FP101x - Functional Programming

0. Introduction

* The software crisis
  - → How can we cope with the size and complexity of modern computer programs?
  - → How can we reduce the time and cost of development
  - → How can we increase our confidence that the programs work correctly

* Programming Languages

One approach - design languages that
- → Allow programs to be written clearly, concisely and with a high level of abstraction
- → Support reusable components
- → Encourage use of formal verification
- → Permit rapid prototyping

Functional languages provide an elegant framework with these ideas in mind

* What is a functional language?

- → Functional programming is a style of programming in which the method of computation is the application of functions to arguments
- → Functional languages support and encourage the functional style of programming
  N.B. Support for lambda expressions

Example: Summation in a loop vs recursive function

History

1930s → Alonso Church develops lambda calculus
1950s → John McCarthy develops Lisp, the first modern functional language but retaining imperative features
1960s → Peter Landin develops ISWIM (If you see what I Mean) the first pure functional language
1970s → John Backus (one of the developers of Fortran) develops FP which emphasises higher-order functions and reasoning about the correctness of code
→ Robin Milner and co. develop ML which introduced type inference and polymorphic types (generics)
1980s → David Turner develops lazy functional languages → Miranda System
1987 → International Committee initiates development of Haskell - standard lazy functional language
2003 → Haskell 98 report - stable version of Haskell
→ Std distn, lib support, new features, dev tools, use in industry, influence on other languages

Example: A taste of Haskell

```
f []     = []
f (x:xs) = f ys ++ [x] ++ f 3s
           where
              ys = [a | a ← xs, a ≤ x]
              3s = [b | b ← xs, b > x]
```

1. First Steps

* Watch Karate Kid! Wax on, wax off!

* Glasgow Haskell Compiler

  GHC is the leading implementation, comprises a compiler and interpreter great for prototyping

  www.haskell.org/platform

* ghci at the cmd pmt starts the REPL (Read Eval Print Loop)

* Std Prelude

  - sqrt
  - head [1,2,3] = 1
  - tail [1,2,3] = [2,3]
  - take 3 [1,2,3,4] = [1,2,3]
  - [1,2,3] !! 2 = 3 ← (2nd element of list indexing starts at zero) time linear op.
  - drop 3 [1,2,3,4] = [4]
  - length [1,2,3,4] = 4 time linear op.
  - sum [1,2,3,4] = 10
  - product [1,2,3] = 6
  - [1,2,3] ++ [4,5] = [1,2,3,4,5]
  - reverse [1,2,3] = [3,2,1]

* Function application

  Mathematics → Haskell

  f(a,b) + cd → f a b + c*d

  white space replaces parentheses in application of funcn to args

f a + b means f(a) + b

funcn op. take priority over all other op.

Yes this is wrong! He corrects it later in the course when he talks about curried funcns see page 9 in these notes!

Mathematics	Haskell
f(x)	f x
f(x,y)	f x y
f(g(x))	f (g x)
f(x, g(y))	f x (g y)
f(x) g(y)	f x * g y

more beautiful?! less typing

* Haskell scripts

Can define your own functions in a Haskell script (.hs) text file

↑ not mandatory but useful

Useful to have the REPL and your script file open simultaneously

At the cmd pmt : > ghci test.hs

→ load & compiles the file and opens a REPL where you can use it! COOL!

binary function (like +) makes a list

back quotes

```
factorial n = product [1..n]
average ns = sum ns `div` length ns
```

x `f` y syntactic sugar for f x y

> :reload in the REPL just recompiles test.hs

* Naming requirements and conventions

Function and argument names must begin with lower case letter, e.g.

myFun , fun1 , arg_2 , x'

can include numbers, underscores, back quotes

By convention list args usually have an s suffix on their name, e.g.

