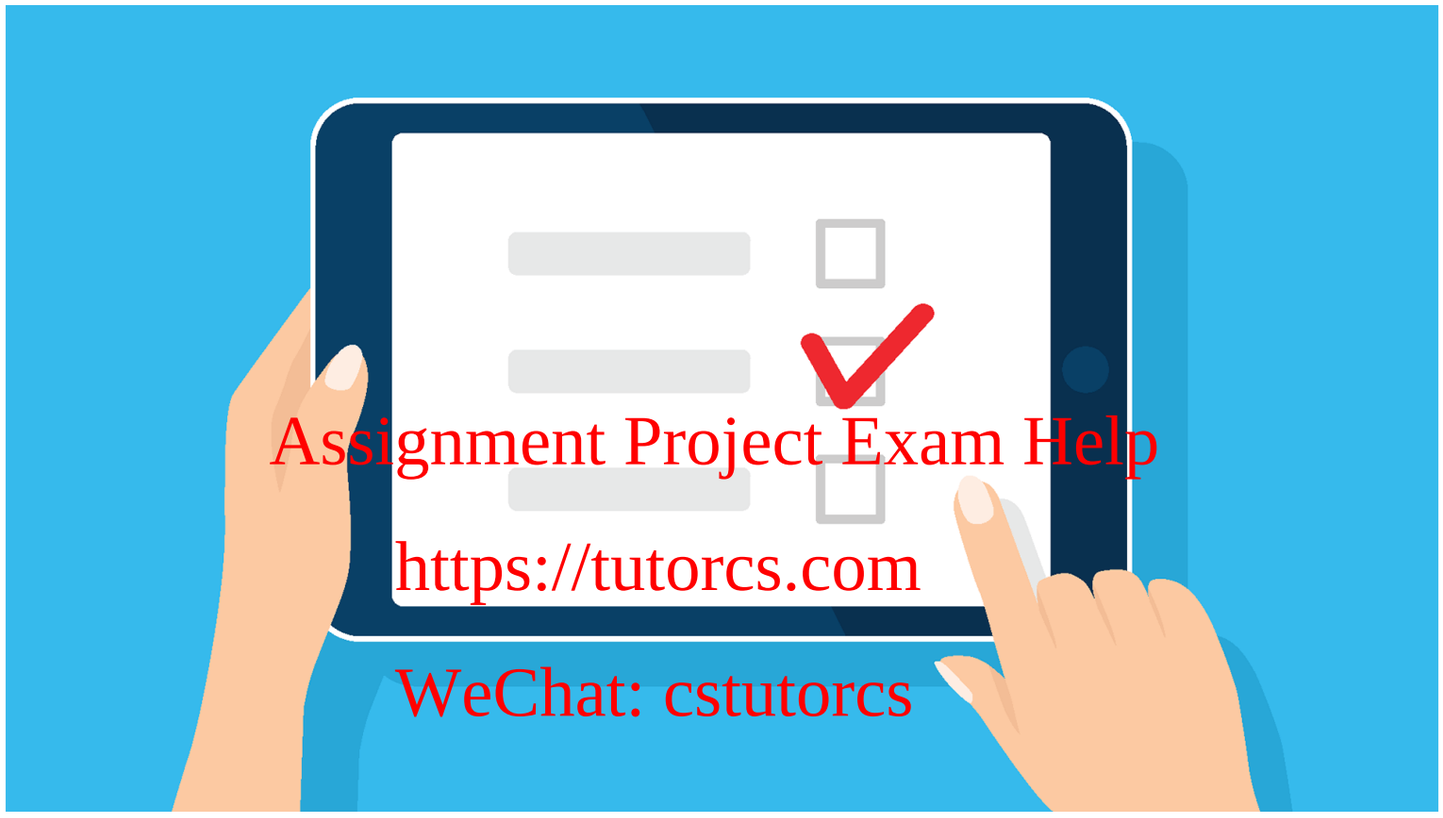
4. Using Jupyter Hub / Haskell stack

5. Core Haskell: Definitions, Types, Tuples, Lists

6. Student Quiz



7. First Haskell Program



# Module Structure

# Teaching Staff



Edmund Robinson
e.p.robinson@qmul.ac.uk

Paulo Oliva
p.oliva@qmul.ac.uk

# Teaching

https://qmplus.qmul.ac.uk/course/view.php?id=15499



- **Online Lecture**
  Thursdays 4-6pm
- **On-Campus Labs (ITL)**
  Fridays 2-4pm

# Outline

- To give a structured introduction to the language Haskell
- To familiarise the student with the underlying type structures and the basic programming methodology
- To exhibit more advanced type structures (such as functors) and programming techniques such as map-reduce and monadic programming, and to illustrate how these are used to give flexible extendible and parallelisable solutions to programming problems



https://intranet.eecs.qmul.ac.uk/courses/descriptor/eecsismodule/mod/ECS713P

# Motivation

- Increased interest in Functional Programming Languages/Functional Programming Techniques
- Modern languages (Ruby, Python,…) incorporated functional programming ideas
- Key motivations:
  - Security: Isolation of risky interactive bits of code
  - Efficiency: Easy to parallelise

# Learning Outcomes

https://LearnOuts.com



## Week 1 (27 Sep - 02 Oct)

### Functional Programming / History

- be familiar with the history of functional programming

