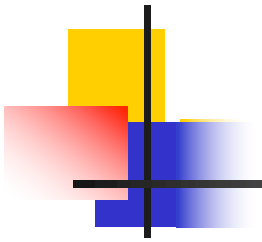
Intro to Haskell: Types and Functions

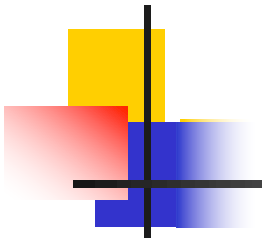


So Far

- Essential functional programming concepts
  - Reduction semantics
  - Lists and recursion
  - Higher-order functions
    - Map and fold (also known as reduce)
  - Scoping
  - Evaluation order
- Scheme
- Lambda calculus --- theoretical foundation

Applicative

Normal



Coming Up: Haskell

- Haskell: a functional programming language
  - Rich syntax (syntactic sugar), rich modules
  - Static typing and type inference
  - Lazy evaluation

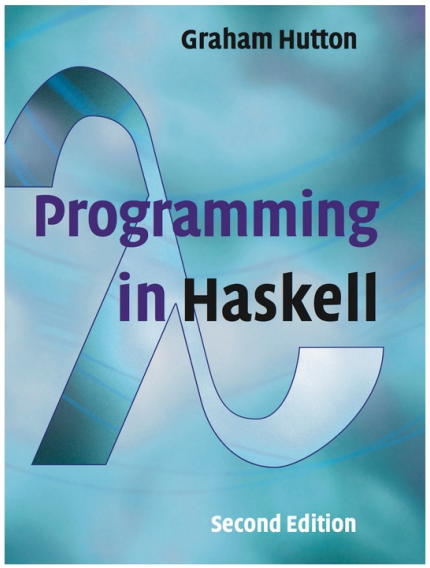
  - Algebraic data types and pattern matching
  - Monads
  - And more…

Haskell Resources

- Resources
  - **Programming in Haskell,**

    Second Edition by Graham Hutton

  - https://www.haskell.org/



Haskell

An advanced, purely functional programming language

Getting Started

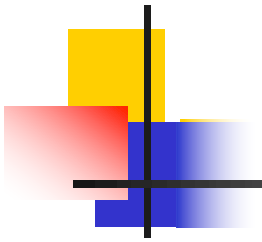
- Download the Glasgow Haskell Compiler:
  - https://www.haskell.org/ghc
- Run Haskell in interactive mode:
  - **ghci**
  - Type functions in a file (e.g., **fun.hs**), then load the file and call functions interactively

Prelude > :l fun.hs

[1 of 1] Compiling Main (fun.hs, interpreted)

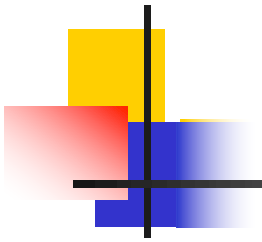
Ok, one module loaded.

***Main > square 25**



Outline

- *Basic types*
- *Lists and tuples*
- *Function types and currying*
- *Type classes (a little bit)*
- *Defining functions*
  - *Pattern matching*
  - *Guarded sections*
  - *Lambda expressions*
- *Recursive functions*



What is a Type?

A type is a name for a collection of related values. For example, in Haskell the basic type

```
Bool
```

contains the two logical values:

```
False
```

```
True
```

Type Errors

Applying a function to one or more arguments of the wrong type is called a type error.

Haskell:

```
> 1 + False
Error…
```

STATIC

Scheme:

```
>(+ 1 #t)
```

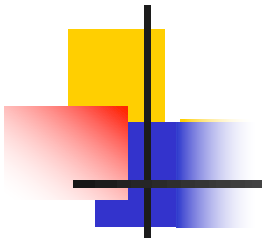
DYNAMIC

1 is a number and False is a logical value, but + requires two numbers.