xs ns nss ← list of lists

White space (like in Python) is significant so there is the Layout Rule

→ each defn must begin in the same column
→ avoids the need for curly parenthesis to indicate groupings of defns, e.g.

```
a = b + c
    where
      { b = 1 ;
        c = 2 }
d = a * 2
```

don't need red stuff

* Useful GHCi commands

Command	Meaning
:load scriptname.hs	
:reload	reload current script
:edit scriptname.hs	
:edit	edit current script
:type expr or :t expr	show type of expr
:?	show all cmds
:quit	quit GCHi

N.B. There's no space between : and the first letter in these

* FP complete is like an online version of Worksheets in Scala but for Haskell
https: www.fpcomplete.com/page/project-build

✱ Lab notes - GCHi interpreter

- The last evaluated value is always available for further computation under the name 'it'.
- Infix notation and Prefix notation
  2+2 (+) 2 2
- Booleans : True and False
- Strings : "Hello"
- Logical op's : || , && , not
  N.B. no need for parent'eses with not
- Args and func^{n}s start with a lower case by convention, data constructors start with an upper case letter e.g. True, False (more later)
- Priority of logical op's : not, &&, ||
- Concatenation: ++ e.g. "Hello" ++ " " ++ "world"
- String is a List of Chars
- Some functions for List:

length	-	number elts
head	-	first elt
tail	-	first elt removed
last	-	last elt
init	-	last elt removed
reverse	-	reverse list
null	-	is the list empty?

- head "" causes an exception i.e. a DYNAMIC ERROR. Parse errors and scoping errors are STATIC ERRORS.
  Dynamic errors occur at runtime
  Static errors occur at compile time
- Parse errors - lexical error, type error
- True :: Bool means True has type Bool
- "Hello" :: [Char] i.e. List with elts of type Char
  i.e. "Hello" = ['H', 'e', 'l', 'l', 'o']

- not :: Bool -> Bool
- length :: [a] -> Int (polymorphic type)
- All elts of a List must have the same type, type checker will check all later elts against the first
- Tuples can have elts of different types
  e.g. (1, "Hello")
  and they have a fixed length i.e. it is part of the type...
  (1,2) :: (Num t1, Num t) => (t, t1)
  (1, "Hello") :: Num t => (t, [Char])
  } type classes I guess these will come later
  Left of => are just deps so order doesn't matter I think ?!
- There are some funcns specifically for pairs
  fst :: (a,b) -> a
  snd :: (a,b) -> b

2. Types & Classes

* A type is a name for a collection of related values

* Applying a funct^{n} to one or more args of the wrong type is called a type error

Haskell is a statically typed language (type checking is done at compile time) as oppose to dynamically typed (where it is done at run time) making it safer and faster!

if 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 type inference.

* Basic types:

Bool, Char, String, Int, Integer, Float

[Char] → String; Int → Fixed precision 32/64 bit; Integer → Arbitrary precision

List is a polymorphic type. Unlike tuples the size is not part of the type:

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

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

* A function is a mapping from values of one type to values of another type. In Haskell notn, t1 -> t2 is the type of functions that map values of type t1 to values of type t2

Example:

```
add :: (Int, Int) -> Int
add (x,y) = x + y
```

* Curried Functions

Another way of writing the above add function would be in curried form:

```
add' :: Int -> (Int -> Int)
(add' x) y = x + y
```

parentheses are not required here since the default orientation is consistent with them see N.B. below

The above says, add' takes an Int x and returns a funcn, Int -> Int, add' x. This in turn takes an Int y and returns the result x + y.

N.B. -> takes priority ordering from right to left whereas funcn application takes priority from left to right, see red parentheses above

N.B.2 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.

Functions that take their args one at a time are curried functns, named after Haskell Curry

Funcⁿs with more than one arg can be curried by returning nested funcⁿs, e.g.

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

Again you don't need parentheses here

* Why is currying useful?

