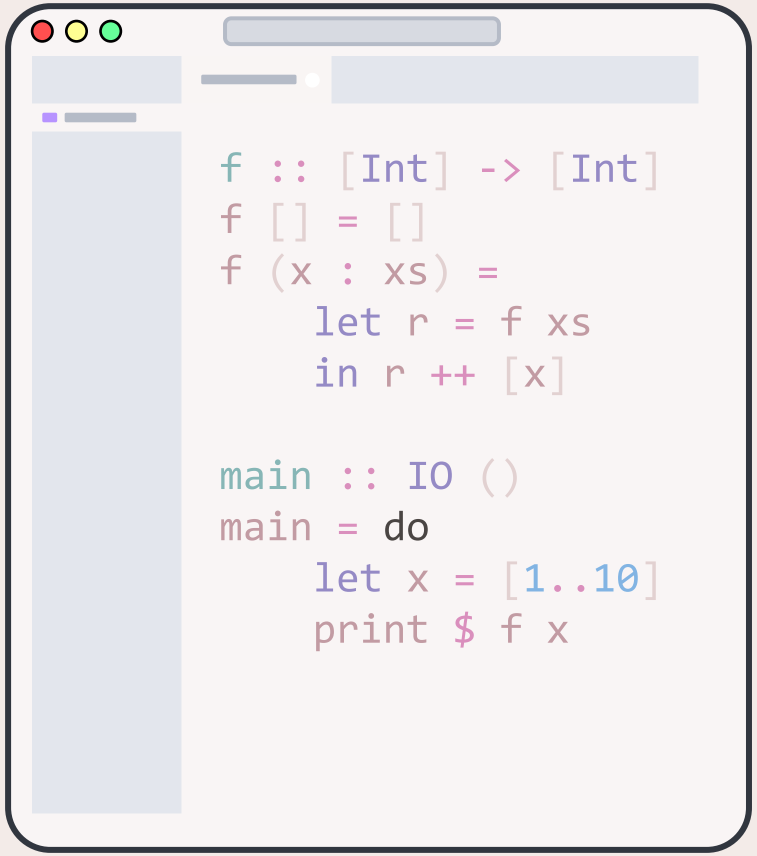
IT5100A

Industry Readiness: Typed Functional Programming

Course Introduction

Foo Yong Qi
yongqi@nus.edu.sg



Course Administration

Course Coordinator

Mr. **Foo** Yong Qi, B.Eng., M.Sc.
Instructor & Ph.D. Student

yongqi@nus.edu.sg

COM2-02-27

https://yongqi.foo/

Course Outline

- Course Introduction
- Types
- Typeclasses
- Railway Pattern
- Monads
- Concurrent Programming
- Course Conclusion

Course Outline

Course Introduction

- Course Administration
- Functional Programming
- Introduction to Haskell

Types

Typeclasses

Railway Pattern

Monads

Concurrent Programming

Course Conclusion

Course Outline

Course Introduction

Types

- Types and Type Systems
- Polymorphism
- Algebraic Data Types
- Pattern Matching

Typeclasses

Railway Pattern

Monads

Concurrent Programming

Course Conclusion

Course Outline

Course Introduction

Types

Typeclasses

- Ad-Hoc Polymorphism
- Typeclasses
- Commonly-Used Typeclasses
- Functional Dependencies
- The Existential Typeclass “Antipattern”

Railway Pattern

Monads

Concurrent Programming

Course Conclusion

Course Outline

- **Course Introduction**
- **Types**
- **Typeclasses**
- **Railway Pattern**
  - Functors
  - Applicative Functors
  - Validation
  - Monads
- **Monads**
- **Concurrent Programming**
- **Course Conclusion**

Course Outline

- **Course Introduction**
- **Types**
- **Typeclasses**
- **Railway Pattern**
- **Monads**
  - More about Monads
  - Commonly-Used Monads
  - Monad Transformers
- **Concurrent Programming**
- **Course Conclusion**

Course Outline

Graded Items

1	2	3	PE
Assignment 1 20%	Assignment 2 20%	Assignment 3 20%	Practical Exam 40%

Plagiarism Notice

No code sharing

- Assignments are on programming, standard plagiarism rules apply
- Do not share code
- All programming solutions must be entirely written by you

ChatGPT for learning only

- You are not allowed to use ChatGPT for assessments
- Asking ChatGPT to generate assignment/exam solutions is not allowed

Functional Programming

A **declarative** programming paradigm that is all about (mathematical) **functions**!