More on Type Error Messages

```
> 1 + False

<interactive>:1:1: error:
    • No instance for (Num Bool) arising from a use of ‘+’
    • In the expression: 1 + False
      In an equation for ‘it’: it = 1 + False
```

```
> ‘a’ ++ “na”
  Char      String = [Char]
<interactive>:2:1: error:
    • Couldn't match expected type ‘[Char]’ with actual type ‘Char’
    • In the first argument of ‘(++)’, namely ‘'a'’
      In the expression: 'a' ++ "na"
      In an equation for ‘it’: it = 'a' ++ "na"
```



Types in Haskell

- If evaluating an expression **e** would produce a value of type **t**, then **e** has type **t**, written

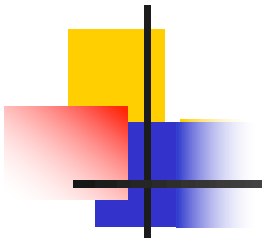
```
e :: t
```

- Every well-formed expression has a type, which can be **automatically calculated at compile time** using a process called type inference

- All type errors are found at compile time, which makes programs safer and faster by removing the need for type checks at run time

- In GHCi, the :type (or just :t) command calculates the type of an expression, without evaluating it:

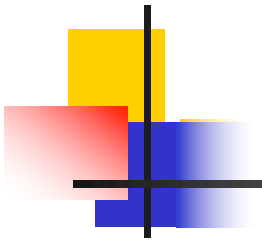
```
> not False
True

> :type not False
not False :: Bool
```



Basic Types

Haskell has a number of basic types, including:

- `Bool` - logical values
- `Char` - single characters
- `String` - strings of characters [Char]
- `Int` - integer numbers
- `Float` - single-precision floating-point
- `Double` - double-precision floating-point



List Types

A list is sequence of values of the same type:

```
[False,True,False] :: [Bool]

['a','b','c','d'] :: [Char]
```

In general:

[a] is the type of lists with elements of type **a**

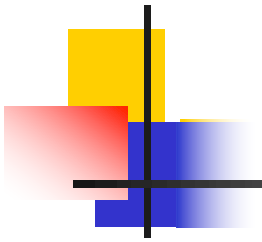
Note:

- The type of a list says nothing about its length:

```
[False,True] :: [Bool]

[False,True,False] :: [Bool]
```

- The type of the elements is unrestricted. For example, we can have lists of lists:

```
[['a'],['b','c']] :: [[Char]]
```



Tuple Types

A tuple is a sequence of values of different types:

```
(False,True) :: (Bool,Bool)

(False,'a',True) :: (Bool,Char,Bool)
```

In general:

(t1,t2,…,tn) is the type of n-tuples whose i-th components have type **ti** for any **i** in **1…n**

Note:

- The type of a tuple encodes its size:

```
(False,True) :: (Bool,Bool)

(False,True,False) :: (Bool,Bool,Bool)
```

- The type of the components is unrestricted:

```
('a',(False,'b')) :: (Char,(Bool,Char))

(True,['a','b']) :: (Bool,[Char])
```

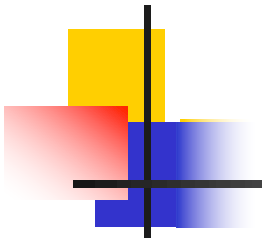
Exercises

```
>:t [False,True,False]
>?
>:t (False,True,1)
>?

>maxBound::Int
>?
>:i maxBound
>?

>:t [False,True,1]
>?
```

[Bool]

(Bool, Bool, Int)

ERROR



Function Types

A function is a mapping from values of one type to values of another type:

```
not :: Bool → Bool

even :: Int → Bool
```

In general:

t1 → t2 is the type of functions that map values of type **t1** to values to type **t2**

Note:

- The arrow → is typed at the keyboard as ->.

- The argument and result types are unrestricted. For example, functions with multiple arguments or results are possible using lists or tuples:

TYPE: →
DEFINITION: →

```
add :: (Int,Int) → Int
add (x,y) = x + y

zeroto :: Int → [Int]
zeroto n = [0..n]
```

Curried Functions

Functions with multiple arguments are also possible by returning functions as results:

x y result

```
add' :: Int → (Int → Int)
add' x y = x + y
```

add' takes an integer x and returns a function add' x. In turn, this function takes an integer y and returns the result x+y.

Note:

- add and add' produce the same final result, but add takes its two arguments at the same time, whereas add' takes them one at a time:

```
add :: (Int,Int) → Int

add' :: Int → (Int → Int)
```

- Functions that take their arguments one at a time are called curried functions, celebrating the work of Haskell Curry on such functions.

- Functions with more than two arguments can be curried by returning nested functions:

```
mult :: Int → (Int → (Int → Int))
mult x y z = x * y * z
```

x y z result

mult5 :: Int → (Int → Int)

mult5 = mult 5

mult takes an integer x and returns a function mult x, which in turn takes an integer y and returns a function mult x y, which finally takes an integer z and returns the result x*y*z.