### Functional Programming / Functional vs imperative programming languages

- be able to explain the difference between functional and non-functional languages

### Functional Programming / Installing Haskell

- have successfully edited and run a basic Haskell program
- have successfully downloaded and installed Haskell on your own computer (or be using Jupyter notebooks on the cloud)

### Core Haskell / Definitions / declarations

- write down declarations with basic expressions

### Types / Basic Types

- understand the distinction between Char and String
- use the :type (or simply :t) command in GHCI to discover the type of a given expression

### Types / Tuples

| Week | Topics |
|---|---|
| 1 | Introduction, history, first Haskell program |
| 2 | Core Haskell |
| 3 | Recursion |
| 4 | Higher-order functions |
| 5 | Type classes |
| 6 | IO Actions |
| 7 | Database and HTTP requests |
| 8 | Parsing and Exceptions |
| 9 | Functor and Monads I |
| 10 | Functor and Monads II |
| 11 | Concurrency and Parallelism I |
| 12 | Concurrency and Parallelism II |

# Language

- We will be using Haskell (Glasgow, Yale, Microsoft Research)
- Version: 8.10
- Modern standard functional languages
- Alternatives: F#, ML, Occaml

# Assessment

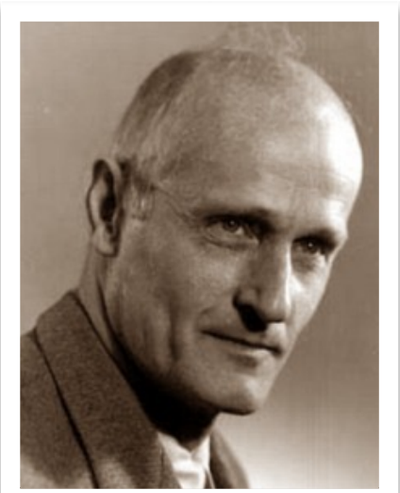
- Two QM+ online quizzes (10% each)
  During labs on weeks 5 & 9
- Group project (45%)
  Due end of term
- Individual project (35%)
  Due during exam period (Jan 2022)