Recap: Programming Paradigms

Imperative

Procedural
Programs as series of **procedures**

Object-Oriented
Objects with data and behaviour

Declarative

Logic
Programs as sets of logical statements

Functional
Programs as composition of functions

FP Principles

- Immutability
- Pure Functions
- Recursion
- Types
- First-Class Functions

Immutability

Only use immutable data

```python
# Python
def add_one(fraction):
    """fraction is a tuple of (num, den)"""
    old_num, den = fraction
    num = old_num + den
    return (num, den)

my_fraction = (3, 2)
new_fraction = add_one(my_fraction)

print(new_fraction) # (5, 2)
print(my_fraction) # (3, 2)
```

Immutability

Only use immutable data

```python
# Python
def add_one(fraction):
    """fraction is a tuple of (num, den)"""
    old_num, den = fraction
    num = old_num + den
    return (num, den)

my_fraction = (3, 2)
new_fraction = add_one(my_fraction)

print(new_fraction) # (5, 2)
print(my_fraction) # (3, 2)
```

Nothing is mutated within the function, even the variables!

Immutability

Only use immutable data

```
# Python
def add_one(fraction):
    """fraction is a tuple of (num, den)"""
    old_num, den = fraction
    num = old_num + den
    return (num, den)

my_fraction = (3, 2)
new_fraction = add_one(my_fraction)

print(new_fraction) # (5, 2)
print(my_fraction) # (3, 2)
```

Immutable data structures are used; result of function stored as variable

Immutability

Only use immutable data

```
# Python
def add_one(fraction):
    """fraction is a tuple of (num, den)"""
    old_num, den = fraction
    num = old_num + den
    return (num, den)

my_fraction = (3, 2)
new_fraction = add_one(my_fraction)

print(new_fraction) # (5, 2)
print(my_fraction) # (3, 2)
```

All data, bindings etc. are preserved; no (unexpected) changes in state

Immutability

forces us to be **disciplined with state**

Stark contrast with programs that use mutable state

```python
# Python
def f(ls):
    ls[0] = 4
    return ls
my_ls = [1, 2, 3]
print(f(my_ls)) # [4, 2, 3]
print(my_ls) # [4, 2, 3]
```

Pure Functions

Just like functions in math, functions (in code) should be **pure**

$$f: \mathbb{N} \to \mathbb{N}$$
$$f(x) = x^2 + 2x + 3$$

Pure functions only **receive input** and **return output**

Pure Functions

Pure functions only receive input and return output

They do not **produce side effects** or **depend on external state**

Pure Functions

```python
# Python
def double(ls):
    return [i * 2 for i in ls]

x = [1, 2, 3]

print(double(x)) # [2, 4, 6]
print(double(x)) # [2, 4, 6]
print(double(x)) # [2, 4, 6]
```

Pure Functions

Function does nothing except receive parameters and return output; it is **pure**

```python
# Python
def double(ls):
    return [i * 2 for i in ls]

x = [1, 2, 3]

print(double(x)) # [2, 4, 6]
print(double(x)) # [2, 4, 6]
print(double(x)) # [2, 4, 6]
```

Pure Functions

Pure functions are **referentially transparent**: `double(x)` and `[2, 4, 6]` are the same

```python
# Python
def double(ls):
    return [i * 2 for i in ls]

x = [1, 2, 3]

print(double(x)) # [2, 4, 6]
print(double(x)) # [2, 4, 6]
print(double(x)) # [2, 4, 6]
```

Pure Functions

Pure functions are much simpler to reason about since they only do one thing

Impure functions are more frustrating to use/debug

```python
# Python
def f():
    global ls
    x = ls # use of global variable
    addend = x[-1] + 1
    x.append(addend) # is there a side-effect?
    ls = x + [addend + 1] # mutate global variable
    return ls

ls = [1, 2, 3]
x = ls

print(f()) # [1, 2, 3, 4, 5]
print(ls) # [1, 2, 3, 4, 5]
print(x) # [1, 2, 3, 4]
```

Recursion

Recursive functions simulate loops

Recursion

```python
# Python
def sum2D(ls):
    total = 0
    for row in ls:
        for num in row:
            total += num
    return total
```

Perfectly reasonable way to sum over 2D list

Loops typically useful for side-effects (mutation)—can we write same function without **any** mutation?

