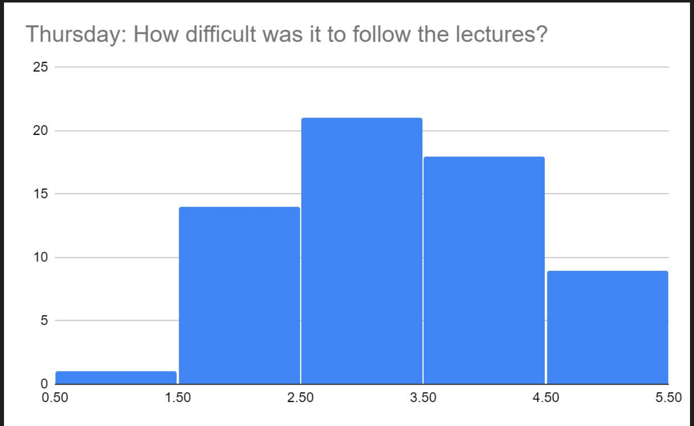
FIT2102
Programming Paradigms
Lecture 7

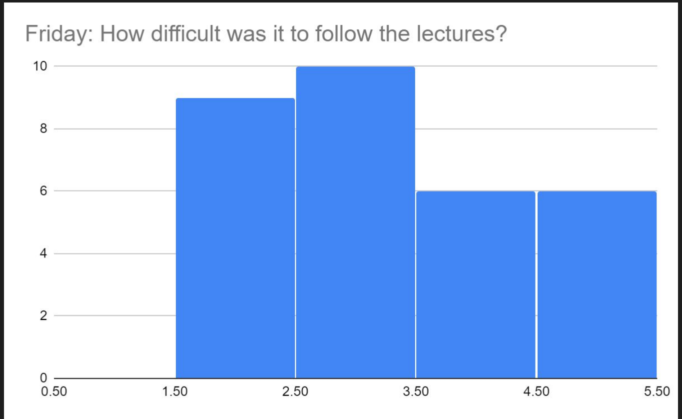
Y-Combinator
Typeclasses
Maybe

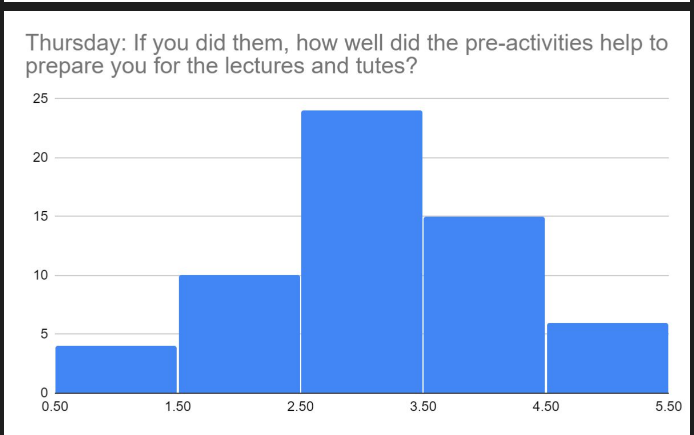
Faculty of Information Technology

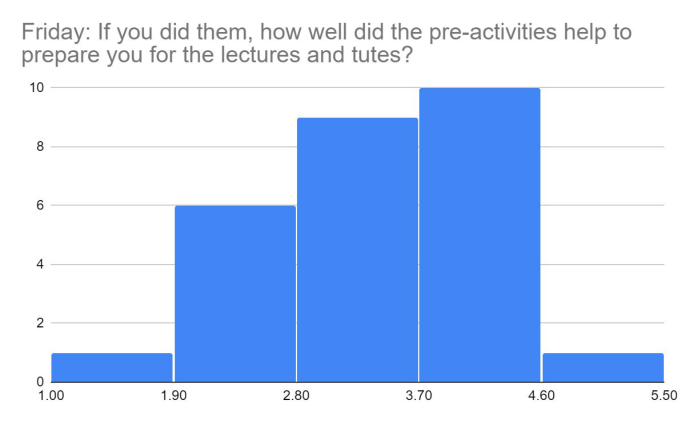
MONASH University

Informal Feedback



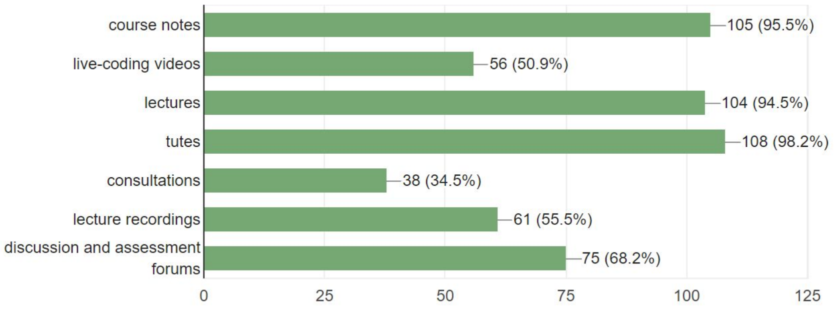






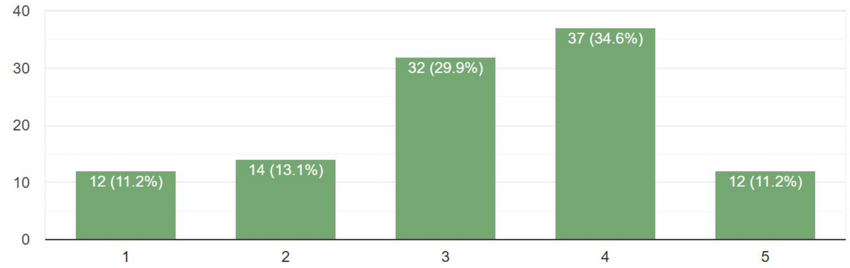
Which of the resources provided for this unit have you accessed?

110 responses



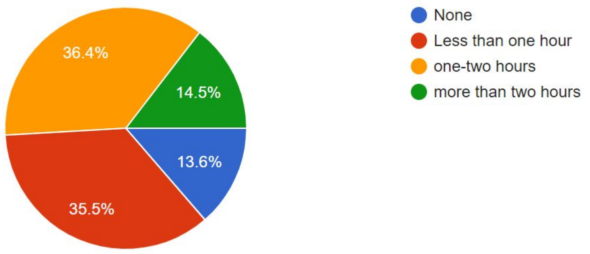
Were the in-lecture activities useful?

107 responses



How much time did you spend before the lecture working on the pre-activities prescribed each week (reading course notes, slides, watching videos)?

110 responses



Learning Outcomes

- Apply beta reduction to expressions involving the Y-Combinator to see how it is divergent and results in recurrence
- Describe how Haskell typeclasses afford polymorphism
- Create custom data types that:
    - derive typeclasses to so that standard functions can be applied to them
    - use new instances of existing typeclasses to provide custom implementations of standard functions
- Apply the Maybe data type to achieve alternative behavior when a value is empty or improperly formed

Y-Combinator

$\mathbf{Y} = \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$

$\mathbf{Y}\ g$

Y-Combinator

$\mathbf{Y} = \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$

$\mathbf{Y}\ g$

$\Rightarrow (\lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)))\ g$ <= expand Y

Y-Combinator

$\mathbf{Y} = \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$