Curried funcⁿs are more flexible than functions on tuples, because they allow partial application to define functions, e.g.

```
add' 1 :: Int → Int
mult 1 :: Int → Int → Int
```

And if you check, you'll find

```
take 5 :: [Int] → [Int]
drop 2 :: [Int] → [Int]
```

oops, another error, Int should all be type variables a for example. See Polymorphic functions on page 11

* Currying conventions

As mentioned before

- → associates to the right first
- funcⁿ application associates to the left first

so

Int → Int → Int means Int → (Int → Int)

and

mult x y means (mult x) y

Unless tuples are explicitly required <u>all</u> functions in Haskell are in curried form

* Polymorphic functions.

A func^{n} is called polymorphic if its type contains one or more variables

e.g. `length :: [a] -> Int`

↑ type variable, hence lower case

Type variables can be instantiated to diff types in diff circumstances

* Some polymorphic func^{n}s in Prelude:

```
fst  :: (a,b) -> a
head :: [a] -> a
take :: Int -> [a] -> [a]
zip  :: [a] -> [b] -> [(a,b)]
id   :: a -> a
```

Phil Wadler's paper 'Theorems for Free' explains how every polymorphic type gives a theorem which holds for the corresponding function.

* Overloaded functions

A polymorphic func^{n} is called overloaded if its type contains one or more constraints on the classes/types

e.g. `sum :: Num a => [a] -> a`

actually all this defines the type of sum

says for any numeric type a sum :: [a] → a

Constrained type vbles can be instantiated to any types that satisfy the constraints.

e.g. Char is not a numeric type

* Some type classes in Haskell

Num	numeric types
Eq	equality types
Ord	ordered types

```
(+)  :: Num => a -> a -> a
(==) :: Eq  => a -> a -> Bool
(<)  :: Ord => a -> a -> Bool
```

3. Defining functions

* Conditional expressions

As in most languages, funcⁿs can be defined using conditional expressions

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

They can be nested

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

In Haskell, condⁿal expressions must always have an else branch to avoid any ambiguity

* Guarded eqⁿs

As an alternative to condⁿals, funcⁿs can also be defined using guarded eqⁿs

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

which make nested condⁿals easier to read

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

N.B. The catch all condⁿ otherwise is defined in Prelude by otherwise = True

* Pattern matching

Many funcⁿs have particularly clear defⁿs using pattern matching on their arguments

```
not :: Bool -> Bool
not True  = False
not False = True
```

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
```

and further more efficiently by

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

since it doesn't evaluate the second arg if the first arg is false

N.B. Underscore is a wildcard pattern that matches any arg

N.B.2 Patterns are matched in order so

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

would always return False

N.B.3 Patterns may not repeat vbls. E.g.

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

would return an error.

b is a dummy vbl, it doesn't make sense to repeat it in the args of a binary func^n

* List Patterns

Internally, every non-empty list is constructed by repeated application of the (:) operator aka 'cons' that adds an elt to the start of a list

[1,2,3,4] means 1:(2:(3:(4:[])))

* Functions can be defined using x:xs patterns

e.g.

```
head :: [a] -> a
head (x:_) = x

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

N.B. x:xs only pattern matches to non-empty lists so head [] would give an error

N.B.2 x:xs patterns must be parenthesised in function defns because func^n app^n has priority over (:)

* Lambda Expressions

Funcⁿs can be constructed without naming them by using lambda expressions

E.g. $\lambda x \rightarrow x + x$
is equivalent to the nameless funcⁿ
$x \longmapsto x + x$
in mathematical notⁿ

N.B. λ is typed using a backslash on the keyboard

N.B.2 The use of λ comes from lambda calculus - the theory of funcⁿs on which Haskell is based.

* Why are lambdas useful?

Often one wants to avoid naming funcⁿs that are only referenced once

Example:
```
odds n = map f [0..n-1]
         where
           f x = x*2 + 1
```
can be simplified to

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

* Lambdas provide the formal meaning of funcⁿs defined using currying, e.g.
```
add x y = x + y
```
is syntactic sugar for
```
add = \x -> (\y -> x+y)
```

* Lambda expressions can also make it more explicit when defining funcⁿs that return funcⁿs as results, e.g.

```
const :: a -> b -> a
const x _ = x
```

can be defined also as

```
const :: a -> (b -> a)
const x = λ_ -> x
```

* Sections

An operator written between its two args can be converted into a curried funcⁿ written before its two args by using parentheses

Example: 1 + 2

can be written		(+) 1 2
"	"	(1+) 2
"	"	(+2) 1

In general, if $\oplus$ is an operator then funcⁿs of the form $(\oplus)$, $(x\oplus)$ and $(\oplus y)$ are called sections.