Recursion

Yes, compute it recursively!

```python
# Python
def row_sum(row):
    return 0 if not row else \
           row[0] + row_sum(row[1:])

def sum2D(ls):
    return 0 if not ls else \
           row_sum(ls[0]) + sum2D(ls[1:])
```

Recursion is elegant when solving problems structural-inductively

```python
@dataclass
class Tree: pass

@dataclass
class Node(Tree):
    val: object
    left: Tree
    right: Tree

@dataclass
class Leaf(Tree):
    val: object
```

```python
def preorder(tree):
    match tree:
        case Node(v, l, r):
            return [v] + preorder(l) + preorder(r)
        case Leaf(v):
            return [v]
```

Recursive functions are amenable to formal reasoning; some languages support proofs (can even be automatically synthesized)

```
-- Lean 4
 inductive Tree (α : Type) : Type where
   | node : α -> Tree α -> Tree α -> Tree α
   | leaf : α -> Tree α

-- compiler automatically synthesizes proof of termination
 def Tree.preorder { β : Type } : Tree β -> List β
   | .node v l r -> v :: (preorder l) ++ (preorder r)
   | .leaf v -> [v]

 def myTree : Tree Nat := .node 1 (.leaf 2) (.leaf 3)
 #eval myTree.preorder -- [1, 2, 3]
```

Easy to prove properties of recursive functions via **induction**

$$\frac{P(0) \qquad \forall k \in \mathbb{N}.\, P(k) \rightarrow P(k+1)}{\forall n \in \mathbb{N}.\, P(n)}$$

```
-- Lean 4
def fac : Nat -> Nat
  | 0     => 1
  | n + 1 => (n + 1) * fac n
```

Types

Adhering strictly to type information **eliminates type-related bugs** and makes functions **transparent**

Types

Adhering strictly to type information **eliminates type-related bugs** and makes functions **transparent**

Adherence to type information can be **automatically verified by a program**

Types

```python
# Python
x: int = 123
# ...
print(x + 5)
```

Forcing `x` to always be an `int` ensures that last line will never raise a `TypeError`

Types

Loose adherence to typing information deceives users of your code

```python
# Python
def safe_div(num: int, den: int) -> int:
    return None if den == 0 else \
           num // den

x = int(input())
y = int(input())
z = safe_div(x, y) + 1 # hmmm...
print(z)
```

Types

Loose adherence to typing information deceives users of your code

```python
# Python
def safe_div(num: int, den: int) -> int:
    return None if den == 0 else \
           num // den

x = int(input())
y = int(input())
z = safe_div(x, y) + 1 # hmmm...
print(z)
```

Function **doesn't** always return **`int`**

Types

Loose adherence to typing information deceives users of your code

```python
# Python
def safe_div(num: int, den: int) -> int:
    return None if den == 0 else \
           num // den


x = int(input())
y = int(input())
z = safe_div(x, y) + 1 # hmmm...
print(z)
```

Program might crash with `TypeError`!

Type system forces us to return a value of appropriate type, forcing users to handle them appropriately

```python
from dataclasses import dataclass
class Maybe:
    """Represents computation that
       may result in nothing"""
    pass
@dataclass
class Just(Maybe):
    val: int
@dataclass
class Nothing(Maybe):
    pass
```

Type system forces us to return a value of appropriate type, forcing users to handle them appropriately

```python
def safe_div(num: int, den: int) -> Maybe:
    return Nothing() if den == 0 else \
           Just(num // den)
```

Type system forces us to return a value of appropriate type, forcing users to handle them appropriately

```
x: int = int(input())
y: int = int(input())
match safe_div(x, y):
    case Just(j=j):
        print(j + 1)
```

Function purity and using correct types forces functions to be **transparent in effects**

```python
def safe_div(num: int, den: int) -> Maybe:
    return Nothing() if den == 0 else \
           Just(num // den)
```

Type systems are useful for program verification, theorem proving etc. and widely studied

```
-- Lean 4
theorem izero : ∀ (k : Nat) , k = 0 + k
  | 0 => by rfl
  | n + 1 => congrArg (. + 1) (izero n)

theorem isucc (n k : Nat) : n + k + 1 = n + 1 + k :=
  match k with
  | 0 => by rfl
  | x + 1 => congrArg (. + 1) (isucc n x)

def Vect.concat {α : Type} {n k : Nat} :
    Vect α n -> Vect α k -> Vect α (n + k)
  | .nil, ys => izero k ▸ ys
  | .cons x xs, ys => isucc _ _ ▸ .cons x (xs.concat ys)
```