$\mathbf{Y}\ g$

$\Rightarrow (\lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)))\ g$

$\Rightarrow (\lambda f\ [f := g].\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)))$ <= beta reduce

Y-Combinator

$\mathbf{Y} = \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$

$\mathbf{Y}\ g$

$\Rightarrow (\lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)))\ g$

$\Rightarrow (\lambda f\ [f := g].\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)))$ <= beta reduce

$\Rightarrow \lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)$ $[f := g]$

Y-Combinator

$\mathbf{Y} = \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$

$\mathbf{Y}\ g$

$\Rightarrow (\lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)))\ g$

$\Rightarrow (\lambda f\ [f := g].\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)))$ <= beta reduce

$\Rightarrow (\lambda x.\ g\ (x\ x))\ (\lambda x.\ g\ (x\ x))$

Y-Combinator

$\mathbf{Y} = \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$

$\mathbf{Y}\ g$

$\Rightarrow (\lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)))\ g$

$\Rightarrow (\lambda f\ [f := g].\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)))$

$\Rightarrow (\lambda x.\ g\ (x\ x))\ (\lambda x.\ g\ (x\ x))$

$\Rightarrow (\lambda x\ [x := (\lambda x.\ g\ (x\ x))].\ g\ (x\ x))$ <= beta reduce

Y-Combinator

$\mathbf{Y} = \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$

$\mathbf{Y}\ g$

$\Rightarrow (\lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)))\ g$

$\Rightarrow (\lambda f\ [f := g].\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)))$