Why is Currying Useful?

Curried functions are more flexible than functions on tuples, because useful functions can often be made by partially applying a curried function

For example:

```
add' 1 :: Int → Int

take 5 :: [Int] → [Int]

drop 5 :: [Int] → [Int]
```

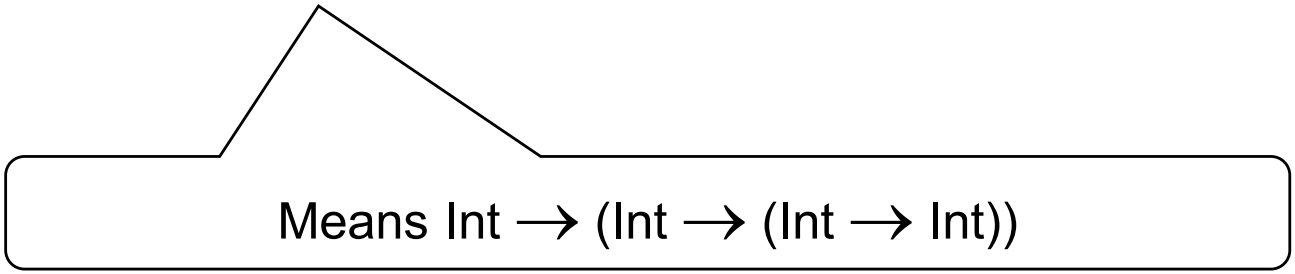
take :: Int → [a] → [a]
>take 2 [1,2,3,4,5]
[1,2]

Currying Conventions

To avoid excess parentheses when using curried functions, two simple conventions are adopted:

- The arrow → associates to the right

```
Int → Int → Int → Int
```

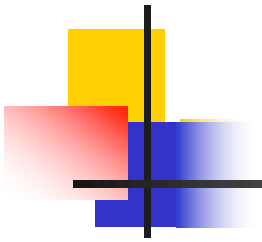


- As a consequence, it is then natural for function application to associate to the left.

```
((mult x) y) z
```

Means ((mult x) y) z

All functions in Haskell are curried.



Polymorphic Functions

A function is called polymorphic ("of many forms") if its type contains one or more type variables

```
length :: [a] → Int
```

For any type a, length takes a list of values of type a and returns an integer.

Note:

- Type variables can be instantiated to different types in different circumstances:

```
> length [False,True]
2

> length [1,2,3,4]
4
```

a = Bool

a = Int

- Type variables must begin with a lower-case letter, and are usually named a, b, c, etc.

Exercises

- Many of the functions defined in the standard prelude are polymorphic. For example:

```
head :: ? [a] → a

take :: ? Int → [a] → [a]

zip :: ? [a] → [b] → [(a,b)]      zipWith
(E.g., zip [1,2] [3,4] = [(1,3),(2,4)]

id :: ? a → a          id x = x      λx.x

fst :: ? (a,b) → a
```

Overloaded Functions

A polymorphic function is called overloaded if its type contains one or more class constraints

type classes

```
(+) :: Num a ⇒ a → a → a
```

A type class constraint

For any numeric type a, (+) takes two values of type a and returns a value of type a

Int, Float, Double are istances of type class Num

Note:

- Constrained type variables can be instantiated to any types that satisfy the constraints:

```
> 1 + 2
3

> 1.0 + 2.0
3.0

> 'a' + 'b'
ERROR
```

a = Int

a = Float

Char is not a numeric type

More Type Errors

```
> 'a' + 'b'

<interactive>:29:1: error:
    • No instance for (Num Char) arising from a use of ‘+’
    • In the expression: 'a' + 'b'
      In an equation for ‘it’: it = 'a' + 'b'
```

- Haskell has a number of type classes, including:

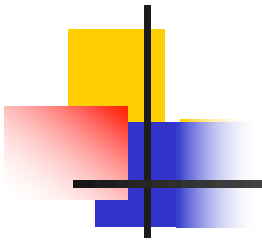
  `Num` - Numeric types

  `Eq` - Equality types

  `Ord` - Ordered types

- For example:

```
(+)  :: Num a ⇒ a → a → a

(==) :: Eq a  ⇒ a → a → Bool

(<)  :: Ord a ⇒ a → a → Bool
```