* Why are sections useful?

Because they allow us to construct useful functions in a simple and concise manner

Example sections :

(1+) - successor
(1/) - reciprocal
(*2) - double
(/2) - halve

4 List Comprehensions

* Set comprehensions in mathematics can be used to construct new sets from old, e.g.

$$\{x^2 \mid x \in \{1,2,\ldots,5\}\} = \{1,4,9,16,25\}$$

List comprehensions in Haskell can be used similarly:

```
[x^2 | x <- [1..5]] = [1,4,9,16,25]
```

N.B. The expression `x <- [1..5]` is called a generator as it states how to generate values for `x`

Comprehensions can have multiple generators, e.g.

```
[(x,y) | x <- [1,2,3], y <- [4,5]]
  = [(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

Changing the order of the generators, changes the order of the elements in the final list, e.g.

```
[(x,y) | y <- [4,5], x <- [1,2,3]]
  = [(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
```

Multiple generators are like nested loops, later generators are more deeply nested

* Dependent generators

Later generators can depend on variables introduced in earlier generators, e.g.

```
[(x,y) | x ← [1..3], y ← [x..3]]

= [(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

Example :

- Concatenate a list of lists :

```
concat :: [[a]] → [a]
concat xss = [x | xs ← xss, x ← xs]
```

```
concat [[1,2,3],[4,5]] = [1,2,3,4,5]
```

* Guards

List comprehensions can use guards to restrict the values produced by earlier generators, e.g.

```
[x | x ← [1..10], even x] = [2,4,6,8,10]
```

Examples :

- Function which maps a +ve integer to a list of its factors

```
factors :: Int → [Int]
factors n = [x | x ← [1..n], n `mod` x == 0]
```

```
factors 15 = [1,3,5,15]
```

- Function which decides if a number is prime

```
prime :: Int -> Bool
prime n = factors n == [1,n]
```

- Function which returns the list of all primes upto a given limit

```
primes :: Int -> [Int]
primes n = [x | x <- [2 ], prime x ]
```

```
primes 40 = [2,3,5,7,11,13,17,19,23,29,31,37]
```

* The zip function

Function which maps two lists to a list of pairs of their corresponding elements

```
zip :: [a] -> [b] -> [(a,b)]
```

e.g. `zip ['a','b','c'] [1,2,3,4]`

`= [('a',1),('b',2),('c',3)]`

- Using zip we can define a function which returns the list of all pairs of adjacent elts from a list

```
pairs :: [a] -> [(a,a)]
pairs xs = zip xs (tail xs)
```

```
pairs [1,2,3,4] = [(1,2),(2,3),(3,4)]
```

- Using pairs we can define a function which decides if a given list is sorted e.g.

```
sorted :: Ord a => [a] -> Bool
sorted xs = and [x<=y | (x,y) <- pairs xs]
```

[Bool]

- Using zip we can define a function that returns a list of all the positions of a value in a list

```
positions :: Eq a => a -> [a] -> Int
positions x xs = [i | (x',i) <- zip xs [0..n], x==x']
                 where n = length xs - 1
```

```
positions 0 [1,0,0,1,0,1,1,0] = [1,2,4,7]
```

* String comprehensions

Recall that in Haskell a String is just [Char] i.e. a list of characters

```
"abc" :: String  means  ['a','b','c'] :: [Char]
```

So, any polymorphic func^{n} that operates on lists can also be applied to strings

Examples:

- `length "abcde" = 5`
- `take 3 "abcde" = "abc"`
- `zip "abc" [1,2,3,4] = [('a',1),('b',2),('c',3)]`