# A Bit of History…

# Some history

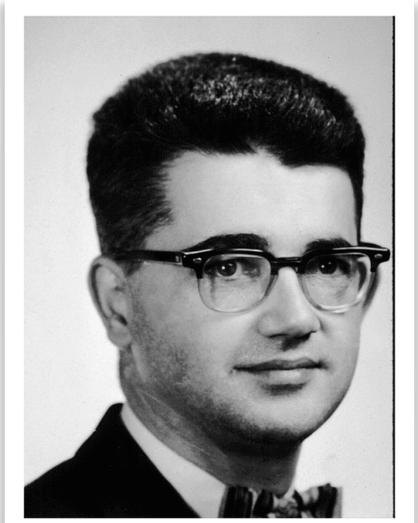
- The idea of functional programming has been around a long time...
- Alonzo Church (1903-1995)
  - lambda calculus (1936)
- Stephen Kleene (1909-1994)
  - formal recursive functions (1930s)
- Predates Turing Machines (just)

A. Church



S. Kleene

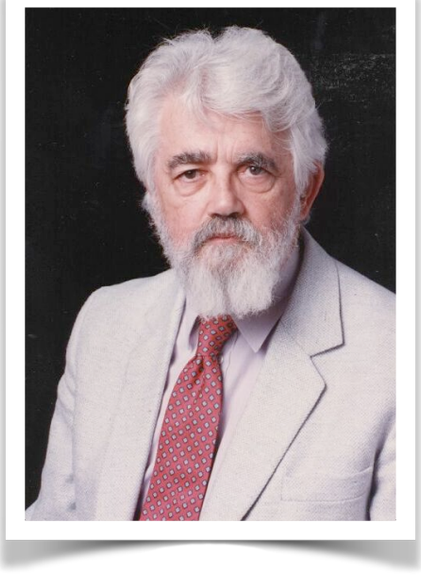
# John McCarthy (Stanford, 1927-2011)



young



- Key pioneer of AI
- Invented Lisp (1958)
- One of the first high-level programming languages (along with Fortran, Cobol, Algol)
- Used in (symbolic) AI as the primary language for over 30 years
- Idea: lists as a fundamental data structure
- First taught language at MIT for many years



not so young

# Peter Landin (1930-2009, QM)

If you See What I Mean



- Invented languages (ISWIM, 1965), and implementation techniques (SECD machine)





- Pioneered **lazy** functional programming as in Haskell
- Goal: to do equational reasoning (requires referential transparency)

# Robin Milner
# (Edinburgh, Cambridge,1934-2010)



- Won the Turing Prize (1991)
- Standard ML
- Originally developed as a special purpose language for describing proofs and operations on proofs. Developed a life of its own
- Innovative type system
- First formally specified language

For three distinct and complete achievements:

1. LCF, the mechanization of Scott's Logic of Computable Functions, probably the first theoretically based yet practical tool for machine assisted proof construction;
2. ML, the first language to include polymorphic type inference together with a type-safe exception-handling mechanism;
3. CCS, a general theory of concurrency

# David Turner
# (Kent, 1946 - )



- Designed and implemented Miranda (1985)
- Synthesised many of the extant ideas about how to design and structure a lazy functional language
- But tried to sell it (unpopular)

# Haskell (1990)

- Was originally a community answer to Miranda… intended as community property
- Named after Haskell B Curry, an American logician
- Glasgow & Microsoft Research: Simon Peyton-Jones

# Haskell


Declarative, statically typed code.



```haskell
primes = filterPrime [2..]
  where filterPrime (p:xs) =
          p : filterPrime [x | x <- xs, x `mod` p /= 0]
```



**Haskell** /ˈhæskəl/[25] is a standardized, general-purpose purely functional programming language, with non-strict semantics and strong static typing.[26]

# Declarative = Definitions

- The idea behind functional programming is to concentrate on building data structures using functions that are like mathematical functions