Hints and Tips

- When defining a new function in Haskell, it is useful to begin by writing down its type
  - We'll ask that you always add the type!
- When stating the types of polymorphic functions that use numbers, equality or orderings, include the necessary class constraints
  - If you write a type that doesn't, Haskell will complain

More on Type Errors

Num a =>

```
add :: a -> a -> a
add x y = x + y
```

Num a => a -> a -> a

et add

```
> :l add.hs
add.hs:83:11: error:
    • No instance for (Num a) arising from a use of ‘+’
      Possible fix:
        add (Num a) to the context of
          the type signature for:
            add :: forall a. a -> a -> a
    • In the expression: x + y
      In an equation for ‘add’: add x y = x + y
```

Exercises

(1) What are the types of the following values?

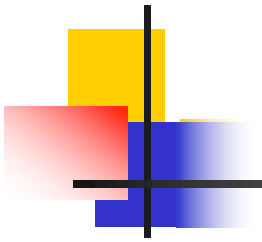
```
['a','b','c']
('a','b','c')
[(False,'0'),(True,'1')]
([False,True],['0','1'])
[tail,init,reverse]
```

```
>:t ['a','b','c']
[Char]
>:t ('a','b','c')
(Char, Char, Char)
>:t [(False,'0'),(True,'1')]
[(Bool, Char)]
>:t ([False,True],['0','1'])
([Bool], [Char])
>:t [tail,init,reverse]
[ [a]→[a] ]
```

Exercises

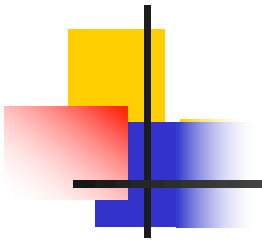
(2) What are the types of the following functions?

```
second :: [a] -> a
second xs = head (tail xs)

swap :: (a,b) -> (b,a)
swap (x,y) = (y,x)

pair :: a -> b -> (a,b)
pair x y = (x,y)

double :: Num a => a -> a
double x = x*2

palindrome :: Eq a => [a] -> Bool
palindrome xs = reverse xs == xs

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



Outline

- *Basic types*
- *Lists and tuples*
- *Function types and currying*
- *Type classes*
- *Defining functions*
  - *Pattern matching*
  - *Guarded equations*
  - *Lambda expressions*
- *Recursive functions*



Conditional Expressions

As in most programming languages, functions can be defined using conditional expressions

```
abs :: Int → Int
abs n = if n ≥ 0 then n else -n
```

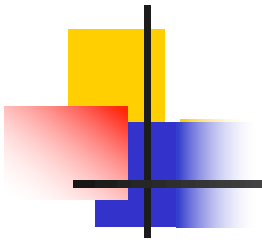
abs takes an integer n and returns n if it is non-negative and -n otherwise.

Conditional expressions can be nested:

```
signum :: Int → Int
signum n = if n < 0 then -1 else
               if n == 0 then 0 else 1
```

Note:

- In Haskell, conditional expressions must always have an else branch, which avoids any possible ambiguity problems with nested conditionals.



Guarded Equations

As an alternative to conditionals, functions can also be defined using guarded equations.

```
abs n | n ≥ 0     = n
      | otherwise = -n
```

```
abs n
  | n ≥ 0     = n
  | otherwise = -n
```

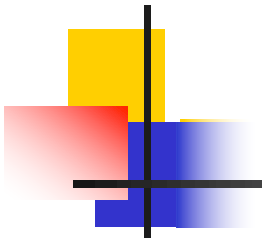
As previously, but using guarded equations.

Guarded equations can be used to make definitions involving multiple conditions easier to read:

```
signum n
     | n < 0      = -1
     | n == 0     = 0
     | otherwise = 1
```

Note:

- The catch all condition otherwise is defined in the prelude by otherwise = True.



Pattern Matching

Many functions have a particularly clear definition using pattern matching on their arguments.

```
not :: Bool → Bool
not False = True
not True  = False
```

not maps False to True, and True to False.

Functions can often be defined in many different ways using pattern matching. For example:

```
(&&) :: Bool → Bool → Bool
True  && True  = True
True  && False = False
False && True  = False
False && False = False
```

can be defined more compactly by

```
True && True = True
_    && _    = False
```

However, the following definition is more efficient, because it avoids evaluating the second argument if the first argument is False:

```
True  && b = b
False && _ = False
```

Note:

- The underscore symbol _ is a wildcard pattern that matches any argument value.

- Patterns are matched in order. What happens if we changed the order of patterns?