$\Rightarrow (\lambda x.\ g\ (x\ x))\ (\lambda x.\ g\ (x\ x))$

$\Rightarrow (\lambda x\ [x := (\lambda x.\ g\ (x\ x))].\ g\ (x\ x))$ <= beta reduce

$\Rightarrow g\ ((\lambda x.\ g\ (x\ x))\ (\lambda x.\ g\ (x\ x)))$

Y-Combinator

$\mathbf{Y} = \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$

$\mathbf{Y}\ g$

$\Rightarrow (\lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)))\ g$

$\Rightarrow (\lambda f\ [f := g].\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)))$

$\Rightarrow (\lambda x.\ g\ (x\ x))\ (\lambda x.\ g\ (x\ x))$

$\Rightarrow (\lambda x\ [\mathrm{x} := (\lambda x.\ g\ (x\ x))].\ g\ (x\ x))$

$\Rightarrow g\ ((\lambda x.\ g\ (x\ x))\ (\lambda x.\ g\ (x\ x)))$ <= alpha equivalent to: Y g

Y-Combinator

$\mathbf{Y} = \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$

$\mathbf{Y}\ g$

$\Rightarrow (\lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)))\ g$

$\Rightarrow (\lambda f\ [f := g].\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)))$

$\Rightarrow (\lambda x.\ g\ (x\ x))\ (\lambda x.\ g\ (x\ x))$

$\Rightarrow (\lambda x\ [\mathrm{x} := (\lambda x.\ g\ (x\ x))].\ g\ (x\ x))$

$\Rightarrow g\ ((\lambda x.\ g\ (x\ x))\ (\lambda x.\ g\ (x\ x)))$ <= alpha equivalent to: Y g

$\Rightarrow g\ (\mathbf{Y}\ g)$

Y-Combinator

$\mathbf{Y} = \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$

$\mathbf{Y}\ g$

$\Rightarrow (\lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)))\ g$

$\Rightarrow (\lambda f\ [f := g].\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)))$

$\Rightarrow (\lambda x.\ g\ (x\ x))\ (\lambda x.\ g\ (x\ x))$

$\Rightarrow (\lambda x\ [\mathrm{x} := (\lambda x.\ g\ (x\ x))].\ g\ (x\ x))$

$\Rightarrow g\ ((\lambda x.\ g\ (x\ x))\ (\lambda x.\ g\ (x\ x)))$

$\Rightarrow g\ (\mathbf{Y}\ g)$

$\Rightarrow g\ g\ (\mathbf{Y}\ g)$

$\Rightarrow g\ g\ g\ (\mathbf{Y}\ g)$

...

Y-Combinator meets JavaScript *- Not examinable*

$$\mathbf{Y} = \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$$

```
const Y = f=> (x => f(x)(x))(x=> f(x)(x)) // Direct translation from Lambda Calc

// A simple function that recursively calculates 'n!'.
const FAC = next => n => n>1 ? n * next(n-1) : 1
const fac = Y(FAC)
console.log(fac(5))

… stack overflow
```

Strict Evaluation

- Not examinable

$\lambda f[\, f := \mathtt{FAC}\ ].\ (\lambda x.\ f\,(\,x\ x\,)\,)\ (\lambda x.\ f\,(\,x\ x\,)\,)$

$\lambda x[x{:=}(\lambda x.\ \mathtt{FAC}(\,x\ x\,)\,)].\ \mathtt{FAC}(\,x\ x\,)$ <= beta reduction

$\mathtt{FAC}\ (\ (\lambda x.\ \mathtt{FAC}\ (\,x\ x\,)\,)\ (\lambda x.\ \mathtt{FAC}\ (\,x\ x\,)\,)\ )$ <= need to evaluate args before calling function

$\mathtt{FAC}\ (\ \lambda x[\ x{:=}\ (\lambda x.\ \mathtt{FAC}(\,x\ x\,)\,)\ ].\ \mathtt{FAC}\ (\,x\ x\,)\ )$ <= beta reduction again...

$\mathtt{FAC}\ (\ \mathtt{FAC}\ (\ (\lambda x.\ \mathtt{FAC}\ (\,x\ x\,)\,)\ (\lambda x.\ \mathtt{FAC}\ (\,x\ x\,)\,)\ )\ )$

$\mathtt{FAC}\ (\ \mathtt{FAC}\ (\ (\ \lambda x[\ x{:=}\ (\lambda x.\ \mathtt{FAC}\ (\,x\ x\,)\,)\ ].\ \mathtt{FAC}\ (\,x\ x\,)\ )\ )\ )$ <= beta reduction again...