First-Class Functions

Functions are **objects**; higher-order functions support **code-reuse**

```python
# Python
@dataclass(frozen=True)
class Tree:
    def map(self, f):
        match self:
            case Leaf(v):
                return Leaf(f(v))
            case Node(v, l, r):
                newval = f(v)
                newl = l.map(f)
                newr = r.map(f)
                return Node(newval, newl, newr)

@dataclass(frozen=True)
class Node(Tree):
    val: object
    left: Tree
    right: Tree

@dataclass(frozen=True)
class Leaf(Tree):
    val: object
```

Functions can receive other functions, thereby allowing us to **parameterize over behaviour**

Functions can return other functions, thereby supporting **partial function application**

```python
def add(x):
    return lambda y: x + y
```

Resulting programs are expressive, modular yet flexible

```
x = Node(1, Leaf(2), Leaf(3))
print(x.map(add(2))) # Node(3, Leaf(4), Leaf(5))
```

Languages with first-class support for functional programming make using higher-order functions easy

```haskell
-- Haskell
main :: IO ()
main = do
  let x = [1, 2, 3]
  print $ map (+2) x -- [3, 4, 5]
```

So what?

Ideas from functional programming languages are increasingly being adopted in commonly-used imperative programming languages

`|x| ->`

Closures
C++/Rust/Java 8…

`case`

Structural Pattern Matching
Python 3.11/Java 21…

`A | B`

Algebraic Data Types
Rust/Scala…

`{ }`

Records
Java 14…

FP is more than just a set of programming language features and principles…

Learning FP is about **rethinking the way we solve problems**

Goal for IT5100A

Learn to write programs in a **purely functional programming language**, and **transfer concepts** to commonly used programming languages

Haskell

Python

Things you need

- Glasgow Haskell Compiler (GHC)
- Python 3.12
- Any text editor you like (Visual Studio Code, Neovim etc.)

Introduction to Haskell

Haskell

Statically-typed, purely-functional, nonstrict evaluation
programming language

- No mutation
- No loops
- No objects
- No dynamic typing

GHCi

Open GHCi to start the interactive shell

Try entering some basic mathematical expressions!

```
ghci> 1 + 2 - 3
0

ghci> 1 * 2 / 4
0.5

ghci> 5 ^ 2 `mod` 5
0
```

GHCi

Functional programming language: (virtually) everything is a function

Use `:t` in GHCi to investigate the type of a term

Operators are functions, and can be called in the usual prefix way

Note: `f x y z` in Haskell is the same as `f(x, y, z)` in other languages

```
ghci> :t (+)
(+) :: Num a => a -> a -> a

ghci> 2 + 3
5

ghci> (+) 2 3
5
```

All functions are **curried by default**

```
ghci> y = (+2)
ghci> y 3
5
```

Haskell

```
>>> def add(x): return lambda y: x + y
>>> y = add(2)
>>> y(3)
5
```

Python

Note: `f x y z` in Haskell is the same as `f(x)(y)(z)` in other languages

Writing Programs

Programs have `.hs` extension
Every program has entry file with `main` function

```haskell
-- MyCode.hs
main :: IO () -- entry point to the program
main = putStrLn "Hello World!"
```

Compile program into executable with GHC

```
> ghc MyCode.hs
> ./MyCode
Hello World!
```

Immutability

Definitions/bindings are like assignment statements except variables are **immutable**

```haskell
-- MyCode.hs
z = 1 -- ok
y = 2 -- ok
y = 3 -- not ok!
```

Use `:l` to load file into GHCi

```
ghci> :l MyCode.hs
[1 of 2] Compiling Main ( MyCode.hs, interpreted )

MyCode.hs:3:1: error:
    Multiple declarations of ‘y’
    Declared at: MyCode.hs:2:1
                 MyCode.hs:3:1
  |
4 | y = 3 -- not ok!
  | ^
```

In Haskell you mostly write **expressions, not statements**; there are only if-else **expressions**

```
ghci> x = 2 * (-1)
ghci> y = if x == 2 then "pos" else "neg"
ghci> y
"neg"
```

Haskell

```
>>> x = 2 * -1
>>> y = 'pos' if x == 2 else 'neg'
>>> y
'neg'
```

Python

If-else expressions are **expressions** and therefore evaluate to the appropriate value

```
ghci> (if 1 /= 2 then 3 else 4) + 5
8
```