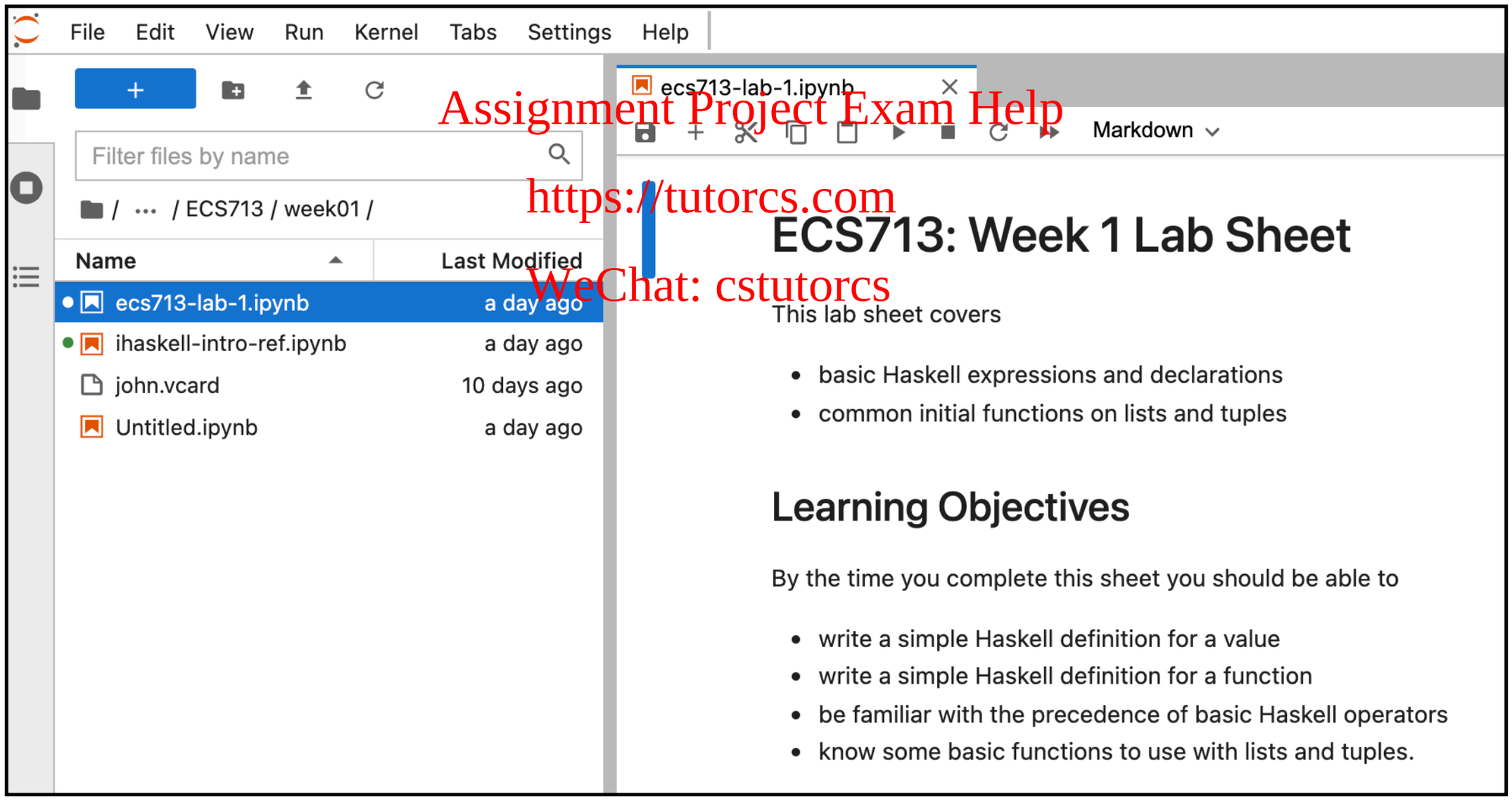


- So the code can look very like the kind of definitions you get in Maths

```
-- increment
inc n = n + 1
```