```
_    && _    = False
True && True = True
```

- Patterns may not repeat variables. For example, the following definition gives an error:

```
b && b = b
_ && _ = False
```

Exercises

Define **`safediv n d`** to compute n `div` d but safely. If d is 0, **`safediv`** returns [], otherwise it returns the singleton list containing the result. E.g., **`safediv 10 0 = []`** and **`safediv 6 2 = [3]`.**

a) conditional expression

```
safediv :: Int -> Int -> [Int]
safediv n d = if d==0 then [] else [n `div` d]
```

b) guarded sections

```
safediv n d
  | d == 0    = []
  | otherwise = [n `div` d]
```

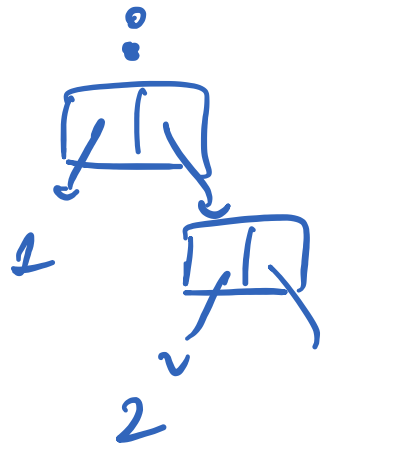
c) pattern matching

```
safediv _ 0 = []
safediv n d = [n `div` d]
```

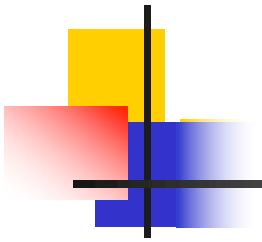
List Patterns

Internally, every non-empty list is constructed by repeated use of an operator (:) called "cons" that adds an element to the start of a list.

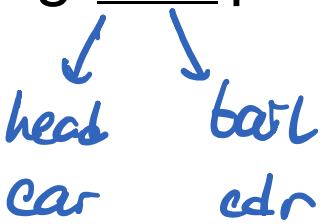
```
[1,2,3,4]
```



Means 1:(2:(3:(4:[]))).



Functions on lists can be defined using x:xs patterns.




```
head :: [a] → a
head (x:_) = x

tail :: [a] → [a]
tail (_:xs) = xs
```

head and tail map any non-empty list to its first and remaining elements.

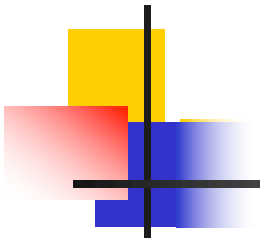
Note:

- x:xs patterns only match non-empty lists:

```
> head []
*** Exception: empty list
```

- x:xs patterns must be parenthesised, because application has priority over (:). For example, the following definition gives an error:

```
head (x:_) = x
```



Lambda Expressions

Functions can be constructed without naming the functions by using lambda expressions.

```
λx → x + x
```

```
\x -> x + x
```

Same as Scheme's

```
(lambda (x) (+ x x))
```

the nameless function that takes a number x and returns the result x + x.

Why Are Lambda's Useful?

Lambda expressions can be used to avoid naming functions that are only referenced once. One-off functions.

where is syntactic sugar for let

For example:

```
odds n = map f [0..n-1]
         where
           f x = x * 2 + 1
```

Same as:

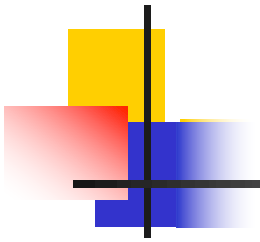
```
odds n =
  let f x = x*2+1 in
    map f [0..n-1]
```

simplifies to:

```
odds n = map (λx → x * 2 + 1) [0..n-1]
```

Rendered this way in Haskell:

```
odds n = map (\x -> x * 2 + 1) [0..n-1]
```



Operator Sections

An operator written between its two arguments can be converted into a curried function written before its two arguments by using parentheses.

For example:

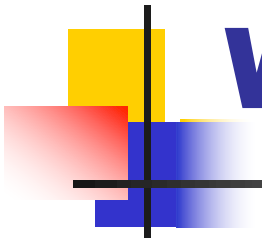
```
> 1 + 2
3

> (+) 1 2
3
```

This convention also allows one of the arguments of the operator to be included in the parentheses.

For example:

```
> (1+) 2
3

> (+2) 1
3
```

In general, if ⊕ is an operator then functions of the form (⊕), (x⊕) and (⊕y) are called sections.



Why Are Sections Useful?

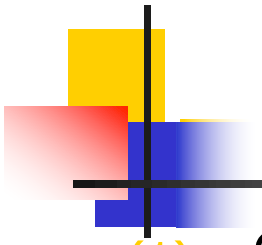
Useful functions can sometimes be constructed in a simple way using sections. For example:

```
(1+)
```
- successor function

```
(1/)
```
- reciprocation function

```
(*2)
```
- doubling function

```
(/2)
```
- halving function



Exercises

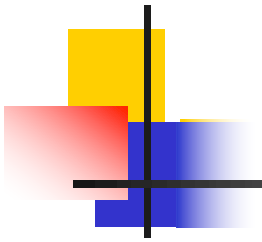
(1) Consider a function safetail that behaves in the same way as tail, except that safetail maps the empty list to the empty list, whereas tail gives an error in this case. Define safetail using:

(a) a conditional expression;
(b) guarded equations;
(c) pattern matching.

Hint: the library function null :: [a] → Bool can be used to test if a list is empty.

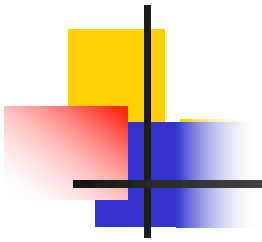
Solve all of these!!!

55



Outline

- *Basic types*
- *Lists and tuples*
- *Function types and currying*
- *Type classes*
- *Defining functions*
  - *Pattern matching*
  - *Guarded equations*
  - *Lambda expressions*
- *Recursive functions*

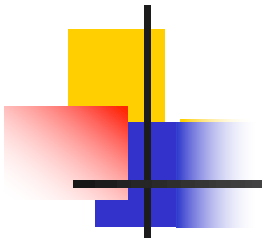


Recursive Functions

In Haskell, functions are often recursive. E.g.:

```
fac 0 = 1
fac n = n * fac (n-1)
```

fac maps 0 to 1, and any other integer to the product of itself and the factorial of its predecessor.



Recursion on Lists

Recursion is not restricted to numbers; it can also be used to define functions on lists

```
product :: Num a ⇒ [a] → a
product []     = 1
product (n:ns) = n * product ns
```

product maps the empty list to 1, and any non-empty list to its head multiplied by the product of its tail.

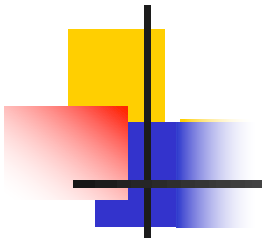
More Functions on Lists

```
length' :: [a] → Int
length' []     = 0
length' (_:xs) = 1 + length' xs
```

- sumByTwo. E.g.,

```
>sumByTwo [1,2,3,4]
>[3,7]
```

```
sumByTwo :: ?
sumByTwo [] = ?
sumByTwo ? = ?
```



Multiple Arguments

Functions with more than one argument can also be defined using recursion. For example:

- Zipping the elements of two lists:

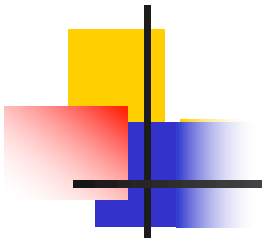
```
zip :: [a] → [b] → [(a,b)]
zip []     _      = []
zip _      []     = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

- Remove the first n elements from a list:

```
drop :: ?
```

- Appending two lists:

```
(++) :: [a] → [a] → [a]
```



Exercises

(1) Define the following library functions using recursion:

- Decide if all logical values in a list are true:

```
and :: [Bool] → Bool
```

- Concatenate a list of lists:

```
concat :: [[a]] → [a]
```

- Produce a list with n identical elements:

```
replicate :: Int → a → [a]
```

- Select the nth element of a list:

```
(!!) :: [a] → Int → a
```

- Decide if a value is an element of a list:

```
elem :: Eq a ⇒ a → [a] → Bool
```

(2) Define a recursive function

```
merge :: Ord a ⇒ [a] → [a] → [a]
```

that merges two sorted lists of values to give a single sorted list. For example:

```
> merge [2,5,6] [1,3,4]
[1,2,3,4,5,6]
```

(3) Define a recursive function

```
msort :: Ord a ⇒ [a] → [a]
```