We can use list comprehensions to define funcⁿs on strings

Example : A funcⁿ that counts the lower case letters in a string

```
lowers :: String -> Int
lowers xs = length [x | x ← xs, is ower x]

lowers "Haskell" = 6
```

5 Recursive Functions

We have seen, many funct^ns can be naturally defined in terms of other funct^s

Example: Factorial

```
factorial :: Int → Int
factorial n = product [1..n]
```

Funct^ns are evaluated by a stepwise process of applying funct^s to their args:

```
factorial 4 = product [1..4]
            = product [1,2,3,4]
            = 1*2*3*4 = 24
```

Factorial - Recursive version

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

defn diverges for -ve n because the base case is never reached → stack overflow!

```
factorial 3 = 3 * factorial 2
            = 3 * (2 * factorial 1)
            = 3 * (2 * (1 * factorial 0))
            = 3 * (2 * (1 * 1))
            = 3 * (2 * 1)
            = 3 * 2 = 6
```

* Why is recursion useful?

Some funct^s such as factorial are simpler to define in terms of other funct^s

Many funcns can naturally be defined in terms of themselves

Properties of funcns defined using recursion can be proved using induction

* Recursion on lists

Recursion can be used to define funcns on lists

-
```
product :: [Int] → Int
product [] = 1
product (n:ns) = n * product ns
```

```
product [2,3,4] = 2 * product [3,4]
                = 2 * (3 * product [4])
                = 2 * (3 * (4 * product []))
                = 2 * (3 * (4 * 1))
                = 2 * (3 * 4)
                = 2 * 12 = 24
```

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

```
length [1,2,3] = 1 + length [2,3]
               = 1 + (1 + length [3])
               = 1 + (1 + (1 + length []))
               = 1 + (1 + (1 + 0))
               = 1 + (1 + 1)
               = 1 + 2 = 3
```

-
```
reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

```
reverse [1,2,3] = reverse [2,3] ++ [1]
                = (reverse [3] ++ [2]) ++ [1]
                = ((reverse [] ++ [3]) ++ [2]) ++ [1]
                = (([] ++ [3]) ++ [2]) ++ [1]
                = ([3] ++ [2]) ++ [1]
                = [3,2] ++ [1] = [3,2,1]
```

* Multiple arguments

Functns with more than one arg can also be defined using recursion

Examples:

- Zipping the elts of two lists

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

- Remove the first n elts from a list

```
drop :: Int -> [a] -> [a]
drop 0 xs = xs
drop _ [] = []
drop n (_:xs) = drop (n-1) xs
```

- Appending two lists

```
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

* Quicksort

```
qsort :: [Int] -> [Int]
qsort [] = []
qsort (x:xs) = qsort smaller ++ [x] ++ qsort larger
    where smaller = [a | a <- xs, a <= x]
          larger = [b | b <- xs, b > x]
```

```
q [3,2,4,1,5] = q [2,1] ++ [3] ++ q [4,5]
              = (q [1] ++ [2] ++ []) ++ [3]
                ++ (q [] ++ [4] ++ q [5])
              = (([] ++ [1] ++ []) ++ [2] ++ []) ++ [3]
                ++ ([] ++ [4] ++ ([] ++ [5] ++ []))
              = ([1] ++ [2] ++ []) ++ [3]
                ++ ([] ++ [4] ++ [5])
              = [1,2] ++ [3] ++ [4,5]
              = [1,2,3,4,5]
```

Exercises:

Decide if all logical values in a list are true

```
and :: [Bool] -> Bool
and [] = True
and (False : xs) = False
and (True : xs) = and xs
```

- Concatenate a list of list

```
concat :: [[a]] -> [a]
concat [] = []
concat (xs : xss) = xs ++ concat xss
```

- Produce a list with n identical elements

```
replicate :: Int -> a -> [a]
replicate 0 _ = []
replicate n x = [x] ++ replicate (n-1) x
```