# Jupyter Hub

https://jhub.eecs.qmul.ac.uk/



# Installing on your own machines

- Download from http://www.haskell.org/
- "Minimal install" for now:



https://www.haskell.org/downloads#minimal

- Recommended: Using Haskell **stack** (a Haskell package manager)

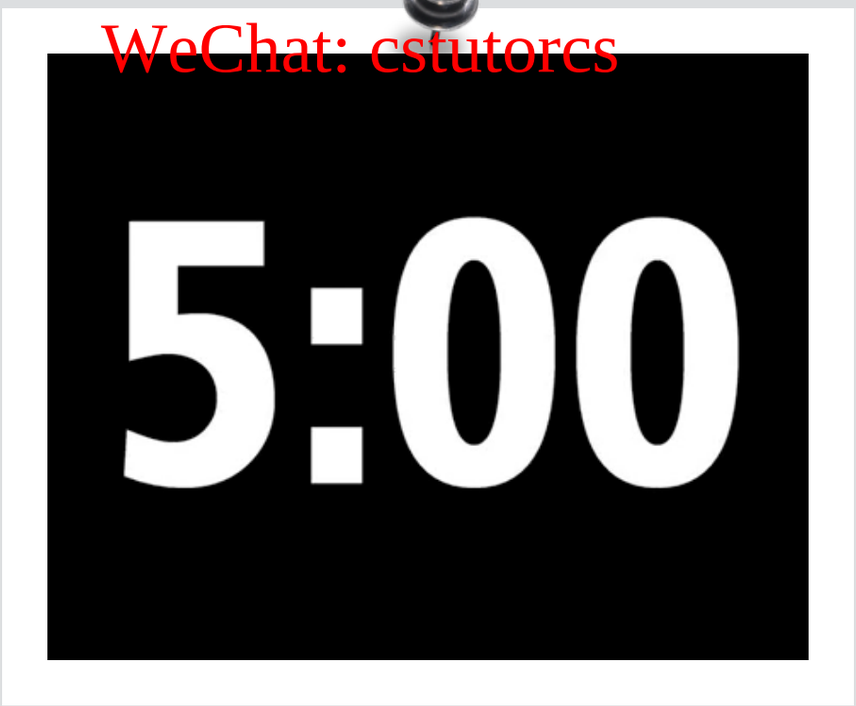
https://www.haskell.org/downloads#stack

# Haskell

- Two key bits
- ghc
  - Glasgow Haskell compiler
  - generates machine code
- ghci
  - Haskell Interpreter
  - an interactive shell

# Short Break





# Three Concepts

- Declarative programming
- Types
- Lists and Tuples

# Learning Haskell

# Back to Haskell…

We can write a whole load of simple arithmetical functions



```
Prelude> let dec n = n - 1
Prelude> dec 3
```



```
2
```



```
Prelude> let square n = n * n
Prelude> square 3
9
Prelude> let disc a b c = (b * b) - (4 * a * c)
Prelude> :type disc
disc :: Num a => a -> a -> a -> a
```

# Declaring Constants

You can of course declare a constant

```
Prelude> let name = "Paulo"
Prelude> :type name
name :: [Char]
Prelude> let age = 40
Prelude> :type age
age :: Num a => a
Prelude> let children = ["Amanda", "Camila"]
Prelude> :type children
children :: [[Char]]
```

# Static Types

- **Static**: The types of all expressions and functions are determined at before running the code



- If types don't match the code won't even run!
- Types are fixed, don't change at run-time
- Most bugs are caught early

# Basic Types

- Char (Characters)
  - { ‘a’, ‘b’,… }
- Bool (Booleans)
  - { True, False }
  - operations: || (or), && (and), not
- Int (Integers)

# Tuples and Lists

- These are how you put stuff together.
- There are two key differences:
  - the elements in a **list** all have to have the same type, the elements in a **tuple** don't
  - **lists** can have an arbitrary number of elements, the number of components in a tuple is fixed by the type of the **tuple** (e.g. 3)