Haskell

```
>>> (3 if 1 != 2 else 4) + 5
8
```

Python

Types

Types of any expression is **fixed**

```
ghci> x = 2 * (-1)
ghci> y = if x == 2 then 2 else "neg"
<interactive>:2:20: error:
    • No instance for (Num String) arising from the literal ‘2’
    • In the expression: 2
      In the expression: if x == 2 then 2 else "neg"
      In an equation for ‘y’: y = if x == 2 then 2 else "neg"
```

Type of expression in `if` branch must be same as `else` branch

Functions

Function definitions are like any other definition

```
ghci> oddOrEven x = if even x then "even" else "odd"
ghci> oddOrEven 1
"odd"
ghci> oddOrEven 2
"even"
```

Functions

Function definitions are like any other definition

```
ghci> quadratic c2 c1 c0 x = c2 * x ^ 2 + c1 * x + c0
ghci> f = quadratic 1 2 3 -- f(x) = x^2 + 2x + 3
ghci> f 4
27
ghci> f 5
38
```

Functions

No loops in Haskell; use **recursion**

```
ghci> fac n = if n == 0 then 1 else n * fac (n - 1)
ghci> fac 4
24
```

Define if-else math-like functions with **guards**

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ f(n-1) + f(n-2) & \text{otherwise} \end{cases}$$

```
ghci> :{
ghci| fib n
ghci|     | n == 0     = 1
ghci|     | n == 1     = 1
ghci|     | otherwise = fib (n - 1) + fib (n - 2)
ghci| :}
ghci> fib 5
8
```

Even better: use **pattern matching** (we will see this soon)

```
ghci> fib 0 = 1
ghci> fib 1 = 1
ghci> fib n = fib (n - 1) + fib (n - 2)
ghci> fib 5
8
```

Compose several expressions into one with `let` bindings

```
ghci> :{
ghci| weightSum n1 w1 n2 w2 =
ghci|   let x = n1 * w1
ghci|       y = n2 * w2
ghci|   in  x + y
ghci| :}
ghci> weightSum 2 3 4 5
26
```

Haskell

```
>>> def weight_sum(n1, w1, n2, w2):
...     x = n1 * w1
...     y = n2 * w2
...     return x + y
```

Python

`let` bindings are (more-or-less) syntax sugar for function calls

```haskell
weightSum n1 w1 n2 w2 =
  let x = n1 * w1
      y = n2 * w2
  in  x + y

-- same as

weightSum n1 w1 n2 w2 =
    f (n1 * w1) (n2 * w2)
f x y = x + y
```

`let` bindings are **expressions**

```
ghci> (let x = 1 + 2 in x * 3) + 4
13
```

`where` clauses also let us define local bindings

```
weightSum n1 w1 n2 w2 =
    let x = n1 * w1
        y = n2 * w2
    in  x + y

-- same as
weightSum n1 w1 n2 w2 = x + y
    where x = n1 * w1
          y = n2 * w2
```

Data Types

Basic data types (ish): numbers, characters etc., and strings which are lists of characters

```
ghci> 1.2 -- number
1.2
ghci> 'a' -- character
'a'
ghci> "abcde" -- string (list of characters)
"abcde"
```

Lists in Haskell are singly linked lists with homogenous data

```
ghci> x = [1, 2, 3]
ghci> x !! 1 -- indexing, like x[1]
2
ghci> y = [1,3..7] -- list(range(1, 8, 2))
ghci> y
[1,3,5,7]
ghci> z = [1..10]  -- list(range(1, 11))
ghci> z
[1,2,3,4,5,6,7,8,9,10]
ghci> inflist = [1..] -- 1,2,3,...
ghci> inflist !! 10
11
```

Strings are lists of characters, we can even build ranges of characters which result in strings

```
ghci> ['h', 'e', 'l', 'l', 'o']
"hello"
ghci> ['a'..'e']
"abcde"
ghci> ['a'..'e'] ++ ['A'..'D'] -- ++ is concatentation
"abcdeABCD"
```

Build lists using **cons** (prepend) operation (**:** is right-associative)

```
ghci> x = [1, 2, 3]
ghci> 0 : x
[0,1,2,3]
ghci> 0 : 1 : 2 : 3 : []
[0,1,2,3]
ghci> 'a' : "bcde"
"abcde"
```

Building infinite lists is easy due to Haskell's lazy evaluation!

```
ghci> y = 1 : y
ghci> take 5 y
[1,1,1,1,1]
```