$\mathtt{FAC}\ (\ \mathtt{FAC}\ (\ \mathtt{FAC}\ (\ (\lambda x.\ \mathtt{FAC}\ (\,x\ x\,)\,)\ (\lambda x.\ \mathtt{FAC}\ (\,x\ x\,)\,)\ )\ )\ )$ <= and so on...

forever...

Strict Y-Combinator

- Not examinable

```
FAC = next => n => n>1 ? n * next(n-1) : 1
```

Y = λf [f := FAC]. (λx. f (λv. x x v)) (λx. f (λv. x x v)) <= beta reduction

(λx. FAC (λv. x x v)) (λx. FAC (λv. x x v))

Λx [x:=(λx. FAC (λv. x x v)]. FAC (λv. x x v)) <= beta reduction

FAC (λv. (λx. FAC (λv. x x v)) (λx. FAC (λv. x x v)) v) <= at this point, FAC actually gets called...

```
const Y = f => ( x => f(v => x(x)(v)) )( x => f(v => x(x)(v)) )
const fac = Y(FAC)
console.log(fac(5))
> 120
```

Lambda functions in Haskell

A lambda function in haskell looks like: `\x -> <some expression of x>`

Compare to a lambda in JavaScript: `x => <some expression of x>`

and lambda calculus: `λx . <some expression of x>`

```
> map (\x->2*x) [1..4]
[2,4,6,8]
```

...but often we can avoid explicit lambdas with partially applied functions (to achieve a point-free style):

```
> map (2*) [1..4]
[2,4,6,8]
```

The Y-Combinator in Haskell - *Not examinable*

It's possible to evaluate the lazy version of the Y-Combinator in Haskell
(but we have to disable type checking):

```haskell
import Unsafe.Coerce
y :: (a -> a) -> a
y = \f -> (\x -> f (unsafeCoerce x x)) (\x -> f (unsafeCoerce x x))

fac next n
  | n > 1 = n * (next (n-1))
  | otherwise = 1
GHCi> (y fac) 5
120
```

There are other versions of the Y-Combinator that do type check in Haskell

```haskell
GHCi> y ("circular reasoning works because " ++)
"circular reasoning works because circular reasoning works because circular reasoning works because ...
```

Recap: declaring data types in Haskell

```haskell
data IntPair = IntPair Int Int
data IntPair = IntPair { first::Int, second::Int } -- record syntax
```

```typescript
// typescript
type Pair = { first: number, second: number}
```

Recap: declaring data types in Haskell

```haskell
data IntPair = IntPair Int Int
data IntPair = IntPair { first::Int, second::Int }


p = IntPair 1 2
> first p
1

plusPair :: IntPair -> Int
plusPair (IntPair  a b) = a + b

> plusPair p
3
```

Parametric Polymorphism in Haskell

```haskell
data PairOfA a = APair a a
data PairOfA a = APair {
    first::a, second::a }
```

```typescript
// typescript
type PairOfT<T> = {
        first: T, second: T }
```

```
ghci> data PairOfA a = APair a a deriving Show
ghci> APair 23 48
APair 23 48
ghci> APair "hello" "tim"
APair "hello" "tim"
ghci> APair "hello" 48

<interactive>:16:25: error:
    * No instance for (Num [Char]) arising from the literal `48'
    ...
```

Polymorphism in Haskell

```haskell
data PairOfInt = PairOfInt { fst::Int, sec::Int }

data PairOfA a = PairOfA { fst::a, sec::a }

data Pair a b = Pair { fst::a, sec::b }
```

```typescript
// typescript
type PairOfT<T> = {fst: T, sec: T}

type Pair<U, V> = {fst: U, sec: V}
```

```
ghci> :kind PairOfInt
PairOfInt :: *
ghci> :kind PairOfA
Int :: * -> *
ghci> :kind Pair
Pair :: * -> * -> *
```

⇦ Constructor returns a type

⇦ Constructor takes one type parameter returns a type

⇦ Constructor takes two type parameters and returns a type

The *kind* of a type is like a little lambda calculus to describe the arity of its constructor's type parameters