# Tuples

- Tuples

```
-- a pair
john = ("John Doe", "01234 567890")
```

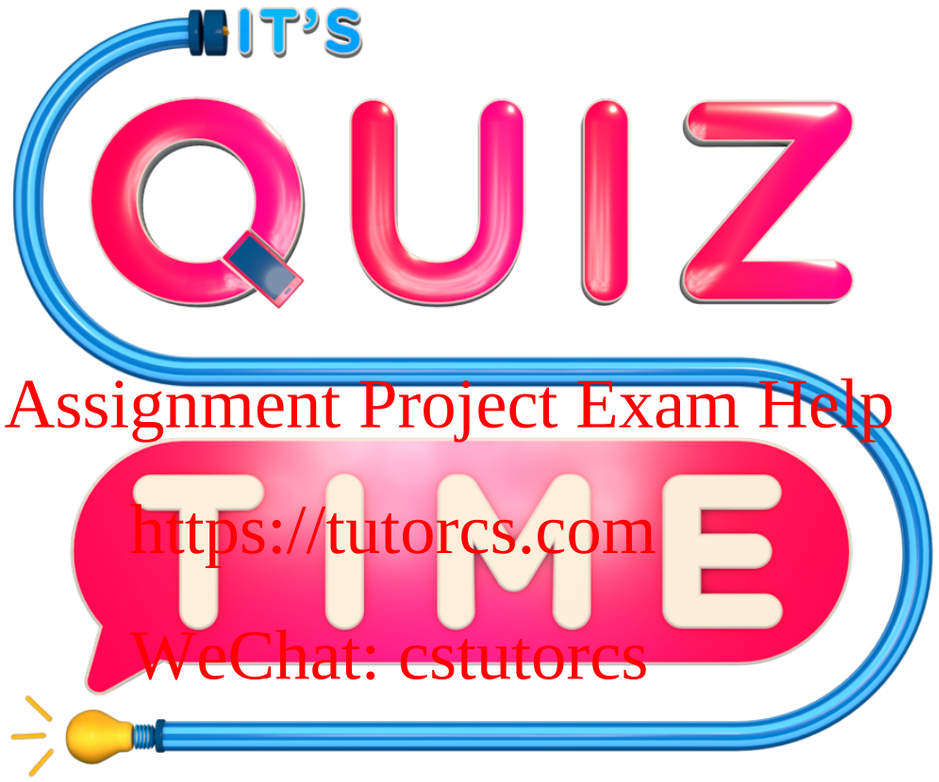
```
-- a triple
john = ("John Doe", "01234 567890", 24)
```

- Get stuff out by pattern matching

```
-- first component
fst (x,y) = x
```

# List and arrays

- Lists are pretty much the arrays of functional programming
- There are two ways to write operations using lists
  - recursion (see later in course)
  - combinators (also later for detail)
- The idea of the combinatory style is that there are various standard operations that allow you to process lists: filter, map, zip, fold,...
- You can get a very long way with just those combinators






https://cloudsurvey.co.uk

# First Haskell Program

# Vcard

```
BEGIN:VCARD
VERSION:3.0
PRODID:-//Apple Inc.//Address Book 6.1.2//EN
N:Doe;John;;;
FN:Doe John
ORG:Queen Mary;
EMAIL;type=INTERNET;type=HOME;type=pref:JohnDoe@nogmail.com
TEL;type=CELL;type=VOICE;type=pref:0751234567
TEL;type=HOME;type=VOICE:02071234567
item1.ADR;type=HOME;type=pref:;;42 Nowhere St;London;;E1 0XX;
item1.X-ABADR:gb
X-ABUID:85152BB5-BFB5-45DA-853A-BA021C7A0FC8:ABPerson
END:VCARD
```

# Vcard

- Let's suppose
  - we already have the card as a string
  - want to extract the list of phone numbers
- Here's a plan of attack:
  1. Split card into lines
  2. Filter out the lines beginning with "TEL"
  3. Process each line removing everything up to and including the first colon