- Select the nth elt of a list

```
(!!) :: [a] -> Int -> a
(x : _) !! 0 = x
(_ : xs) !! n = xs !! (n-1)
```

- Decide if a value is an elt of a list

```
elem :: Eq a => a -> [a] -> Bool
elem _ [] = False
elem x (y : ys)
    | x == y = True
    | otherwise elem x ys
```

- Define a recursive func^{n} that merges two sorted list of integers to give a single sorted list

```
merge :: [Int] -> [Int] -> [Int]
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys) | x ≤ y = x : merge xs (y : ys)
                    | otherwise = y : merge (x xs) ys
```

- Define a recursive func that implements merge sort as specified by the rules
  - Lists of length ≤ 1 are already sorted;
  - Other lists can be sorted by sorting the two halves and merging the resulting lists

```
msort :: [Int] → [Int]
msort [ ] = [ ]
msort [x] = [x]
msort xs = merge (msort ys) (msort zs)
   where (ys, zs) = halve xs

halve :: [a] → ([a], [a])
halve xs = splitAt (length xs `div` 2) xs
```

6. Higher Order Functions

* A funcⁿ is called higher order if it takes a funcⁿ as an arg or returns a funcⁿ as a result.

e.g.

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

is a higher order funcⁿ because it takes a funcⁿ as an arg.

* Why are higher order funcⁿs useful?

→ Common programming idioms can be encoded as funcⁿs in the language itself allowing us to avoid repetition
→ Domain specific languages can be defined as collections of higher order funcⁿs
→ Algebraic properties of higher order funcⁿs can be used to reason about programs

* The Map Functⁿ

The higher order lib funcⁿ map applies a funcⁿ to every elt of a list

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

e.g. `map (+1) [1,3,5,7] = [2,4,6,8]`

Map funcⁿ can be defined using list comprehension i.e.

```
map f xs = [f x | x <- xs]
```

Alternatively, for the purpose of proofs, the map func^n can be defined recursively i.e.

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

* The Filter Func^n

The higher order lib func^n filter selects every elt from a list that satisfies a predicate

```
filter :: (a → Bool) → [a] → [a]
```

e.g. `filter even [1..10] = [2,4,6,8,10]`

Again, filter can be defined using list comprehension

```
filter p xs = [x | x ← xs, p x]
```

and alternatively using recursion

```
filter p []     = []
filter p (x:xs)
  | p x         = x : filter p xs
  | otherwise   = filter p xs
```

* The foldr func^n

A number of func^{ns} on lists can be defined using the following recursive pattern

```
f []     = v      ← maps the empty list to some value v
f (x:xs) = x ⊕ f xs
             ↑ some binary func$^n$
```

Examples:

- ```
  sum [] = 0
  sum (x:xs) = x + sum xs
  ```
  } v = 0, ⊕ = +

- ```
  product [] = 1
  product (x:xs) = x * product xs
  ```
  } v = 1, ⊕ = *

- ```
  and [] = True
  and (x:xs) = x && and xs
  ```
  } v = True, ⊕ = &&

Note, these are all instances of foldr, i.e.

```
sum = foldr (+) 0

product = fold (*) 1

and = foldr (&&) True
```

and another example:

```
or = foldr (||) False
```

* foldr itself can be defined using recursion

(← recall default priority →)

```
foldr :: (a → (b → b)) → (b → ([a] → b))
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```

however it's more intuitive to just think of foldr as replacing each (:) in a list by a given function and [] by a given value and computing from <u>right</u> to left
```

e.g. sum [1,2,3]
= foldr (+) 0 [1,2,3]
= foldr (+) (1:(2:(3:[]))) ← replace : with +, replace [] with 0
= 1+(2+(3+0)) = 6

e.g. product [1,2,3]
= foldr (*) 1 [1,2,3]
= foldr (*) (1:(2:(3:[]))) ← replace : with *, replace [] with 1
= 1*(2*(3*1)) = 6

* foldr encapsulates a simple pattern of recursion. It can be used to define many more funcns than one might expect