'*' represents any concrete type

Polymorphism in Haskell

```haskell
data Pair a = Pair a a
data Pair a = Pair { first::a, second::a }

p = Pair 1 2
> first p
1
>:t p
p :: Pair Integer

plusPair (Pair a b) = a + b
> :t plusPair
plusPair :: Num a => Pair a -> a
```

Polymorphism in Haskell

Typeclass constraint

Input parameter type

Return type

```
plusPair :: Num a => Pair a -> a
plusPair (Pair a b) = a + b
```

Typeclasses define a set of functions that can have different implementations depending on the type of data they are given. (Real World Haskell)

Using typeclasses with custom data types

```haskell
data WeekDay = Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

```
> Mon == Wed
<interactive>:1:1: error:
    * No instance for (Eq WeekDay) arising from a use of `=='
    * In the expression: Mon == Wed
      In an equation for `it': it = Mon == Wed
```

```
> :i Eq
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

Using typeclasses with custom data types

```haskell
data WeekDay = Mon | Tue | Wed | Thu | Fri | Sat | Sun
  deriving Eq
```

```
> Mon == Wed
False
> Wed == Wed
True
```

Using typeclasses with custom data types

```haskell
data WeekDay = Mon | Tue | Wed | Thu | Fri | Sat | Sun
  deriving Eq
```

```
> print Mon

<interactive>:1:1: error:
    * No instance for (Show WeekDay) arising from a use of `print'
    * In the expression: print Mon
      In an equation for `it': it = print Mon
```

Using typeclasses with custom data types

```haskell
data WeekDay = Mon | Tue | Wed | Thu | Fri | Sat | Sun
  deriving (Eq, Show)
```

```
> print Mon
Mon
```

Custom instances of typeclasses

```haskell
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun

instance Show Day where
    show Sat = "Sleep in"
    show Sun = "Oh no it's nearly Monday"
    show _ = "Sigh"
```

```
> print Mon
"Sigh"
```

Exercise 1:

… to be announced

Ord typeclass

```
GHCi> :i compare
class Eq a => Ord a where
  compare :: a -> a -> Ordering

GHCi> :i Ordering
data Ordering = LT | EQ | GT
```

Ord typeclass

```haskell
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
  deriving (Eq, Ord, Show)

week = [ Thu, Mon, Sun, Wed, Tue, Fri, Sat ]
```

```
> sort week
[Mon,Tue,Wed,Thu,Fri,Sat,Sun]

> sort [(12, "Sally"), (7, "Sam"), (7, "Alice")]
[(7,"Alice"),(7,"Sam"),(12,"Sally")]
```

Haskell already has an instance of Ord for Tuples

Ord typeclass

```
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
 deriving (Eq, Show)

instance Ord Day where
   compare Mon Tue = LT
   compare Tue Wed = LT
   compare Wed Thu = LT

> Mon < Tue
True
> Mon < Wed
*** Exception: src\DaysOfTheWeek.hs:(7,5)-(9,24):
    Non-exhaustive patterns in function compare
```

```haskell
instance Ord Day where
    compare Mon Tue = LT
    compare Mon Wed = LT
    compare Mon Thu = LT
    compare Mon Fri = LT
    compare Mon Sat = LT
    compare Mon Sun = LT
    compare Tue Wed = LT
    compare Tue Thu = LT
    compare Tue Fri = LT
    compare Tue Sat = LT
    compare Tue Sun = LT
    compare Wed Thu = LT
    compare Wed Fri = LT
    compare Wed Sat = LT
    compare Wed Sun = LT
    compare Thu Fri = LT
    compare Thu Sat = LT
    compare Thu Sun = LT
    compare Fri Sat = LT
    compare Fri Sun = LT
    compare Sat Sun = LT
    compare a b
      | b == a = EQ
      | otherwise = GT
```

A custom instance of Ord has to fully specify all possible comparisons

Exercise 2:

… to be announced

Some Typeclasses Used for Numbers

Eq

```
(==) (/=)
```

Ord

```
(<)  (>)
(<=) (>=)
min  max
```

Num

```
(+)(-)(*)
negate  abs
signum
fromInteger
```

Enum

```
succ pred toEnum fromEnum
enumFrom       [n..]
enumFromThen   [n,s,..]
enumFromTo     [n..m]
enumFromThenTo [n,s..m]
```