# The Program

```
-- does line begin with "TEL"?
isphone s = (take 3 s) == "TEL"

-- remove up to colon
strip = tail . init . dropWhile (/=':')

-- phone list
getPhones = (map strip) . (filter isphone) . lines
```



Process each line removing everything up to and including the first colon

Filter out the lines beginning with “TEL”

Split card into lines

# Explicit Types

```
-- does line begin with "TEL"?
isphone :: String -> Bool
isphone s = (take 3 s) == "TEL"

-- remove up to colon
strip :: String -> String
strip = tail . init . dropWhile (/=':')

-- phone list
getPhones :: String -> [String]
getPhones = (map strip) . (filter isphone) . lines
```

String

1. Split card into lines

[String]



2. Filter out the lines beginning with “TEL”





[String]

3. Process each line removing everything up to and including the first colon

[String]

# 1. Split card into lines

There is a built in function to do this!

```
Prelude> :type lines
lines :: String -> [String]
Prelude> lines "line 1\nline 2\nline 3"
["line 1","line 2","line 3"]
```



How do I know the built-in functions??

https://www.haskell.org/hoogle/

# 2. Filter out the lines beginning with “TEL”

- The basic combinator we need is: **filter**
- Type of filter: (a -> Bool) -> [a] -> [a]
- a is generic here, this means the function takes a test for things of type a, and a list and returns another list
- Example: filter even [1,2,3,4] == [2,4]
- So all we need is a test to see if a line begins with “TEL”

## 2. Filter out the lines beginning with “TEL”

- The test is simple:
  - take the first three elements and see if they are the string “TEL”



- Again we use a standard function: take
- The type of take is:
  take :: Int -> [a] -> [a]
- Example:
  take 2 [1,2,3,4] == [1,2]

## 2. Filter out the lines beginning with “TEL”



So what we need is:





isphone s = (take 3 s)=="TEL"

# 3. Process each line removing everything up to and including the first colon

- This means we have to do something (the same thing) for each line
- There's another standard function: map
- The type of map is: (a -> b) -> [a] -> [b]
- It takes an operation and a list, and applies the operation to each element of the list
- Example: map even [1,2,3] == [False,True,False]
- The operation we need is "remove everything up to and including the first colon"

# 3. Process each line removing everything up to and including the first colon

- No surprise that there's another standard function... dropWhile
- dropWhile takes a test and a list, and returns the list minus the longest prefix for which the test returns true on each element
- Work out the type of dropWhile
- So our test is: notcolon c = not (c==':')
- And our processing will be:

  dropWhile notcolon line

# More Verbose Version

```
-- does line begin with "TEL"?
isphone s = (take 3 s) == "TEL"

-- remove up to colon
strip s = tail.init $ dropWhile notcolon s
   where
     notcolon c = not (c == ':')

-- phone list
getPhones card = phones
   where
     all_lines = lines card
     phone_lines = filter isphone all_lines
     phones = map strip phone_lines
```

# Testing the Program

```
iMac{oliva}: ghci
GHCi, version 7.10.2: http://www.haskell.org/ghc/
Prelude> :load lect01.hs
[1 of 1] Compiling Main             ( lect01.hs, interpreted )
Ok, modules loaded: Main.
*Main> johnDoe
"BEGIN:VCARD\r\nVERSION:3.0\r\nPROD...END:VCARD\r\n"
*Main> getPhones johnDoe
["0751 234567","020 7123 4567"]
```

# References

- Learn you a Haskell for Great Good
  Miran Lipovača, Chapters 1 and 2
  http://learnyouahaskell.com/chapters
- Real World Haskell
  B. O'Sullivan et al, Chapters 1 and 2
  http://book.realworldhaskell.org/read/
- Programming in Haskell
  Graham Hutton, Chapters 1 and 2