When performing recursion over a list, simplest approach is to split list into head element (`ls[0]`) and tail list (`ls[1:]`)

```
ghci> :{
ghci| sum' ls =
ghci|     if length ls == 0 then
ghci|         0
ghci|     else
ghci|         head ls + sum' (tail ls)
ghci| :}
ghci> sum' [1,2,3,4,5]
15
```

List comprehension also works in Haskell

```haskell
ghci> x = [1, 2, 3]
ghci> y = "abc"
ghci> [(i, j) | i <- x, j <- y, odd i]
[(1,'a'),(1,'b'),(1,'c'),(3,'a'),(3,'b'),(3,'c')]
```

Haskell

```python
>>> x = [1, 2, 3]
>>> y = 'abc'
>>> [(i, j) for i in x for j in y if i % 2 == 1]
[(1, 'a'), (1, 'b'), (1, 'c'), (3, 'a'), (3, 'b'), (3, 'c')]
```

Python

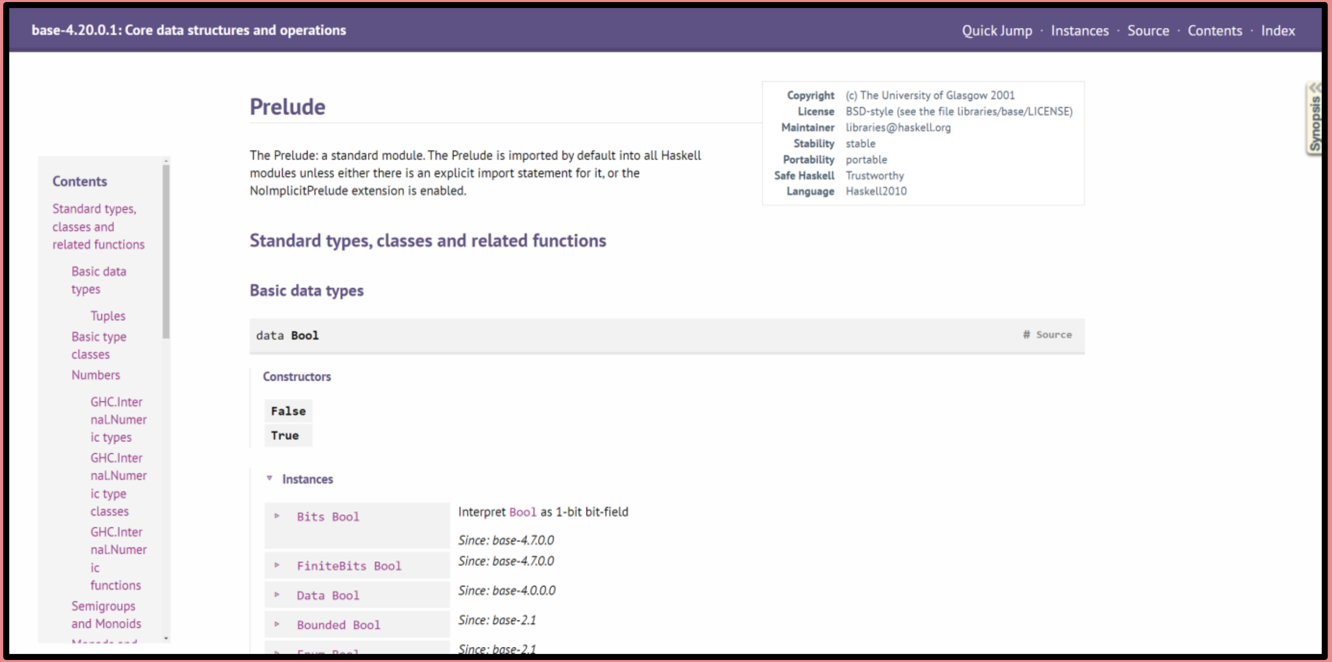
Tuples in Haskell are similar to those in Python except they are not sequences; more like products of several types

```
ghci> fst (1,"abc")
1
ghci> snd (1,(2,[3,4,5]))
(2,[3,4,5])
ghci> snd (snd (1,(2,[3,4,5])))
[3,4,5]
```

Your turn!

Try applying what you've learnt in the exercises!

Built-in functions can be found in Haskell's **Prelude**



Thank you

Foo Yong Qi

yongqi@nus.edu.sg

https://yongqi.foo/