Real

```
toRational
```

Fractional

```
(/)  recip
fromRational
```

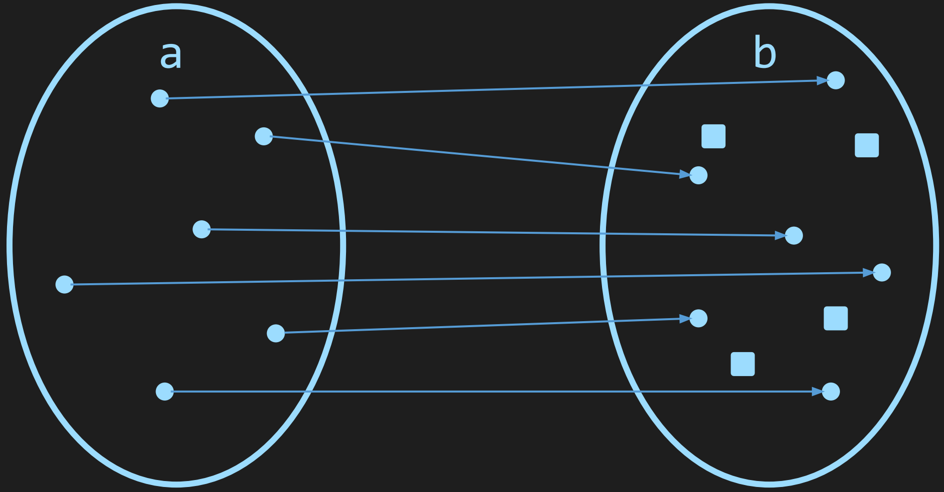
Integral

```
div  mod
quot rem
quotRem divMod
toInteger
```

Total Function

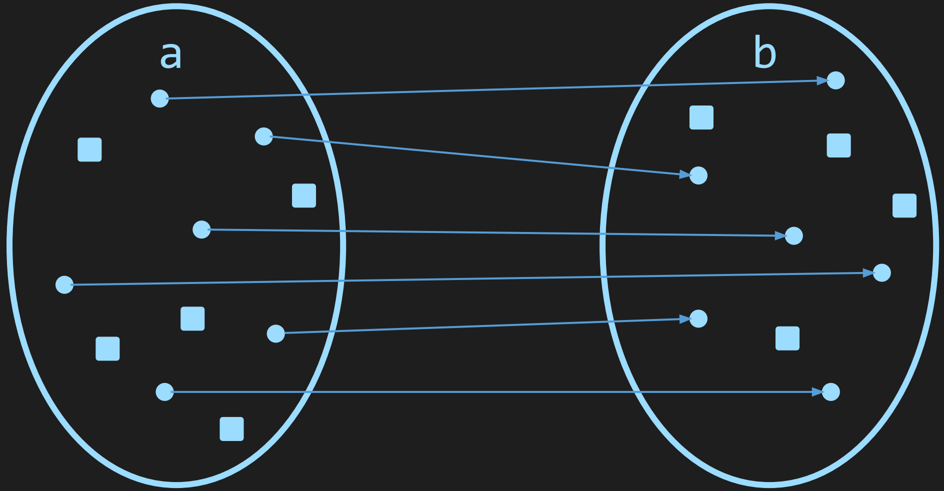
`f :: a -> b`

Everything in a is mapped by f to a value in b



Partial Function

`f :: a -> b`



`f` is partial because it is undefined for some inputs

Maybe

```haskell
data Maybe a = Just a | Nothing
    deriving (Eq, Ord)

phonebook :: [(String, String)]
phonebook = [ ("Bob",   "01788 665242"), ("Fred",  "01624 556442"), ("Alice", "01889 985333") ]

> :t lookup
lookup :: Eq a => a -> [(a, b)] -> Maybe b

> lookup "Fred" phonebook
Just "01624 556442"

> lookup "Tim" phonebook
Nothing
```

lookup is a *partial function*
A partial function is not defined over all the elements of its input set (domain)

We can pattern match Just a or Nothing to give default behaviour

Pattern matching Maybes

```haskell
printNumber name = msg $ lookup name phonebook
   where
       msg (Just number)   = print number
       msg Nothing         = print $ name ++ " not found in database"
```

```
*GHCi> printNumber "Fred"

"01624 556442"

*GHCi> printNumber "Tim"

"Tim not found in database"
```

Conclusions

- The Y-Combinator is an important theoretical concept that enables the Lambda Calculus to be Turing Complete
- Typeclasses give Haskell flexible polymorphism
- Maybe is a datatype that enables alternative behaviour for *partial functions*