Examples

- Length function:

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs
```

e.g. length [1,2,3]
= length (1:(2:(3:[]))) ← replace : with (λ_ n -> 1+n), replace [] with 0
= 1+(1+(1+0)) = 3

i.e. length = foldr (λ_ n -> 1+n) 0

- Reverse function:

```
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

e.g. reverse [1,2,3]
= reverse (1 : (2 : (3 : [ ]))) ← replace : with λx xs → xs++[x]; replace [] with []
= (([ ] ++ [3]) ++ [2]) ++ [1])
= [3,2,1]

i.e. reverse = foldr (λ x xs → xs ++ [x]) [ ]

- Append function: (++) has a particularly compact defn using foldr

(++ ys) = foldr (:) ys (pointfree style)

N.B. (++ ys) xs = xs ++ ys = (++) xs ys

✱ Why is foldr useful?

→ Many recursive funcns are simpler to define using foldr
→ Properties of funcns defined using foldr can be proved using algebraic properties of foldr, such as fusion and the banana split rule
→ Advanced optimisations can be simpler if foldr is used in place of explicit recursion

✱ Other library functions

The library function (.) returns the composition of two funcns as a single funcn

(.) :: (b → c) → ((a → b) → (a → c))   (f :: b → c, g :: a → b; recall prioritisation)
f . g = λx → f (g x)

Example: odd :: Int → Bool
odd = not . even ← pointfree style

* The lib func^n all decides if every elt of a list satisfies a given predicate

```
all :: (a -> Bool) -> [a] -> Bool
all p xs = and [p x | x <- xs]
```

e.g. all even [2,4,6,8] = True

* The lib func^n any decides if at least one elt of a list satisfies a given predicate

```
any :: (a -> Bool) -> [a] -> Bool
any p xs = or [p x | x <- xs]
```

e.g. any isSpace "abc def" = True

* The lib func^n takeWhile selects elements from a list while the predicate is satisfied

```
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p [] = []
takeWhile p (x:xs)
    | p x          = x : takeWhile p xs
    | otherwise    = []
```

e.g. takeWhile isAlpha "abc def" = "abc"

* The lib func^n dropWhile removes elts while a predicate is satisfied

```
dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile p [] = []
dropWhile p (x:xs)
    | p x       = dropWhile p xs
    | otherwise = x:xs
```

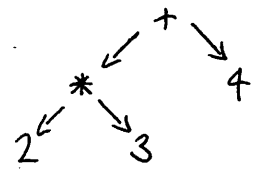
e.g. dropWhile isSpace " abc" = "abc"

7. Functional Parsers & Monads

* What is a parser?

A program that analyses a piece of text to determine its sytactic structure e.g.

2*3+4 means



* Where are they used?

Almost every real life program uses some form of parser to pre-process its input

GHC	parses →	Haskell programs
UNIX		Shell scripts
Explorer		HTML documents

* The parser type

In a functional language such as Haskell, parsers can naturally be viewed as functions

```
type Parser = String -> Tree
```

But, it may not require all its input so we can return unused input

```
type Parser = String -> (Tree, String)
```

A string might be parsable in many ways or none

so we might generalise to a list of results

```
type Parser = String -> [(Tree, String)]
```

and finally a parser might not always produce a tree so we generalise to a value of any type

```
type Parser a = String -> [(a, String)]
```

N.B. We will consider parsers that either fail and return an empty list or succeed and return a singleton list.

* Basic Parsers

Examples:

->
```
item :: Parser Char
item = λinp -> case inp of
    []     -> []
    (x:xs) -> [(x, xs)]
```
fails if input is empty
consumes the first character otherwise

->
```
failure :: Parser a
failure = λinp -> []
```
always fails

->
```
return :: a -> Parser a
return v = λinp -> [(v, inp)]
```
always succeeds returning the value v without consuming any input

```
(+++) :: Parser a -> Parser a -> Parser a
p +++ q = λinp -> case p inp of
                    []       -> parse q inp
                    [(v,out)] -> [(v,out)]

parse :: Parser a -> String -> [(a, String)]
parse p inp = p inp
```

Here the parser p +++ q behaves as parser p if it succeeds, and as parser q otherwise

In GHCi:

```
•> parse item "" = []
•> parse item "abc" = [('a', "bc")]
•> parse failure "abc" = []
•> parse (return 1) "abc" = [(1, "abc")]
•> parse (item +++ return 'd') "abc" = [('a', "bc")]
•> parse (failure +++ return 'd') "abc" = [('d', "abc")]
```

N.B. The Parser type is a monad, a mathematical structure that has proved useful for modelling many different types of computations

* Sequencing

A sequence of parsers can be combined as a single composite parser using the keyword do.

String -> [((Char, Char), String)]

Example:

```
p :: Parser (Char, Char)
p = do x ← item
       item
       y ← item
       return (x, y)
```

N.B. → Each parser must begin in the same column, i.e. the layout rule applies

→ The values returned by intermediate parsers are discarded by default but if required can be named using the ← operator

→ The value returned by the last parser is the value returned by the sequence as a whole

→ If any parser in a sequence of parsers fails, then the sequence as a whole fails, e.g.

```
> parse p "abcdef" = [(('a','c'), "def")]
> parse p "ab" = []
```

→ The do notation is not specific to the Parser type, but can be used with any monadic type.

* Derived Primitives

Parsing a character that satisfies a predicate:

```
                                  String → [(Char, String)]
sat :: (Char → Bool) → Parser Char
sat p = do x ← item
           if p x then
              return x
           else
              failure
```

Parsing a digit and specific characters:

```
digit :: Parser Char
digit = sat isDigit

char :: Char → Parser Char
char x = sat (x ==)
```

Applying a parser zero or more times:

```
many :: Parser a -> Parser [a]
many p = many1 p +++ return []
```

Applying a parser one or more times:

String -> [(a, String)] String -> [([a], String)]

```
many1 :: Parser a -> Parser [a]
many1 p = do v <- p
             vs <- many p
             return (v:vs)
```

Parsing a specific string of characters

```
string :: String -> Parser String
string [] = return []
string (x:xs) = do char x
                   string xs
                   return (x:xs)
```

Example:

We can now define a parser that consumes a list of one or more digits from a string:

```
p :: Parser String
p = do char '['
       d <- digit
       ds <- many (do char ',' digit)
       char ']'
       return (d:ds)
```

For example:

```
parse p "[1,2,3,4]" = [("1234", "")]

parse p "[1,2,3,4" = []
```

N.B. More sophisticated parsing libraries can indicate and/or recover from errors in the input string.

* Arithmetic Expressions

Consider a simple form of expressions built up from single digits using operations of addition + and multiplication *, together with parentheses

We also assume that

- * and + associate to the right
- * has higher priority than +

Formally the syntax of such expressions is defined by the following context free grammar:

```
expr   → term '+' expr | term
term   → factor '*' term | factor
factor → digit | '(' expr ')'
digit  → '0' | '1' | ... | '9'
```

However, for reasons of efficiency, it is important to factorise the rules for expr and term:

```
expr → term ( '+' expr | ε )
term → factor ( '*' term | ε )
```

ε denotes the empty string

It is now easy to translate the grammer into a parser that evaluates expressions, by simply rewriting the grammar rules using parsing primitives

```
expr :: Parser Int
expr = do t ← term
          do char '+'
             e ← expr
             return (t + e)
          +++ return t

term :: Parser Int
term = do f ← factor
          do char '*'
             t ← term
             return (f * t)
          +++ return f

factor :: Parser Int
factor = do d ← digit
            return (digitToInt d)
         +++ do char '('
                e ← expr
                char ')'
                return e
```

finally we depne

```
eval :: String → Int
eval xs = fst (head (parse expr xs))
```

Examples :

eval "2 * 3 + 4" = 10

eval "2 * (3 + 4) = 14